

|  |
| --- |
| **Exploit Development**  **Alexandra Cherry - 1700315**  CMP320: Ethical Hacking 3  BSc Ethical Hacking Year 3  2019/20 |

Abstract

This section should be an **attention grabber**. It should provide a short summary of what your paper is about so provide enough detail to satisfy your client that you met his/her needs and allows the reader to decide if the report is of interest. This is stand alone and should not refer to any other part of the document.

You should include 3 short sections:

• Background to the paper and aim of what you’re trying to achieve.

• What you did (and how).

• What you found and what you conclude from your findings (not too much detail but enough to show that your project is clearly wonderful). Not all projects are a success – and that’s ok too for the purpose of this work, and you can point that out – but preferably with solutions.

[Figure 1‑2.2.1‑a: Stack Buffer Overflow Example (Kiuwan, n.d.) 1](file:///C:\Users\User\Documents\GitHub\Exploit-Development\CMP320_U1-Alexandra_Cherry.docx#_Toc49524267)

[Figure 1‑2.2.1‑b: Buffer Overflow Example (Cloudflare, n.d.) 1](file:///C:\Users\User\Documents\GitHub\Exploit-Development\CMP320_U1-Alexandra_Cherry.docx#_Toc49524268)

[Figure 2.2.1‑a: Initial Crash 3](file:///C:\Users\User\Documents\GitHub\Exploit-Development\CMP320_U1-Alexandra_Cherry.docx#_Toc49524269)

[Figure 2.2.1‑b: Result of `pattern\_create` 4](file:///C:\Users\User\Documents\GitHub\Exploit-Development\CMP320_U1-Alexandra_Cherry.docx#_Toc49524270)

# Introduction

The stack is the section of a computer’s memory temporarily dedicated to a process. They are created when a new function or subroutine is started, it stores both the variables of the parent routine and the subroutine. When a new variable/parameter is declared in a subroutine it is pushed onto the stack. At the exit of the subroutine the stack is cleared by popping the parameters off in a last in first out order.

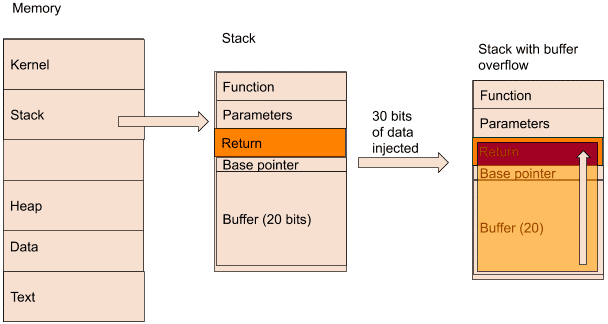
A buffer is a section of memory that is briefly allocated to contain a variable in a function. Buffer overflows are caused when the size of the data written to a fixed length buffer is larger than the size of the buffer, causing the buffer to overflow and write the data in the next buffer (figure 1-a). This could potentially make an access point for a malicious actor or cause the program and/or cause the system to crash.

Figure 1‑2.2.1‑a: Stack Buffer Overflow Example (Kiuwan, n.d.)

Figure 2.2.1‑b: Initial CrashFigure 1‑2.2.1‑c: Stack Buffer Overflow Example (Kiuwan, n.d.)

Figure 1‑2.2.1‑d: Buffer Overflow Example (Cloudflare, n.d.)

Figure 1‑2.2.1‑e: Stack Buffer Overflow Example (Kiuwan, n.d.)Figure 1‑2.2.1‑f: Buffer Overflow Example (Cloudflare, n.d.)

A buffer overflow attack exploits a buffer overflow vulnerability and either writes malicious code onto the stack in order for it to be executed by the program or uses the overwrite of the next buffer to gain access to information or a location. (Veracode, n.d.)

