

|  |
| --- |
| **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.

Contents

[1 Introduction 1](#_Toc47794087)

[1.1 Application Model 1](#_Toc47794088)

[1.1.1 Process Frame 1](#_Toc47794089)

[1.1.2 The Stack 1](#_Toc47794090)

[1.2 Buffer Overflow Exploits 1](#_Toc47794091)

[1.3 Data Execution Prevention 1](#_Toc47794092)

[2 Procedure and Results 2](#_Toc47794093)

[2.1 Overview of Procedure 2](#_Toc47794094)

[2.2 Identifying the Vulnerability 2](#_Toc47794095)

[2.2.1 Skins (.ini) 2](#_Toc47794096)

[2.3 DEP Disabled 3](#_Toc47794097)

[2.3.1 Proof of Concept 3](#_Toc47794098)

[2.3.2 Advanced 4](#_Toc47794099)

[2.4 DEP Enabled 5](#_Toc47794100)

[2.4.1 Proof of Concept 5](#_Toc47794101)

[2.4.2 Advanced 6](#_Toc47794102)

[3 Discussion 7](#_Toc47794103)

[3.1 Buffer Overflow Prevention/Mitigation 7](#_Toc47794104)

[3.2 Evasion of Intrusion Detection Systems 7](#_Toc47794105)

[3.3 Future Work 7](#_Toc47794106)

[References 8](#_Toc47794107)

[Appendices 9](#_Toc47794108)

[Appendix A – Perl Code 9](#_Toc47794109)

[DEP Disabled 9](#_Toc47794110)

[DEP Enabled 9](#_Toc47794111)

[Appendix B – Attaching Skin File to *Cool Player* 9](#_Toc47794112)

# Introduction

## Application Model

### Process Frame

### The Stack

## Buffer Overflow Exploits

How do they work

## Data Execution Prevention

What is it

Options

How to bypass it

ROP Chains

# Procedure and Results

## Overview of Procedure

***The four stages of the methodology used throughout this investigation are: 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 vm***

* ***The application’s memory can be viewed by attaching it to a debugging software.***
* ***Inputs affect underlying process can craft an overflow attack spec for this app***

***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 the Skins.***

## Identifying the Vulnerability

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)

Identifying 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).

* + 3000 As + crash screenshot

An 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 2.2.1b and 2.2.1c). This revealed that the EIP is at an offset of 1056 bytes and that there is 1440 bytes for shellcode.

* + pattern\_create + script
  + findmsp + file

## DEP Disabled

### Proof of Concept

After verifying the existence of the vulnerability in section 2.2.1, which determined that distance to EIP at 1056 bytes and 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 jm -r esp` in Immunity Debugger (figure 2.3.1a).

* `!mona jm -r esp`

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

* Screenshot of exploit

### 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)

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

## DEP Enabled

The next stage of this investigation was to exploit the application with DEP enabled (see section 1.3 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 ***WHAT SECTION*** 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 2.4.1a). 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 2.4.1b.

***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 2.4.1c) to generate a list of return instructions in the msvcrt library that were executable and didn’t contain ‘\x00’ or ‘\x0a’.

Due to the 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 (egg). The hunter shellcode was generated using `!mona egg -t w00t` in Immunity Debugger (figure 2.4.1d).

### 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

## Evasion of Intrusion Detection Systems

## Future Work

* What would you do if given more time and resources?
* More advanced shellcode

# References

**For URLs, Blogs:**

Bremer, J. 2012. *x86 API Hooking Demystified*. [blog]. 2 July. Available from: [http://jbremer.org/x86http://jbremer.org/x86-api-hooking-demystified/api-hooking-demystified/](http://jbremer.org/x86-api-hooking-demystified/) [Accessed 15 April 2016].

# 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*