

EECS 765 Project 5

Author: Jace Kline 2881618

Introduction

The purpose of this programming assignment is to develop and execute a multi-stage attack against the Internet Explorer 8 Web Browser on Windows 7. This attack is triggered through a buffer overflow in the JNLP plugin for Internet Explorer. The attack involves preparing the heap via the heap spray technique, subsequently pivoting the stack pointer to the heap, and then employing return-oriented programming (ROP) to ultimately jump to our implanted shellcode on the heap.

Setup

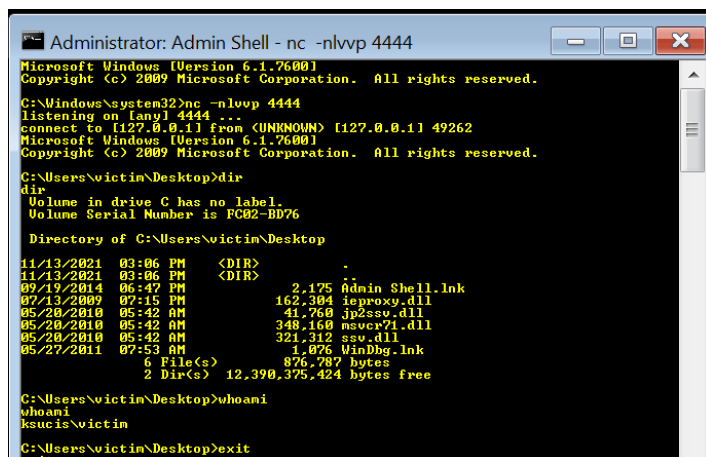
This exploit requires the Windows 7 VM provided on the PA5 instructions page. A webserver is also recommended to host and serve the exploit HTML file, but this file can be opened directly from the victim machine via the Internet Explorer browser. It is assumed that the netcat program is installed on the Windows 7 VM. The first step is to extract the ZIP file holding our exploit contents and then host the `exploit.html` file via a webserver or moving this file to the Windows 7 machine.

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 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 Internet Explorer 8 application on the victim machine. In the browser, navigate to the URL (or local file path) where the `exploit.html` file is located. With this page loaded in the browser, click the "Click Me" button to initiate the attack and spawn the reverse TCP shell. Following this, check the netcat listener terminal to verify that the shell was initiated. We show an example of a successful attack below.



```
Administrator: Admin Shell - nc -nlvvp 4444
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>nc -nlvvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 49262
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\victim\Desktop>dir
dir
Volume in drive C has no label.
Volume Serial Number is FC02-BD76

Directory of C:\Users\victim\Desktop

11/13/2021  03:06 PM    <DIR>          .
11/13/2021  03:06 PM    <DIR>          ..
09/19/2014  06:47 PM                2,175 Admin Shell.lnk
07/13/2009  07:15 PM            162,304 ieproxy.dll
05/20/2010  05:42 AM             41,760 jp2ssv.dll
05/20/2010  05:42 AM            348,160 msocv71.dll
05/20/2010  05:42 AM             321,312 ssv.dll
05/27/2011  07:53 AM              1,076 WinDbg.lnk
               6 File(s)      876,787 bytes
               2 Dir(s)      12,390,375,424 bytes free

C:\Users\victim\Desktop>whoami
whoami
ksucis\victim

C:\Users\victim\Desktop>exit
exit
```