Data execution prevention (DEP) is one method of preventing buffer overflows. It is a feature built into operating systems that works by marking sections of memory non-executable by returning the status code “STATUS\_ACCESS\_VIOLATION” when there is an attempt to execute the memory. This prevents the execution of code stored there creating a barrier between a malicious actor and a successful buffer overflow exploit. (Microsoft, 2018)

There are several methods of bypassing and disabling DEP but this paper is focusing on utilising return-oriented programming chains (ROP chains) to bypass the non-executable area of the stack. ROP chains use pre-existing code in the program to mark the stack as executable. In order to execute custom code on the stack, gadgets (a sequence of instructions ending with a return instruction) are chained together to jump to where the code is stored and execute it – see figure 1-b. (Corlean Team, 2010)

A screenshot of a cell phone

Description automatically generated

Figure 1‑b: Diagram of ROP Chain in Buffer Overflow Attacks (Corlean, 2010)

# Procedure and Results

## Overview of Procedure – ADD REFERENCES

The four stages of the methodology used throughout this investigation were: prove that the vulnerability exists, investigate the vulnerability, perform a proof of concept attack and perform an advanced attack with reverse shell. These stages were repeated with both DEP disabled, and DEP enabled. This exploit was run in a Windows XP S3 virtual machine.

The memory of an application can be viewed using a debugging software, this shows how the underlying processes are affected by inputs. This allows an attacker to create an overflow attack designed for this application.

Cool Player has two inputs – Playlists in the form of .m3u files and skins in the form of .ini files. The focus of this investigation was on the skin files.

## Identifying the Vulnerability – ADD SCREENSHOTS

The first step with assessing a potential vulnerability is to identify that the vulnerability exists. *Cool Player* has two user input fields – playlist files (*.m3u*) and skin files (*.ini* – these files require a specific header). This investigation is focused on exploiting the skin files.

### Skins (.ini)

A screenshot of a computer

Description automatically generatedIdentifying the vulnerability in the skin feature was done by crafting a *Perl* script (see Appendix A for complete *Perl* scripts) to create a skin file that overflowed the buffer, crashing the program and overwriting EIP which was viewed in Immunity Debugger (see figure 2.2.1a).

Figure 2.2.1‑a: Initial Crash

Figure 2.2.1‑b: Result of `pattern\_create`Figure 2.2.1‑c: Initial Crash

A screenshot of a computer

Description automatically generatedAn alphanumeric pattern of 3000 characters was created using the `!mona pattern\_create 3000` command for Immunity Debugger. Running the *Perl* script from the previous step to generate another skin file, this time with the pattern created replacing the string of “A”s. The pattern can be used in conjunction with the `!mona *findmsp*` command for Immunity Debugger to calculate the distance to the EIP and the space available for shellcode (see figures 2.2.1-b through 2.2.1-d). This revealed that the EIP is at an offset of 1056 bytes and that there is 1440 bytes for shellcode.

Figure 2.2.1‑d: Result of `pattern\_create`

Figure 2.2.1‑e: Result of `pattern\_create`

A close up of a keyboard

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure 2.2.1‑d: Result of `findmsp` - shows location of EIP and size available for shellcode

Figure 2.2.1‑c: Result of `pattern\_create` and crash due to the pattern

## DEP Disabled

### Proof of Concept

After, the existence of the vulnerability was verified in section 2.2.1, the distance to EIP was determined to be 1056 bytes and showed that there is 1440 bytes for shellcode. Without this information it is impossible to create a reliable buffer overflow exploit.

In order to gain control of the EIP, the distance to the EIP is filled with characters (in this case 1056 “A”s).

Following the execution of the return in the skin loader, four bytes are popped off the stack; leaving the ESP will point to the start of the shellcode, as it is located right after the bytes that overwrite the EIP in the skin file/exploit.

However, the exact location of the ESP is unknown, so the return address should not be hardcoded into the exploit. To work around this, the EIP is overwritten with a memory address to a `JMP ESP` command that is a fixed address. The `JMP ESP` command tells the assembler to jump to the ESP which is pointing to the shellcode. The address is discovered by running `!mona jmp -r esp` in Immunity Debugger (figure 2.3.1a).

Figure 2.3.1‑a: Results of `jmp -r esp`

Figure 2.3.1‑b: Results of the proof of concept attackFigure 2.3.1‑c: Results of `jmp -r esp`

A picture containing window, computer, sitting, water

Description automatically generated

A screenshot of a computer

Description automatically generatedFinally, shellcode to run “calc.exe” was added to a *Perl* script (containing the header, 1056 “A”s, EIP/JMP ESP location) that was used to exploit the buffer overflow vulnerability and run “calc.exe”.

