Jigyas Sharma

Dr. Alexandru Bardas

EECS 765

21 November 2023

Programming Assignment 5: Report

Introduction

In this Programming Assignment, we develop an exploit which contains a buffer overflow vulnerability in the “jnlp plugin” which was active in the Internet Explorer. The victim machine for the attack was “Windows 7”. The attack used the theory of Return Oriented Programming as the victim machine and application employed all the modern protection mechanisms such as ASLR, DEP, GS and SEH. The general overview of the attack is to make a stack pivot on to the heap, so we can create ROP frames to by pass the major protection which DEP. We also use the mechanism of heap spraying to make sure that we can correctly predict the address of our ROP frames and shell code.

Running the Exploit

Exploit Generation File: Exploit.html

LHOST: 127.0.0.1

LPORT: 8998

Step 1: Place the “Exploit.html” file in your host folder.

Step 2: On the victim computer, access the webpage where the exploit.html is located. Eg: 192.168.32.10/var/www/jnlp/Exploit.html

Step 3: On the webpage, click on the distinct “Click Me” button.

Step 4: The “Click Me” button returns a shell to 127.0.0.1 at port 8998.

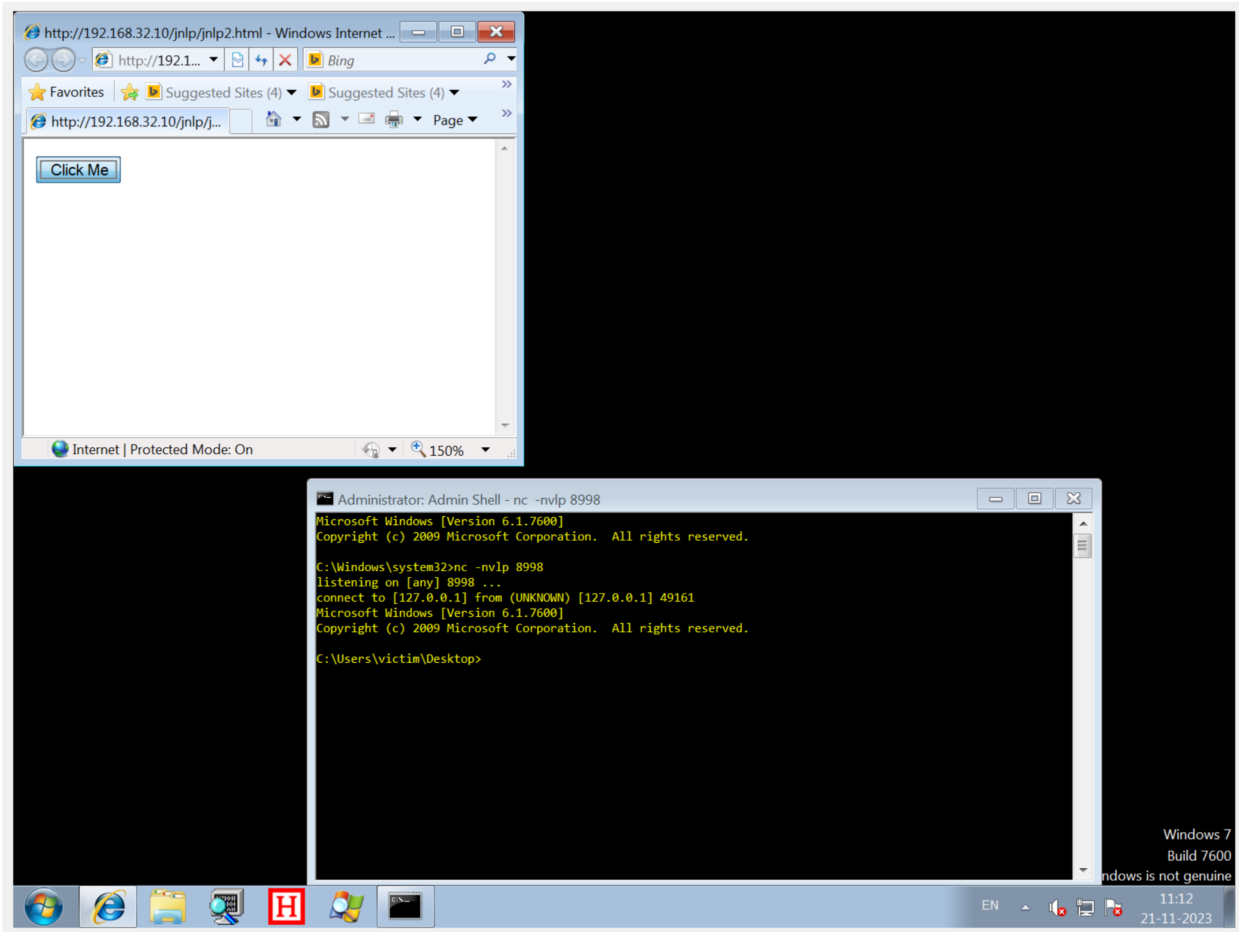

Figure 1: Click Me returns a shell on Port 8998

Developing the Exploit

The exploit for this attack utilized two spaces in memory. Firstly, the stack is used to reach the overflow vulnerability. When the vulnerability is reached, the stack is flipped to the heap. This is done as the memory areas are marked as DEP(Data Execution Prevention) which prevents the memory from executing addresses or code in the memory. The point of flipping the heap is to make sure that we can create ROP(Return Oriented Programming) frames. The ROP chain on the heap then uses a back door to "Virtual Protect" which is a kernel32 function that can change the permission of the memory from read and write only to executable. Lastly, after the heap permissions are changed to executable, another ROP gadget is used to transfer the execution to our shell code. Firstly, to develop the exploit we need to see if and where the stack overflow occurs. The stack overflow is triggered by using excessive "A"s till we can see them in