BUFFER OVERFLOW ATTACK

Akshay Kaikottil
Johns Hopkins University, Information security Institue
Baltimore, MD
akaikot1@jh.edu

***Abstract*— The task was to reverse engineer a given binary and exploit the buffer overflow vulnerability present in it. We were using Ghidra to analyze the binary. We initially, idebtified the vulnerable function in the binary. We then utilized Eclipse to debug and carry out the stack smashing attack. Lastly, we have attached key screenshots to our task.**

Keywords—Buffer overflow, stack smashing, vulnerability.

I. PROJECT OVERVIEW

The project included performing various reverse engineering tasks on the binary file. The binary file is first analyzed using Ghidra. We were able to accomplish this by doing the following:

- Identifying the various functions used by the binary
- Identify the vulnerable function
- Identify the backdoor function
- Identify the design flaw in the program

We were able to visualize the function call graph which helped us identify the vulnerable function and the backdoor functions. This helped us in understanding how to discover the vulnerabilities using reverse engineering methodology.

After identifying the vulnerability, we ran the binary in debug mode in Eclipse. This helped us to discover the various parameters needed to craft a string to exploit the vulnerability.

II. BINARY OVERVIEW AND PREPERATION

We begin our task by installing the required libraries and files required for the binary to function. We were able to accomplish this by doing the following:

- sudo apt-get install libncurses-dev libncurses6 libncursesw6
- mkdir /usr/share/adventure-qnx/
- sudo cp ./advent*.txt /usr/share/adventure-qnx/
- ./binary3

The given binary executes a CLI game.

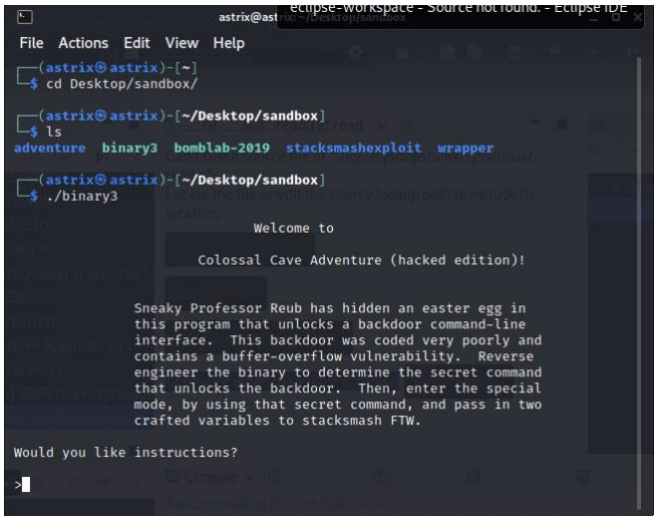

Figure 2.1: Testing the binary

III. PROJECT EXECUTION

After Ghidra analyses the binary, by going through the SymbolTree a vulnerable memory function is found that this software uses. This function is found to be "memcpy", a vulnerable C function that is being used by this application.

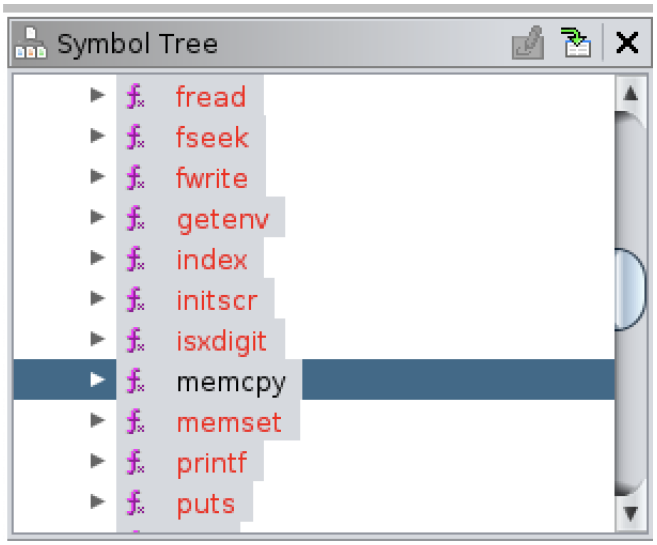

Figure 3.1: Functions called in our program

Using this knowledge, the vulnerable function call that calls this function can be observed and a Functon Call Graph can be plotted correspondingly.