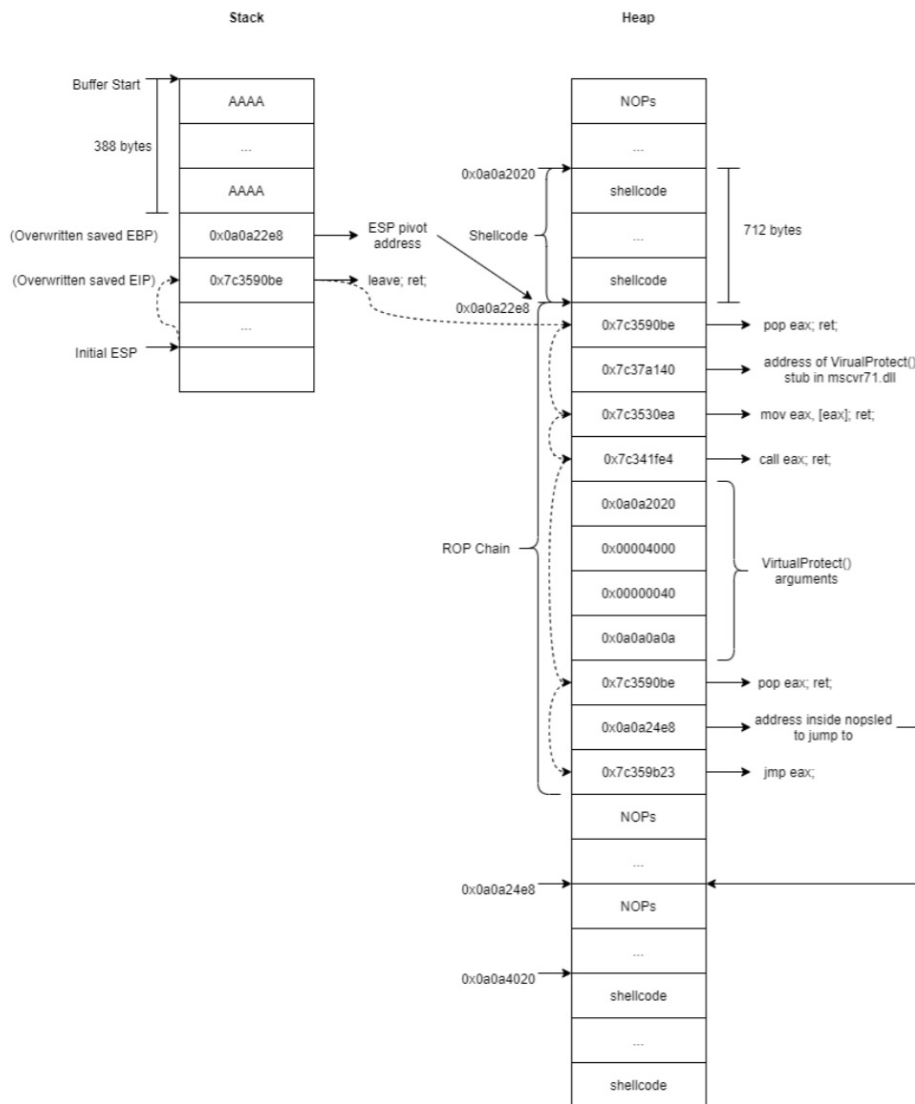
Developing the Exploit

Attack Overview

Our exploit involves utilizing memory on both the stack and the heap. With the help of a JavaScript heap spray library, the heap is where we place our shellcode and ROP chain at deterministic locations. We utilize the stack to overflow a vulnerable buffer and ultimately point the stack pointer to the location of the ROP chain on the heap. After this stack flip, our ROP chain allows us to change the permissions of the memory page on the heap to be RWX (read-write-execute) by calling the `VirtualProtect()` function. Lastly, with an executable heap and the precise knowledge of our shellcode location, we can conclude our ROP chain with a jump to our shellcode.

Malicious Input Structure

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



As we see, this attack involves writing to both the stack and the heap. The job of the stack overwrite is to rewrite the stack pointer (ESP) address such that it points to the top of our ROP chain located on the heap. The goal of the ROP chain is first to change the permission of the heap to include execute permissions and, subsequently, to set up a jump to our shellcode on the newly executable heap. Note that the dashed lines in the diagram correspond to the flow of the stack pointer (ESP) throughout the course of the attack. We discuss the precise attack sequence in the section below.

