Arthur Liou
CS373

Week 4 Writeup

Prompt: Submitting a write-up of your thoughts, impressions, and any conclusions based on the material from the week. Each week will have its own assignment in the grades page.

For this week's writeup, I'm reflecting on the topic – Vulnerabilities and Exploits. While informative, I thought it was a lot of fluff and the material was minimally educational. Looking back, it looks like the material could be condensed into maybe 15-20 slides and 20 minutes of lecture material. BUT this week did allow us to work through the labs and see some of these in real time. With this in mind, I'm wondering if I'm missing the bigger picture that ties this together. When going through this week's material, we went over just the few flaw classes and vulnerabilities in very granular detail. I couldn't shake the feeling that we were just getting 1% visibility of how deep the entire topic could go, and that that 1% was just a small insignificant part of the topic(s) (let's configuration vulnerability as an example) that we could have dived in deeper. What's I'm trying to get here is that we're just getting baby steps in to the topic while what I want to actually learn here is how to run.

Concluding, while this week was a bit of a letdown, I'm hoping that the next few following weeks' material and topics that are being taught is much more engaging and interesting.

Lecture Notes

Lecture 1 – Vulnerabilities and Exploits

- WinDBG, Stack/Stack-based Vulnerability
- Program Control
- Manipulating Software – finding "bugs" which alter the program behavior
- Taking advantage of a misconfiguration or poor programming practice
- Bug Bounty Programs
- Lab 1: Hello Mr. WinDBG, Pouring through the registry, program memory,
- Viewng Memory: dd, da, du; Breakpoints: bp <addr>, Clear all: bc *, Stepping: t, p; Disassembly View -> Disass.; Conversion: .formats; Math: ?1+1; Modules,
- Some Flaw Classes & Vulnerabilities:
    - Configuration: weak password,
    - Logic – authorization issues,
    - Storage – Inadequate encryption
    - Input validation – memory corruption, injection
- Memory Corruption – accessing memory in an invalid way which results in an undefined behavior. Reading/writing, usually stack or heap, originally unintended, what we're looking to control
- Common Categories – lifetime control (exploit tomorrow), uninitialized memory, array index calculations, buffer length calculations (exploit today)

- Exploitation – taking advantage of a vulnerability – control the "undefined" behavior
- vulnerability trigger – invokes the software bug to obtain control of the program
- payload – action to be performed when control is obtained. Ex. "Shell" code" – usually assembly code to execute a shell
- Stack Recap – Main and step functions to recap
- Exploit Round 1 – Stack overflow
- Code Execution
- 1) Crash Triage
- 2) Determine the return address
- 3) position our shell code
- 4) Find the address of our shellcode
- Crash Triage – what do we control? What registers contain or point to attacker-controlled data? Is that data on the stack or heap? Do we control critical data such as stack frames?
- Where are we in the execution of the program? Where's the vulnerability
- 2) Determine the return address offset
    - Figure out the offset to EIP, don't fear JavaScript. Lab helpers – "msfPatternString" variable. From WinDBG: !load byakugan !pattern offset 2000
- 3) Position our shellcode – linear stack overflow
- Then #4
- Trampoline – jump to a location which jumps to another (eg. Addr of a jmp esp instruction)
    - 1) Find a module loaded at a static address
    - 2) Find "jump esp" (or similar instruction within that memory space
- Lab 2: Terx – Smashing the stack
- 1) Triage: k for call stack and disassembly view
- 2) Trigger (build the s variable in the JS), MakeString(Amount) // 1 = 2 byte, remember order
- 3) Find address to jmp esp in windbg, add it to 's' -> s[start][end] ff e4
- 4) Add in "shell code" variable to 's'

Lecture 2 – Vulnerabilities and Exploits

- Use-After-Free
- 1) Free the Object
- 2) Replace the object with our – figure out the size, make allocations of the same size
- 3) Reposition our shellcode
- 4) Use the object again
- Heap! heapAlloc and default process heap
- Low Fragmentation Heap – creates buckets for a specific size after the nth allocation of that size
- Allocating heap data with JavaScript, enable LFH
- Freein' Then Usin'

- Browsers – interprets languages to render pages, allows u sto allocate/de-allocate on demand
- Java Classes, Heap
- Tools: Page Heap – Special “Debugging” heap, Enabled via gflags (elevated cmd prompt), !heap WinDbg extension
- Use-After-Free from 2a above – figure out the size (break on heapFree())
- Freedom
- Browser use (for #3), Heapsray – high memory address
- Lesson 3 – Exploit!