[[1]](#footnote-1)

Executive Summary – Return-to-Libc Exploit - Crunch

Chanakya Gaur, *cgaur1,* Sagar Wani, *swani1*, Prashanth Venkateswaran, pvenkat7

*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 using buffer overflow vulnerability and executing stack smashing and Return-into-Libc exploits.

*Index Terms*—Crunch, SWF, vulnerability, buffer overflow, wordlist generator, arc-injection, return-into-libc, stack

# INTRODUCTION

Through this report, our aim is to explain the process followed to use buffer overflow vulnerability to execute stack smashing and return-into-libc exploits. “A return-to-libc attack is a computer security attack that usually starts with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory.” [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, a character buffer 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.

# EXPLOIT GENERATION PROCESS

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

A. Preparation

C. Stack Smashing using Buffer Overflow

D. Arc-Injection Exploit

## Preparation

 The first step of preparation was to identify a Linux based command line application to execute an arc-injection exploit on. We chose Crunch, a command line tool whose function is to generate a wordlist for dictionary attacks.[2] 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-assignment-4/wikis>) to see how the preparation was completed and updated in Eclipse.[2]

## Stack Smashing using Buffer Overflow

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.[2] 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.[2] 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.

Our first step was to perform a buffer overflow attack on Crunch. To begin, we first wrote a code in which we created our tainted buffer. The tainted buffer contained a series of no operation(NOP) operation codes, 2000 of them, after which it contains a shell code. At the end of the shell code, a series of 16 return pointers are added which transfers control to one of the NOPs in the buffer.

After we created the tainted buffer, we wrote a C program which passes the tainted buffer to the program. Using the tainted buffer from, we then used the return pointer to point to one of the NOPs so that the shell code is executed giving us shell access. Fig 1 shows us a screen shot of the smashed stack using buffer overflow.

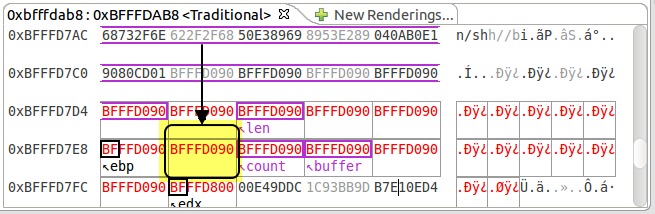


Fig 1: Stack Smashed using Buffer Overflow

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-assignment-4/wikis/home>) for exploit details.

## Arc Injection Exploit

After successfully smashing the stack using buffer overflow, we went forward to execute an arc injection attack. A return-to-libc attack also known as an arc injection attack is an attack that starts with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory. [1]

Sometimes, the stack is not-executable for which we use arc injection attack to mitigate the effect of Data Execution Prevention. Our main goal was to make a libc function call for which we used the system function call and load its arguments from libc which is /bin/sh. We used this in our code from the stdlib.h library to receive the argument: int system (const char \*command). This is passed as the first argument and helps us gain access to the shell. Using gdb we obtained the address of /bin/sh so that it would be our first argument value which is binshaddr. The address we calculated was 0x57f6fcec. We also calculated the value for the second argument “retptraddr” to the value where the system address was stored. This was 0xb7e4d310.

After the return is smashed with the address of the system function, we will be reconstructing a system call stack. Now, immediately following this system address, the compiler expects a value to return to, so we will fill this with garbage "AAAA". After this, we will place the arguement of the system call which will be the address of a string '/bin/sh'. From fig. 2, we can see that the stack has successfully been smashed using an arc injection attack.

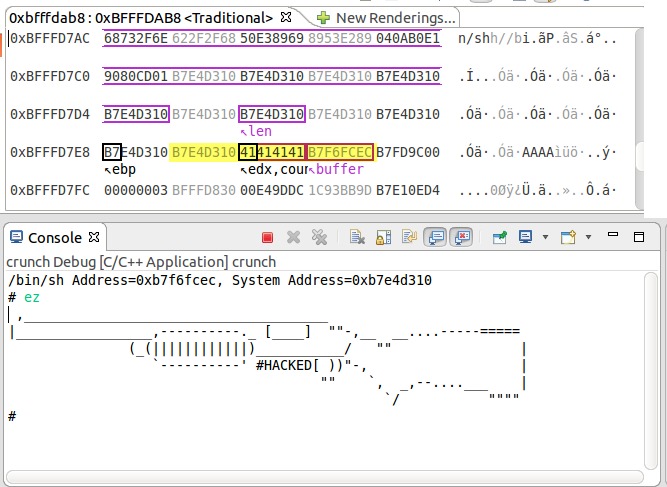
**

Fig 2: Stack smashing using Arc Injection

# Conclusion

## Based on this experience, we have learnt a very structured approach on how to implement an arc-injection exploit uing buffer overflow vulnerabilities. This process sheds light on how to work around a software to perform buffer overflow, stack smashing and arc-injection. After completion of this process, we have successfully buffer overflow, stack smashing and arc injection exploit on Crunch, a wordlist generator.

# 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-assignment-4>

This contains steps to setup the application, execute buffer overflow exploit and arc injection exploit along with the code.

# References

1.The Internet, “Return to libc Attack”, Wikipedia

2. Gaur, C., Venkateswaran, P. and Wani, S. (2017). Executive Summary - Format String Vulnerability.

1. 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). [↑](#footnote-ref-1)