

EECS 765 Project 3

Author: Jace Kline 2881618

Introduction

The purpose of this programming assignment is to develop and execute a local heap overflow attack against an application running on RedHat 8.

Setup & Assumptions

The first step in setting up the exploit involves placing the exploit files on the target RedHat 8 machine. The files necessary to run the automated exploit include `exploit.c`, `gdbcommands.txt`, and `exploit.sh`. We assume that `exploit.sh` has executable permissions enabled. We require that the `getscore_heap.c` victim program source code and the corresponding `score.txt` file are copied to the same directory as the exploit files (or vice-versa, as long as they are all in the same directory). Since this is a local attack, we assume that `getscore_heap.c` can be compiled via `gcc` using the `-g` flag which enables debugging in `gdb`. This assumption allows us to automate the process of finding the target buffer address in the compiled victim program. Additionally, we assume that the programs `gdb`, `objdump`, `grep`, and `awk` are installed on the target machine to facilitate the automated capture of parameters.

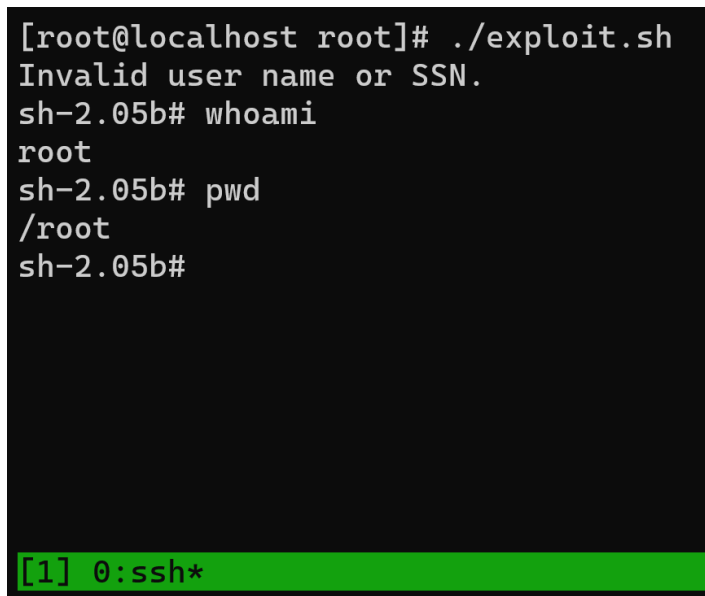
Running the Exploit

Before running the exploit, we assume that the environment is as described in the previous section. We execute the command below to run the exploit.

```
./exploit.sh
```

This script performs compilation, parameter gathering, input construction, and execution. The result of executing this script should be a shell obtained as the result of a heap overflow attack. The details of this script are discussed later in this report.

```
[root@localhost root]# ./exploit.sh
Invalid user name or SSN.
sh-2.05b# whoami
root
sh-2.05b# pwd
/root
sh-2.05b#
```

A terminal window with a black background and white text. The prompt is [root@localhost root]#. The user runs ./exploit.sh, which outputs "Invalid user name or SSN." The user then runs whoami, which outputs root. The user runs pwd, which outputs /root. The user runs another command, which outputs sh-2.05b#. At the bottom of the terminal, there is a green bar with the text [1] 0:ssh* in white.

Developing the Exploit

In this section, we talk about the general thought process used when developing the exploits. We outline the exploit structure, the specific parameters relevant to the attack, and the generation of our malicious input and wrapper script.

Malicious Input Structure

We use the victim source code `getscore_heap.c` to obtain insights as to what the input structure should look like and which buffer we should attempt to overflow. The key observation from the source code is that there is a dynamically allocated heap buffer, named `'matching_pattern'`, that is formed as the concatenation of the contents in buffer `'name'`, the string `":"`, and the contents in buffer `'ssn'`. We also know that the buffers `'name'` and `'ssn'` are derived directly from the arguments provided to the program on the command line. The size of this `'matching_pattern'` buffer is declared to be the size of the `'name'` buffer + 17. This implies that we essentially have control over this `'matching_pattern'` buffer size by controlling the `'name'` buffer input. We can also deduce that if the concatenation of `":"` and the `'ssn'` buffer is allowed 17 bytes total, then the `'ssn'` buffer itself is assumed to be 16 bytes since `":"` is only one byte.