Attack Sequence

The attack sequence starts with a heap spray and a subsequent buffer overflow (on the stack), triggered by the user clicking the "Click Me" button on our malicious web page. The heap spray, made possible by the Heap Feng Shui library, writes our payload consisting of our shellcode and ROP chain to the heap at deterministic intervals of 0x2000 bytes. The gaps between instances of the payload are filled with NOP instructions. One of the payload start addresses is known to be 0x0a0a2020 and therefore we use this address to orient our attack.

The buffer overflow that takes place on the stack has one purpose: to redirect the stack pointer (ESP) to the top of our ROP chain located on the heap. We know that the assembly instruction `leave` is roughly equivalent to the sequence `mov esp, ebp; pop ebp;`. Notice that this moves the value stored in EBP into ESP. Since we can control the value in the EBP register by overwriting the saved EBP in the vulnerable stack frame, we essentially have the power to utilize this `leave` instruction for our benefit. We also have to consider the fact that we want the ROP chain execution to start immediately after this `leave` call and therefore must find the gadget `leave; ret;`. The address of this gadget, 0x7c3590be, is stored in our overwritten saved EIP. In the overwritten saved EBP, we place the address to the top of the ROP chain, 0x0a0a22e8, as discussed above.

After the stack pointer (ESP) is pivoted to the top of our ROP chain on the heap, the execution of the ROP chain immediately begins due to the `ret` instruction in the `leave; ret;` gadget. The goals of the ROP chain are twofold. First, we want to invoke the `VirtualProtect()` function in order to allow execution on the heap where our shellcode is located. After this is achieved, we want to jump to an address on the newly executable heap that will allow the execution of our shellcode.

The ROP chain starts with the address of the gadget `pop eax; ret;`. The goal of this gadget is to load the address of the `VirtualProtect()` function stub, 0x7c37a140, into the EAX register. Since the EAX register holds the address to the `VirtualProtect()` function stub and not the function pointer itself, we dereference the value of the function stub and move this into the EAX register via the gadget `mov eax, [eax]; ret;`. This gadget is located at 0x7c3530ea. Following this, it is time to call the `VirtualProtect()` function since the pointer to this function is now loaded into EAX. We use the gadget `call eax; ret;`, located at 0x7c341fe4, to invoke the `VirtualProtect()` function. The four arguments to this function call are placed in proper order directly beneath this gadget's address in the ROP chain. The arguments specify the start address of our memory page, the size of the region, the new protection option, and the address of the variable that receives the previous access protection value. The values of these arguments are 0x0a0a2020 (start of our memory page), 0x00004000 (size of region to update), 0x00000040 (RWX protections), and 0x0a0a0a0a (any writeable), respectively. Following this call to `VirtualProtect()`, our region of the heap is now executable.

With an executable region of the heap, the rest of the ROP chain aims to direct execution to our shellcode that is also located on this region of the heap. We utilize the `pop eax; ret;` gadget again to load our desired jump address into EAX. The address that we load into EAX is 0x0a0a24e8 which is in the NOP sled safely between our ROP chain and the next instance of our shellcode at 0x0a0a4020. The last step is to simply perform a jump into the NOP sled by invoking the gadget `jmp eax;`.

Execution of this gadget results in our NOP sled being jumped to and eventually the execution of the next occurrence of our shellcode.

Malicious Input Parameters

For this exploit, we require the following parameters:

- System architecture and endianness
- Offset from the buffer start to the saved EBP and saved EIP locations
- Shellcode parameters: listener host, listener port, and encoding
- Address of VirtualProtect() function stub
- Address of gadgets in shared libraries (.dll files)
- Shellcode address on the heap
- ROP chain address on the heap

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

To find the offsets to the saved EBP and saved EIP addresses, we utilized the `pattern_create` and `pattern_offset` tools in Kali with a pattern length of 800 obtained from trial and error. Within the Windows Debugger (WinDbg) on the Windows 7 machine, we analyzed the registers after we triggered an exception via buffer overflow (with the input pattern). We copied the values of EBP and EIP and found the offsets with the `pattern_offset` tool to be 388 and 392, respectively. These offsets are important to our exploit because we can use a `leave; ret; gadget` to move the value of the overwritten EBP register into EIP, and therefore redirect the stack pointer to the top of our ROP chain located on the heap.

