Executive Summary –Crunch

Chanakya Gaur, *cgaur1*, Sagar Wani, *swani1*, Prashanth Venkateswaran, *pvenkat6*

***Abstract*— Crunch is a wordlist generator which generates all possible permutations and combinations when the user specifies a standard or specific character set. This executive report is based on discovering format string vulnerabilities and exploiting them.**

***Index Terms*—Crunch, SWF, format string, vulnerability, buffer overflow, wordlist generator**

I. INTRODUCTION

Through this report, our aim is to explain the process followed to discover and execute format string vulnerabilities. According to OWASP, a format string exploit occurs when the input sting is evaluated as a command by the application. Using this exploit, the attacker could execute code, cause a segmentation fault or read the stack to make modify the behavior of the application. [1]

The report is divided systematically by first identifying a Linux command line based application and setting up an appropriate machine to inspect the binary. Further, an information leak is attempted. After a successful application leak, an arbitrary read and write is used and using a buffer overflow, an exploit is designed. The aim of this report is to reflect upon the process and result of designing and executing the exploit.

II. EXPLOIT GENERATION PROCESS

This process consists of four components which can help discover and exploit format string vulnerabilities, they are:

A. Preparation
B. Information Leak Exploit
C. Buffer Overflow Exploit
D. Arbitrary Write Exploit

A. Preparation

The first step of preparation was to identify a Linux based command line application to execute a format string exploit on. We chose Crunch, a command line tool whose function is to generate a wordlist for dictionary attacks. After downloading the code of Crunch, we ensured that the code is built in Ubuntu 14.04 and that the source code could be debugged using Eclipse.

Next, we switched off the Address Space Layout Randomization (ASLR). ASLR is a memory-protection process for operating systems which protects against buffer overflows by randomizing the location where system executables are loaded into memory. The command used to switch ASLR off is:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Further, we have gone ahead to clear the executable stack flag of ELF binaries and shared libraries using execstack. Execstack is used to avoid breaking binaries and shared libraries which need executable stack, ELF binaries and shared libraries which can be then marked as requiring executable stack or otherwise. Refer to wiki (http://sva-gitlab.mssi-lab.isi.jhu.edu/swani1/sva-assignment3/wikis) to see how the preparation was completed and updated in Eclipse.

B. Information Leak Exploit

"An info leak is the consequence of exploiting a software vulnerability in order to disclose the layout or content of process/kernel memory.", Fermin J. Serna.

To implement an Information Leak Exploit on Crunch, we first altered the source code. We created a macro which received a char[] input from the user. This char[] buffer, then copied into a string using snprintf. Into this macro, we passed a tainted buffer with content of random strings and %x. Upon execution, the program returned hex values from the stack confirming an information leak. From this leak, we confirmed the possibility of the following attacks: Crashing the program and Format String Exploits.

The major advantage of a successful information leak exploit is being able to view the process memory. We can print something like a series of %08x which will partially dump the stack memory. Depending upon the size of the format string buffer, it is possible to reconstruct most of the stack memory. The information leak exploit basically provides us with control over the stack for further exploitation.

Next, we decided to perform an arbitrary read operation on Crunch to specifically read the string given as input through the program. For this we used the concept used in Information Leak and read the addresses of the string input successfully completing the information leak step of the process. Refer to the wiki (http://sva-gitlab.mssi-lab.isi.jhu.edu/swani1/sva-assignment3/wikis/home) for exploit details.

Sagar Wani, Graduate student, Johns Hopkins University (e-mail:swani1@jhu.edu).
Chanakya Gaur, Graduate student, Johns Hopkins University (e-mail:cgaur1@jhu.edu).
Prashanth Venkateswaran, Graduate student, Johns Hopkins University (e-mail:pvenkat7@jhu.edu).