We then identify the vulnerable function:

```
Decompile: Vulnerable_function - (...
1
2  undefined8 Vulnerable_function(void)
3
4  {
5    int lenth_of_copy;
6    undefined4 local_1b4;
7    undefined src_buffer [256];
8    char Dest_buffer [168];
9
10   sprintf(Dest_buffer,"v3");
11   local_1b4 = FUN_00106fc0(&lenth_of_copy,sr
12   memcpy(Dest_buffer,src_buffer,(long)lenth_
13   return 0;
14 }
15
```

Figure 3.2: Vulnerable function

We then plot the function call graph of the vulnerable function.

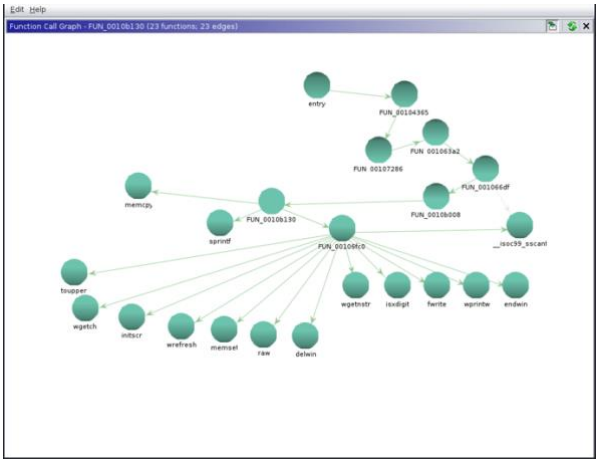

Figure 3.3: Function call of vulnerable function

By going through the Function Call Graph, it is identified that there is backdoor function that calls out this newly identified vulnerable function.

```
Decompile: FUN_001066df - (binary3)
1
2  void FUN_001066df(void)
3
4  {
5    int iVar1;
6    char local_68 [88];
7    char *local_10;
8
9    fputc(0x3e,stdout);
10   DAT_00115710 = 0;
11   DAT_00115250 = 0;
12   fgets(local_68,0x50,stdin);
13   local_10 = local_68;
14   while( true ) {
15     iVar1 = tolower((int)*local_10);
16     *local_10 = (char)iVar1;
17     if (*local_10 == '\0') break;
18     local_10 = local_10 + 1;
19   }
20   __isoc99_sscanf(local_68,"%19s %19s",&DAT_00115250,&DAT_00115710);
21   if (DAT_0011524c != 0) {
22     printf("WORD1 = %s, WORD2 = %s\n",&DAT_00115250,&DAT_00115710);
23   }
24   iVar1 = strncmp(&DAT_00115250,"unlock",6);
25   if (iVar1 == 0) {
26     iVar1 = strncmp(&DAT_00115710,"easteregg",9);
27     if (iVar1 == 0) {
28       FUN_0010b008();
29     }
30   }
31   return;
32 }
33
```

Figure 3.4: The backdoor function

It is seen that the backdoor function can be accessed by the easter egg code "unlock easteregg". This can be used to access the vulnerable function and exploit the memcpy function call.

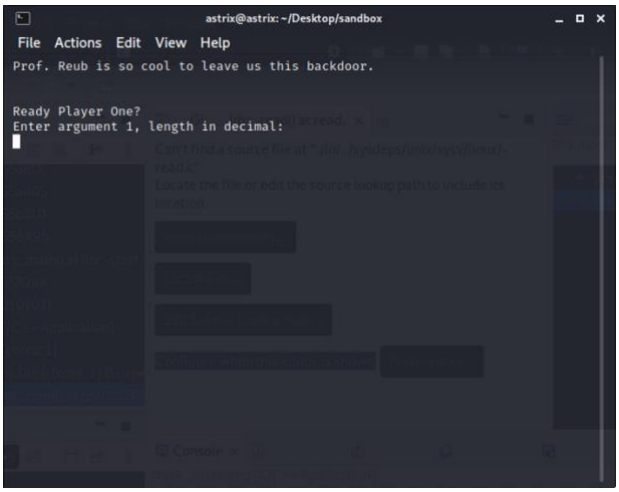

Figure 3.5: Accessing the backdoor function

We then setup the debug environment in Eclipse. The GDB command file, ~/.gdbinit is set with the following content:

set disassembly-flavor intel

set disable-randomization on

We turn off Address Space Randomization off with the following command:

sudo sysctl kernel.randomize_va_space=0

Now the debug configurations for the project is set. This is done by providing the application path in the debug configurations.