Figure 2.3.1‑d: Results of the proof of concept attack

### Advanced

The only difference between the *Perl* script used in this advanced exploit and the one used in the basic exploit in section 2.3.1 is the shellcode used. The shellcode used in this exploit was a reverse TCP shell generated using “msfvenom” in Kali Linux. (figure 2.3.2a)

[CREATE SCREENSHOT]

Before running the exploit, a listener was set up on Kali Linux with “msfconsole” (figure 2.3.2-b and 2.3.2-c).

## DEP Enabled

The next stage of this investigation was to exploit the application with DEP enabled (see section 1 for more information on what DEP is and how to bypass it). See figure 2.4a on how DEP was enabled.

Enabling DEP disables the execution of the stack and attempts to execute shellcode from the stack cause access violation errors (figure 2.4b). There are several methods that can be used to work around this.

### Proof of Concept

For this proof of concept attack ROP Chains were used to disable DEP (see section 1 for more information on ROP Chains) and egg-hunting shellcode was used.

ROP Chains are chains of instructions (known as gadgets) that return to the next gadget, chaining the instructions together.

The ROP Chain was generated using `!mona rop -m \*.dll -cpb '\x00\x0a\x0d'` in Immunity Debugger (figure 2.4.1-a). This created ROP Chains in several scripting languages, using memory addresses that do not contain ‘\x00’, ‘\x0a’ or ‘\x0d’, after hunting through all the DLLs. The ROP Chain used can be seen in figure 2.4.1-b.

***ROP CHAIN \* 2***

***Mona***

***Result***

To start a ROP Chain a return pointer is used to jump to the next location provided by the stack frame starting the ROP Chain, if a `JMP ESP` command is used it will cause an access error as it will attempt to execute the shellcode in the stack causing an access violation error. The command `!mona find -type instr -s "retn" -m msvcrt.dll -cp '\x00\x0a' -x x` in Immunity Debugger (figure 2.4.1-c) to generate a list of return instructions in the msvcrt library that were executable and didn’t contain ‘\x00’ or ‘\x0a’ as they would terminate the code execution.

[MONA]

Due to the combined size of the ROP Chain and the shellcode for “calc.exe” being greater than the 1440 bytes between the EIP and a null byte, egg hunting shellcode was used.

Egg hunting shellcode is shellcode that allows for the bypass of the maximum size for shellcode in the stack by placing it in another location in memory. The hunter shellcode searches for and runs the payload shellcode which is marked twice by a tag (also known as an egg). The hunter shellcode was generated using `!mona egg -t w00t` in Immunity Debugger (figure 2.4.1d). This created an egg hunter shellcode using “w00t” as the tag.

[MONA]

### Advanced

The only difference between the *Perl* script used in this advanced exploit and the one used in the basic exploit in section 2.4.1 is the shellcode used. The shellcode used in this exploit was a ***reverse TCP shell*** generated using “msfvenom” in Kali Linux. (figure 2.4.2a)

Before running the exploit, a listener was set up on Kali Linux with “msfconsole” (figure 2.4.2b).

# Discussion

## Buffer Overflow Prevention and Mitigation

There are several ways to prevent buffer overflow attacks.

During and after development of an application, testing to locate and patch overflow vulnerabilities should be performed regularly to prevent successful attacks.

One method to almost completely prevent buffer overflow attacks is to develop in a language, such as PERL, Python, C# or Java, that does not have direct access to memory and/or the languages automatically perform bounds checking. This layer of abstraction prevents an overflow from occurring by not writing the variable directly to the memory and by checking that the variable will fit the buffer without overflowing. (Synopsys, 2017) (Imperva, n.d.)

Other methods require the use of secure handling of buffers. One such method is to use functions, that may not be part of the standard libraries, that perform bounds checking or truncate variables that are too long, preventing overflow of the buffer, instead of using unsecure standard libraries. This prevents the buffer overflowing and overwriting the next buffer.