C. *Buffer Overflow Exploit*

A Buffer Overflow attack is when an attacker, intentionally, puts in more data in the buffer than the buffer can hold which is when the data writes past the buffer. Buffer Overflow exploits can be used for crashing the program or to gain access control by executing arbitrary code. Buffer overflow is a popular vulnerability as these are common against both legacy as well as new software. This is because buffers can work in a variety of ways which makes it vulnerable and sometimes it can be the code to prevent the buffer overflow which can be erroneous. Typically, in a buffer overflow attack, the attacker sends data to a software which it stores in a stack whose size is less than the size of the data leading to overwriting of the stack including the function pointer. The attacker then uses the data to set a return value to the stack so that when the function returns, the control is transferred to a malicious code.

Continuing from the Information Leak Exploit, our next step was to perform a buffer overflow attack on Crunch. To begin, we first wrote a code in which we created out tainted buffer. The tainted buffer contained a series of no operation(NOP) operation codes, 1999 of them, after which it contains a shell code. At the end of the shell code, a return pointer is added which transfers control to one of the NOPs in the buffer.

```
int j=0; int i=0;

strcpy(taintedbuf, "\x90");

for(j=0;j<1999;j++)
{
        strcat(taintedbuf, "\x90");
}

strcat(taintedbuf, shellcode);

taintedbufWritePtr = strlen(taintedbuf);

for(i=0; i<15; i++)
{
intUnion.integer=(retfptraddr);
taintedbuf[taintedbufWritePtr+3]=intUnion.byte[3];
taintedbuf[taintedbufWritePtr+2]=intUnion.byte[2];
taintedbuf[taintedbufWritePtr+1]=intUnion.byte[1];
taintedbuf[taintedbufWritePtr]=intUnion.byte[0];
taintedbufWritePtr+=4;
}
```

Fig 1: Code Snippet - Tainted Buffer

To ensure that our buffer overflow exploit is working appropriately, we followed a few steps. After we created the tainted buffer, we wrote a c program which passes the tainted buffer to the program and overwrites the return pointer with garbage. Then we went further to crash the program with the buffer. Using the tainted buffer from Fig. 1, we then used the return pointer to point to one of the NOPs so that the shell code is executed giving us shell access.

From this exploit, we gained control of the stack and used it to gain shell access. Refer to the wiki (http://sva-gitlab.mssi-lab.isi.jhu.edu/swani1/sva-assignment3/wikis/home) for exploit details.

D. *Arbitrary Write Exploit*

After successfully executing the buffer overflow attack and executing the shell code, we went forward to execute an arbitrary write exploit. An Arbitrary Write Exploit is an extension of a buffer overflow in which instead of redirecting the pointer into the stack it actually points to a specific address from where the shell code starts.

To execute this successfully, we first executed %n%n%n%n. The result from this was E6E6E6E6. We know from the previous execution that the address of the shellcode is BFFFD288. Now that we have both these values, we mathematically calculate how many characters we needed to add in the tainted buffer so that each %n wrote the corresponding address of the shell code. After the calculation, we ran a successful arbitrary write exploit on Crunch. Fig. 2 shows the successful exploit.

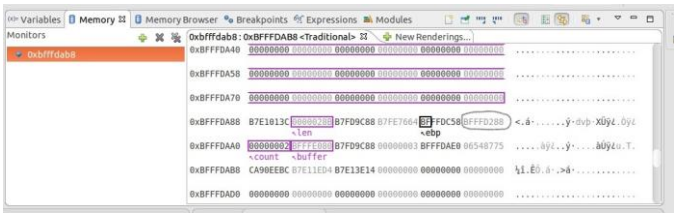

Fig 2: Arbitrary Write - Memory Stack

III. CONCLUSION

Based on this experience, we have learnt a very structured approach on how to exploit software using format string vulnerabilities. This process sheds light on how to work around a software to perform arbitrary read, buffer overflow and arbitrary write exploits using format string vulnerabilities. After completion of this process, we have successfully executed information leak exploit, buffer overflow exploit and arbitrary write exploit on Crunch, a wordlist generator.

IV. APPENDIX

We have setup a GitLab which contains the code as well as documentation for this project. The link for the GitLab is:

http://sva-gitlab.mssi-lab.isi.jhu.edu/swani1/sva-assignment3

V. REFERENCES

1.The Internet, “Format string attack”, OWASP
2. 2001, scut, team teso, “Exploiting Format String Vulnerabilities”