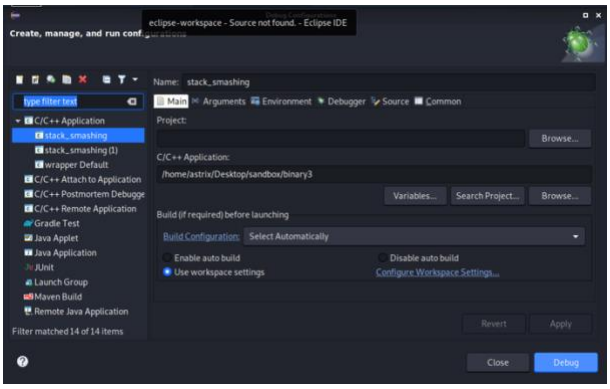

Figure 3.6: Setting up debug environment

Upon debugging the disassembly can be viewed. The debugger can be paused to analyze the disassembly. The easter egg code "unlock easteregg" that was found by reverse engineering can be used to access the vulnerable function. A break point was placed where the memcpy function is being called.

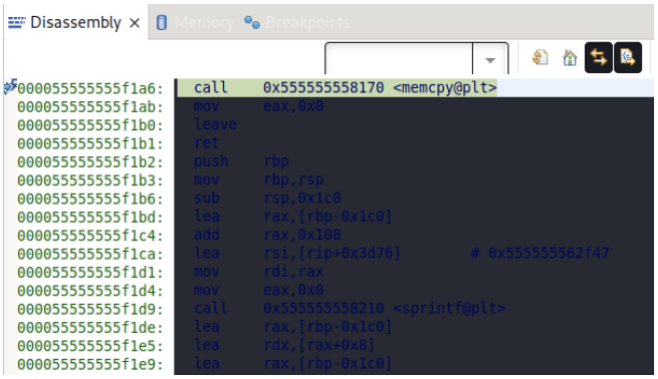

Figure 3.7: Setting up break point

Use instruction stepping mode to analyze the memory for the register $rsp after providing inputs for the vulnerable function. Initially a small length of 32 is provided as the first input. Correspondingly input of 32 \x12 is provided for argument2. This input can be seen in the memory browser of $rsp. We find that the input is being written to the memory at 0x7FFFFFFFDC48.

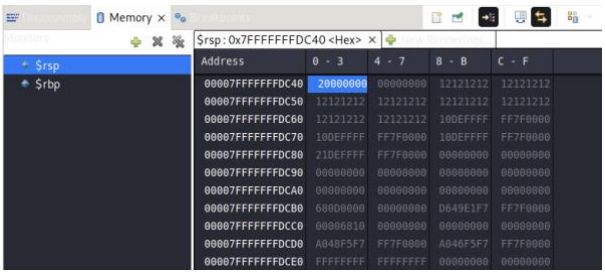

Figure 3.8: Input value being populated

With this knowledge a shellcode can be constructed, the C code stacksmashexploit.c can be used or a hexadecimal string consisting of shellcode in the middle and the return address of the shellcode present at the end should be provided.

The format will be similar to <NOP sled> <ShellCode><more NOPs> <return address of shellcode> <valid return address>. The NOPs are placed such that the $rbp register will point towards the shellcode. The length of the constructed string is chosen as 250. Where the return address is places at position 168+8.

The return adddress is provided as the beginning of the stack which is 0x7FFFFFFFDC48 where the shellcode is placed after a few NOPs.

The input generated by our exploit program is inputted to our vulnerable program and we successfully get a shell.

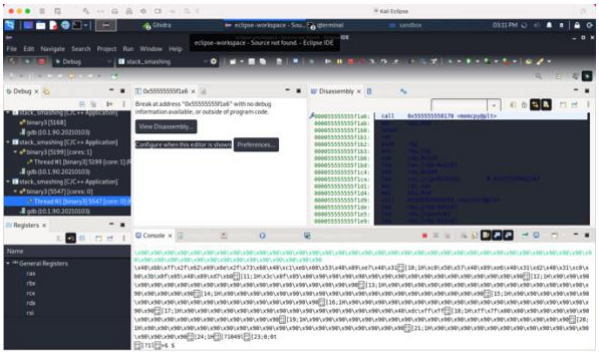

Figure 3.9: Successful generation of shell

IV. Accomplishments

We were able to understand in much more detail how to interpret the various forms in which data is presented during reverse engineering. We were then able understand in depth how to discover and conduct the buffer overflow attack.

Acknowledgment

The author wishes to thank Reuben Johnston for his support for the successful completion of Reverse Engineering the Server-Client Live Chat Service.

References

[1] Buffer overflow, https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

Key Terms

Buffer overflow: A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer.

Stack Smashing: Stack smashing is a form of vulnerability where the stack of a computer application or OS is forced to overflow.

Vulnerability: A flaw in a system that leaves information expose to a threat.