With the findings above, we can form a strategy for how to use the `'name'` and the `'ssn'` buffer inputs to complete a heap overflow attack against the `'matching_pattern'` buffer. To stay consistent with previously studied attacks, we declare that we want the entire size of the `'matching_pattern'` buffer to be 128 bytes. This implies that the `'name'` buffer must be 111 bytes. The first 8 bytes of the `'name'` buffer are filled with garbage values `"XXXX"` and `"YYYY"` since they will be overwritten in the execution of `free()`. The next two bytes contain 6 NOPs followed by the hex bytes `EB 04` (`JMP +4`). The next word is filled with trash bytes `"ZZZZ"` since these will also be overwritten in the `free()` operation. We include the `JMP +4` instruction above to jump over these bytes to reach the shellcode. Following these bytes, we place the 45 bytes of shellcode and then append NOP padding values (`0x90`) to reach the end of the desired 111 bytes for the `'name'` buffer. Within the victim program, the `":"` (one byte) is concatenated and therefore the `'name'` buffer + `":"` makes 112 bytes and aligns on a word boundary. We now must consider the `'ssn'` buffer contents. As discussed above, the `'ssn'` buffer has 16 bytes allocated for it on the copy to the `'matching_pattern'` buffer. Hence, we can exploit this buffer to perform the overflow and append a fake `free` heap block to the end of the `'matching_pattern'` buffer. To fill the legitimate 16 bytes of this buffer, we simply use NOPs (`0x90`). The construction of the fake `free` heap block requires 16 bytes of overflow. The first 2 words of this are values `0xffffffff`. The next word corresponds to

the address of (what - 12) where 'what' is the Global Offset Table (GOT) address entry for the function free(). We choose the free() function to overwrite because it is used later in the victim program. The next 4 bytes correspond to the (where + 8) where 'where' is the address of the 'matching_pattern' buffer in memory.

We have shown above that we must use both the 'name' and 'ssn' buffers to craft a working exploit. The exploit shall be carried out by running `./getscore_heap NAME SSN` where the inputs of NAME and SSN are those described in this section.

Malicious Input Parameters

For this exploit, we require the following parameters:

- System architecture
- The length of the 'matching_pattern' buffer
- Global Offset Table (GOT) address of free()
- Address of the exploitable 'matching_pattern' buffer referenced in `getscore_heap.c`

For this attack, the RedHat 8 machine uses the x86 architecture. This implies little-endianness and a 4-byte word length.

For consistency with the sample exploit seen in class, we declare that the length of the 'matching_pattern' buffer shall be 128 bytes, which implies that the 'name' buffer shall be 111 bytes.

Our exploit requires that we overwrite an address of a function used in the victim program to hijack the execution of the program and jump to our shellcode written into the heap buffer. This is the 'what' address referenced in the exploit. By examining the source code, we choose the free() function since it is executed in the program after the call to `free(matching_pattern)`, which performs the overwrite. We use the command `objdump -R getscore_heap` to list the GOT functions and their addresses and we extract the address listed for free().

To perform the overwrite of the free() GOT entry, we must also obtain the address of the start of our injected code, which occurs at the start of the 'matching_pattern' buffer + 8 bytes. We use the `gdb` tool on the `getscore_heap` program to achieve this. We set a breakpoint after the allocation of the buffer in code and then output its address. We also automate this process in our exploit script.

Generating the Malicious Input

To generate the malicious input and perform the exploit, we have provided a wrapper script called `exploit.sh`. The contents of this script are seen below.

```
#!/bin/bash

# compile exploit.c and getscore_heap.c if they aren't already
gcc -g exploit.c -o exploit
gcc -g getscore_heap.c -o getscore_heap

# get parameters for exploit programmatically
BUF_LEN=128
GOT_FREE=$(objdump -R getscore_heap | grep free | awk '{print $1}')
BUF_ADDR=$(gdb -batch -x gdbcommands.txt 2>/dev/null | grep '$1' | awk '{print $3}')

# run the exploit with the parameters instantiated
./exploit $BUF_LEN $GOT_FREE $BUF_ADDR
```

The execution of this script involves three main steps. The first is to compile the `exploit.c` and `getscore_heap.c` files to the binary executables `exploit` and `getscore_heap`, respectively (with debugging enabled). The second is to automatically obtain the 3 parameters needed for running the compiled `exploit` executable, namely the buffer length, the GOT entry address to overwrite, and the target buffer address to overflow. The last step involves running the `exploit` executable with the obtained parameters from step 2. This exploit executable generates the input buffers to send to the victim `getscore_heap` program and runs the victim program with those buffers. This `exploit` program is a modification of the `heap_exploit` program provided by Dr. Bardas. The modifications simply involve the division of the inputs correctly between the 'name' and 'ssn' buffers as discussed earlier.

References and Collaborations

References

- [GDB Man Pages](#)

Collaborations

I did not collaborate with any other students on this programming assignment.