the EIP which would suggest that the stack can be overflowed and we can control the EIP. We use "./pattern_create" and "./pattern_offset" to determine the exact parameters for EIP and EBP as we need to use both of these registers as our first ROP gadget to do a stack pivot. The first ROP gadget that we use is called "leave; ret". The "leave; ret" gadget helps us do a stack pivot on to the heap. The address of the "leave; ret" is loaded on to the EIP and the address of where to send the execution is loaded on to the EBP. Now, before the execution is flipped on to the heap, actually, when the page is loaded up, we execute a heap spray function. The heap spray loads up copies of the same data in a large area of the heap, so it is easier to send our execution to an area with the exploit code. Now, after the execution has been flipped on to the heap, we land our execution towards a chain of ROP frames. These ROP chains, essentially, use a backdoor to a "Virtual Protect" which is a part of kernel32. The ROP chain is constructed using multiple gadgets, the first gadget loads up the "Virtual Protect" and then other gadgets are then used as helpers to load up the arguments to the "Virtual Protect" function which changes the memory to be executable. The gadgets used for the ROP chain start with a "pop eax, ret" which is used to call "Virtual Protect". The next gadgets used are "move eax, [eax]; ret" and "call eax; ret" which are used to load up the arguments in to the function. The last part of the ROP chain is a hard coded address to where our shell code lies in the heap.

Structure of the Input

payload = AAAA + ROP_Chain + NOPS + SHELLCODE

ROP_Chain = pop_eax_ret + Virtual_Protect + move_eax_eax_ret + call_eax_ret + address

Above is the general structure of the input which I will explain below.

AAAA*:* The four A characters are added as the first gadget("leave; ret") that we use pops whatever is loaded in the EBP. We use "A" and not NOPs as NOPs may be flagged as executable and the operating system might take over.

ROP_Chain: The next is the ROP chain which contains multiple ROP gadgets.

pop_eax_ret: This gadget pops what is in the eax and returns to the address next to it, which in our case is "Virtual_Protect".

