Nicolas Barraclough
10/24/2020
CS373 - Defense Against the Dark Arts

Week 4 Write-Up

This week, we wore our red hats and got to learn about exploiting known vulnerabilities. We used WinDBG instead of GDB as our program debugging tool. Aside from having commands and command outputs separated from each other, WinDBG can help identify when/where/and why a program crashes. After being introduced to the basic WinDBG commands and how to sort of dissect a program, we learned about exploiting. The exploits we will be looking at are designed to access memory in an invalid way to result in an undefined behavior that we will orchestrate.

The lecturer described the general idea of the exploit we'd be looking at by reviewing the stack and what sort of things a program will put onto the stack. One thing that the stack needs to keep track of program execution is the point in memory that a function returns to after its completion. If this part of the stack is overwritten, it can cause the program to crash. However, by overwriting the part of the stack with this saved, intended point in memory to a different point in memory, we can disrupt the intended path of the program. This can lead to taking control of everything the program has access to because the system will "think" that the program is intending to perform these actions.

This specific exploit has 4 parts: Crash Triage, Determining the the return address offset, Positioning the shellcode, and Finding the address of the shellcode. In the crash triage, we attempt to understand what is happening when the crash takes place. Where the crash happens, the state of the program upon crashing, and other investigatory information will tell us what we could have control over. Once we find where the program crashes, we can take a look at the return address offset for that particular function return. Next, we have to position our filler code and shellcode at (or near) the return address offset so that we can give the program a valid path to follow which leads to our injected shellcode. I'm not sure why the next step is to find the address of the shellcode because the more reliable method is to trick the execution into performing a jump to the esp (top of the stack). Coined "trampolining," The attacker would find a particular byte within the program's dll, *ffe4*, that can act as the desired executable instruction (jmp esp). This instruction will get the eip to point directly to our shellcode which now exists at the top of the stack. And voilà! We successfully tricked the program into running our shellcode and gained control over the program and possibly the system.

Use-after-free vulnerability exploit. The memory we are accessing is on the heap. This vulnerability takes advantage of the way that the system allocates memory on a process heap. The lecturer gave a pretty complex description of how exactly this

vulnerability presents itself, so I will likely have to research more on this through the next week. In short, the use-after-free vulnerability allows an exploit to the freed memory. An attacker can put shellcode at that freed space in memory and the program will then have access to run that shellcode. I believe that is the jist of what is going on in this exploit.

Post-Lab Notes:

I really enjoyed writing my first exploit! The lecturer mentioned that this type of exploit would not work on modern operating systems because this was such a common form of attack that new OS's automatically detect when the stack frame has been altered. I learned about canary values in Intro to Security. Through this lab, I learned a lot more about the stack, a program's flow, the EIP, and the stack frame. It was a bit challenging using an old OS, but once I got the hang of it, it wasn't too bad. I appreciated having a pre-configured VM. The only thing that I am still a bit confused about is how to turn "launch calc.exe" into the shellcode that is executed. I will need to do more research. Aside from all that, I now fully understand the excitement that the lecturer, Brad, was talking about once the calculator pops up!