Another method is to user compiler tools that warn the developer when they are using functions that do not prevent overflow of buffers. (Grover, 2003)

Utilising address space randomization (ASLR), which randomly changes the location in memory of the stack, heap and other program components will not completely prevent buffer overflow attacks but will make it more difficult to carry out a successful attack.

Using canary words – values that are placed on the stack between buffers and return addresses. These values are overwritten when the buffer overflows into them. The values are checked during function return and if the value has been overwritten, the program is terminated, preventing the execution of an overflow attack.

A screenshot of a cell phone

Description automatically generated

Figure 2.4.2‑a: How Terminator Canaries Work (Sidhpurwala, 2018)

There are three types of canary words currently in use. Terminator canaries (shown in figure 2.4.2-a), based on the fact that the majority string operations that end at string terminators, that contain NULL(0x00), CR (0x0d), LF (0x0a), and EOF (0xff) – these characters terminate the majority of string operations, preventing the execution of code. It is possible to bypass this canary using methods that are not stopped at string terminators. A side effect of this type of canary is that the value is known and thus can be overwritten with the correct value.

Random canaries are canaries that are chosen at random when the program is executed – this makes it impossible for an attacker to know the value before running the program. The value is created from hashing the time or taken from /dev/urandom. The value can be discovered if there is an information leak in the application.

The final type of canary in use is random XOR canaries – these are random canaries that are exclusive or scrambled using either all or part of the control data. This means that if the control data is wrong the canary is wrong and will cause the program to terminate. (Sidhpurwala, 2018)

## Evasion of Intrusion Detection Systems

* Signature based
  + Looks for specific patterns
    - Evade by encoding??
* Anomaly based
  + Creates model of baseline activity and compares new behaviour to this

## Future Work

* What would you do if given more time and resources?
  + More advanced shellcode
  + Different methods of bypassing dep

# References

Charles, K., 2018. *Mitigating Buffer Overflow Attacks in Linux/Unix – SecurityOrb.com.* [Online]   
Available at: https://www.securityorb.com/general-security/mitigating-buffer-overflow-attacks-in-linux-unix/  
[Accessed 27 August 2020].

Cloudflare, n.d. *What Is Buffer Overflow? | Cloudflare.* [Online]   
Available at: https://www.cloudflare.com/learning/security/threats/buffer-overflow/  
[Accessed 25 August 2020].

Corlean Team, 2010. *Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube | Corelan Team.* [Online]   
Available at: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/  
[Accessed 23 August 2020].

Corlean, 2010. *Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube | Corelan Team.* [Online]   
Available at: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/  
[Accessed 27 August 2020].

Grover, S., 2003. *Buffer Overflow Attacks and Their Countermeasures | Linux Journal.* [Online]   
Available at: https://www.linuxjournal.com/article/6701  
[Accessed 25 August 2020].

Imperva, n.d. *What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva.* [Online]   
Available at: https://www.imperva.com/learn/application-security/buffer-overflow/  
[Accessed 25 August 2020].

Kiuwan, n.d. *Prevent Buffer Overflow Attacks - Kiuwan.* [Online]   
Available at: https://www.kiuwan.com/prevent-buffer-overflow/  
[Accessed 28 August 2020].

Microsoft, 2018. [Online]   
Available at: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention  
[Accessed 25 August 2020].

Sidhpurwala, H., 2018. *Security Technologies: Stack Smashing Protection (StackGuard) - Red Hat Customer Portal.* [Online]   
Available at: https://access.redhat.com/blogs/766093/posts/3548631  
[Accessed 27 August 2020].

Synopsys, 2017. *How to detect, prevent, and mitigate buffer overflow attacks | Synopsys.* [Online]   
Available at: https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attack­­­s/  
[Accessed 26 August 2020].

Veracode, n.d. *Buffer Overflow Vulnerabilities, Exploits & Attacks | Veracode.* [Online]   
Available at: https://www.veracode.com/security/buffer-overflow  
[Accessed 23 August 2020].

# Appendices

## Appendix A – Perl Code

### DEP Disabled

#### Proof of Concept

#### Advanced

### DEP Enabled

#### Proof of Concept

#### Advanced

## Appendix B – Attaching Skin File to *Cool Player*