Virtual_Protect: This is an address to a pointer that can access the Virtual Protect function.

mov_eax_eax_ret: This gadget is used to load up arguments in to the eax register that will be used by the Virtual Protect function.

call_eax_ret: This gadget then calls the arguments for the Virtual_Protect function to finally change the memory to be executable.

address: This is the address to a place where the shell code starts however, we point it to where we have some NOPS just in case.

NOP: We then add some NOPs as now the data should be executable to make sure that we have some space between our shell code and the frames of ROP gadgets in case memory were to shift during execution.

SHELLCODE: This is the shell code that is generated using "Metasploit Framework".

The payload is loaded into the spray_heap function so that this payload is replicated throughout the heap and we can land on our payload at multiple different addresses and it is easier to locate.

Determining the Parameters used in the Malicious Input

DLL: “msvcr71.dll”

Leave_eax_ret: 0x7c372f33

Pop_eax_ret: 0x7c344cc1

Virtual_Protect: 0x7c37a140

Mov_eax_eax_ret: 0x7c3530ea

Call_eax_ret: 0x7c341fe4

EIP_Offset: 392

EBP_Offset: 388

Address of the shellcode: 0x0a0a2050

Architecture: Little Endian

Shell Code: “shellcode_noencoder.txt”

The first parameters that we need to figure out are EIP and EBP offset. We use a pattern of size 800 using “./pattern_create” from Metasploit Frameworks’s exploit tools and push them on the stack. We then use the “./pattern_offset” to check what the offset at which EIP and EBP are overwritten. Next, we need to find the all the gadgets that we need. We chose the dll to look for the gadgets to be “msvcr71.dll” by trial and error. The criteria for finding the DLL is to make sure that it contains a pointer to “Virtual_Protect”, furthermore, has the least amount of protection mechanisms in place to make developing the exploit easier. Thirdly, we need to find our gadgets. I used “skyrack” to find the gadgets, however, skyrack sometimes returns single gadgets and it became tedious to check every gadget containing a return manually, therefore, I moved to using a tool called Ropper that returns gadgets with return already after the gadget(Refer figure 2 and 3). Lastly, we figure out the address to where the shellcode will be in the heap using “WinDBG” and the command “dd 0x0a0a2020” which displays the heap memory contents at which point we match the start of the shell code to the memory that it begins at in the heap(Refer figure 5). Note: We point the shellcode to NOPs as at that point, the memory should be executable.

```
jigya@Darksst:~/Desktop/Github_Projects/Cryptography/Return_Oriented_Programming$ ropper --file msvcr71.dll --search "le
ave; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: leave; ret

[INFO] File: msvcr71.dll
0x7c351d30: leave; ret 0x10;
0x7c34349f: leave; ret 0xc;
0x7c34ea34: leave; ret 4;
0x7c359aef: leave; ret 8;
0x7c377432: leave; ret;
```

Figure 2: "leave; ret" using Ropper

```
0x7c344ad1: pop eax; ret 4;
0x7c344b1d: pop eax; ret 8;
0x7c344cc1: pop eax; ret;

jigya@Darksst:~/Desktop/Github_Projects/Cryptography/Return_Oriented_Programming$ ropper --file msvcr71.dll --search "ca
ll [eax]; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: call [eax]; ret

jigya@Darksst:~/Desktop/Github_Projects/Cryptography/Return_Oriented_Programming$ ropper --file msvcr71.dll --search "mo
v eax, [eax]; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: mov eax, [eax]; ret

[INFO] File: msvcr71.dll
0x7c3530ea: mov eax, dword ptr [eax]; ret;

jigya@Darksst:~/Desktop/Github_Projects/Cryptography/Return_Oriented_Programming$ ropper --file msvcr71.dll --search "ca
ll eax; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: call eax; ret

[INFO] File: msvcr71.dll
0x7c341fe4: call eax; ret;
```