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.

Our attack hinges on the idea of utilizing the VirtualProtect() function in kernel32.dll to change the protection of the heap to be executable. Since kernel32.dll utilizes ASLR, we must find a stub in another shared library that points to the VirtualProtect() function at runtime. To find a stub to this function, we first find a candidate library via WinDbg that does not have ASLR enabled. We find four potential DLLs: ieproxy.dll, jp2ssv.dll, ssv.dll, and msvcr71.dll. Since msvcr71.dll is also a shared library developed by Microsoft, we tried to find the function stub in this file first. We used the `!dh msvcr71` command in WinDbg to find the offset of the import address table in the msvcr71.dll library. This offset was found to be 3a000. From this, we queried the import address table by running `dps msvcr71+3a000`. We found the 'kernel32!VirtualProtect' stub at address 0x7c37a140.

Gadgets are small sequences of instructions, often followed by a `ret` instruction, that allow us to control the execution flow of a hijacked program by controlling the stack pointer (ESP). Referring to the diagram of our input structure above, we want to find the following gadgets: `leave; ret; , pop eax; ret; , mov eax, [eax]; ret; , call eax; ret; , and jmp eax; .` To ensure consistent addresses across runs, we must find our gadgets in the non-ASLR shared libraries listed in the last paragraph. To find these gadgets, we copy these DLL files to our Kali VM and utilize the tools `sky_search_raw` and `msfpescan`. We used the former to find addresses of a desired instruction and we used the latter to filter the addresses that were followed by a `ret` instruction. In other words, we run `sky_search_raw` to generate potential gadget addresses containing our target instruction, and then we use `msfpescan` to filter which addresses are followed by a `ret` instruction. Since this is tedious to achieve manually, we wrote a helper python script, `findgadget.py`, that combines the outputs from `sky_search_raw` and `msfpescan` to only show addresses that have the desired gadgets ending in a `ret` instruction. The values for each of the listed gadgets can be seen in the diagram. Interestingly, we were able to find all the gadgets in the msvcr71.dll library.

The last parameters relevant to our attack are the locations of the shellcode and ROP chain on the heap. The heap spray library we utilize allows us to deterministically predict the layout of the heap spray contents. Specifically, we know that our payload (shellcode and ROP chain) will start at the address 0x0a0a2020 and reappear every 0x2000 bytes. We also know that our shellcode has length 712 (or 0x2c8) bytes. Since we choose to place our ROP chain after our shellcode in the payload, we can deduce that our shellcode starts at address 0x0a0a2020 and our ROP chain starts at address 0x0a0a2020 + 0x2c8 = 0x0a22e8. Our ROP chain utilizes a total of 11 words in memory which equates to 44 bytes. Therefore, when we perform the final `jmp eax` instruction in the ROP chain, we choose the value of the EAX register to be the address of 0x0a0a24e8 which is safely in the NOP sled leading to the next shellcode instance.

Generating the Malicious Input

The malicious input is dynamically generated and delivered via the `exploit.html` file, which includes the exploitative JavaScript code. The JavaScript code performs two key functions. The first is to enact the heap spray which sprays the heap of the Internet Explorer application with our shellcode and ROP chain. Next, the trigger function initiates the buffer overflow attack responsible for causing a stack pivot (of the ESP) to the top of our implanted ROP chain on the heap. The attack is carried out when the victim user clicks on the "Click Me" button. Refer to the `exploit.html` file for implementation specifics.

References and Collaborations

References

- [EECS765 PA5 Description](#)
- [EECS765 ROP and Heap Spray Slides](#)
- [EECS765 Lecture Video - 11/11/2021](#)
- [Microsoft VirtualProtect\(\) Documentation](#)
- [Heap Spray Library Paper \(Heap Feng Shui\)](#)

Collaborations

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