Figure 3: Other gadgets using Ropper

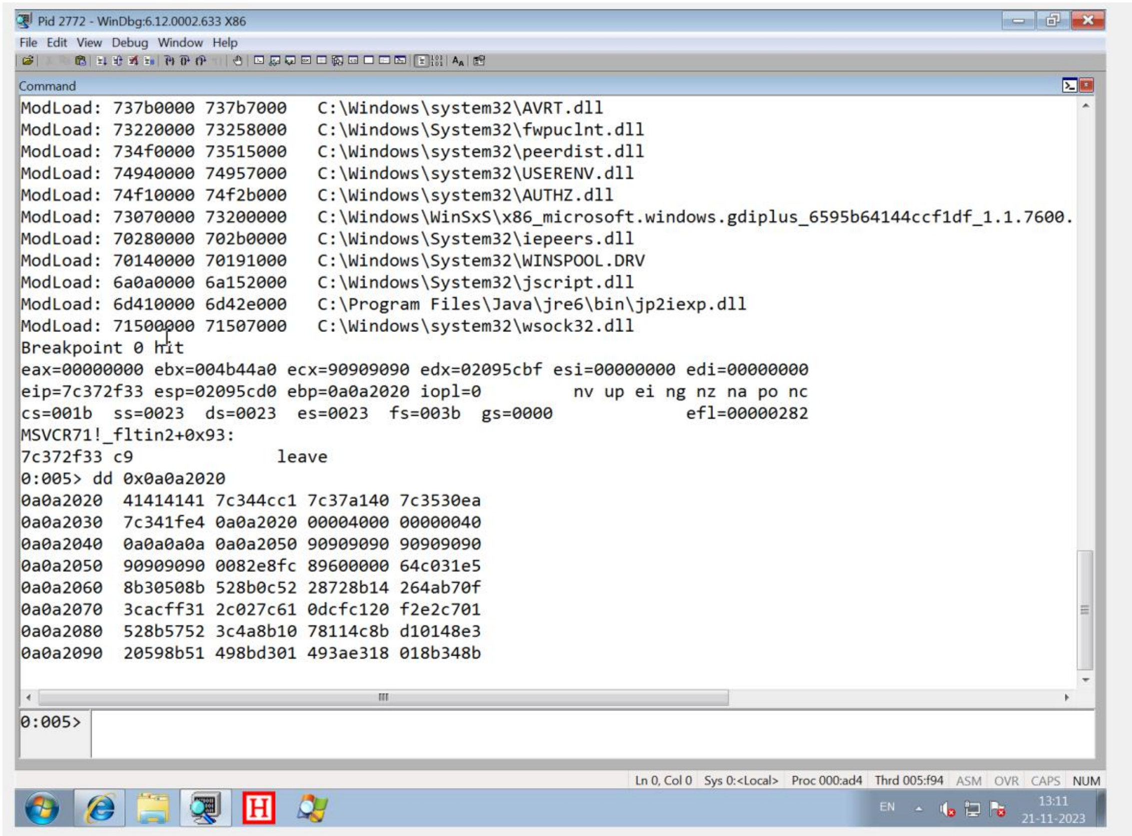

Figure 4: Heap analysis to find shellcode address

Generating Malicious Input

The malicious is input is generated at two points during this exploit. Firstly, when the page is loaded, the heap gets sprayed with our malicious input payload. The heap is constructed with multiple of the same input with large NOP sleds in the middle. The second malicious input is triggered when the “Click Me” button is pressed. This malicious input triggers the buffer overflow vulnerability which leads to the dominos falling one after the other. The other malicious input, shellcode, was generated using Metasploit Frameworks “msfconsole”. The shellcode was generated using no encoders and the language used for the shell code was js_be(JavaScript Bigendian).

References

None Applicable

Collaborations

Ina Fendel : Heap Spray Functionality, Non-Working Gadgets

Yuying Li: Ropper Tool