Week 4 Write Up

In Week 4, the lectures centered around the topic of vulnerabilities and exploitation. In specific we studied the more common methods in which attackers use to gain access inside a network defensive perimeter, and that’s the internet browser.

Nowadays, most business have strong DMZ and firewalls to protect the business from attacks outside the perimeter, but as we’ve learned in this course, a business is only as strong as its weakest link. In modern times, attackers employ social engineering tactics to gain access inside a network. One strategy that employs social engineering along with sophisticated attacks is having the user visit a page, have the page load JavaScript code that will take advantage of a browser function or vulnerability.

This week we discussed two ways we can manipulate memory via the stack through the browser.

* Stack Smashing / Memory Corruption
* Use After Free Vulnerabilities

Stack Smashing

In order to corrupt the stack, we need to a few things as per the lectures:

1. Perform a crash triage
2. Determine the return address offset
3. Position our shellcode
4. Find the address of our shell code so it can “live” on the stack

A stack smash occurs when the stack buffer is filled with data supplied from an untrusted user and the data is corrupts the stack and malicious executable code is injected into the stack. The code is executed when the stack attempts to perform a return to a function via the return address on EIP, but the attacker uses a NOP slide to position EIP to their malicious code on the stack.

For reference, as mentioned in the lectures, a NOP slide is a sequence of no operations meant to “slide” the CPU’s instruction pointer to a desired destination whenever the program branches to a memory address anywhere on the slide.

[INSERT PICTURE HERE]

Operating systems have now implemented various protection schemes to protect against a stack smash. In my research I found these protections can be classified into three categories:

* Detect the stack buffer overflow has occurred and prevent redirection of EIP to the malicious code
* Prevent the execution of malicious code from the stack without directly detecting the stack buffer overflow
* Randomize the memory space such that finding executable code becomes unreliable (i.e. Address Space Layout Randomization / ASLR)

Use-After-Free

Use after free is another method used to exploit memory via the browser. This vulnerability is triggered when an object in memory is freed and then reference later by the application, but the attacker has control of the freed object/memory. In order for this the following conditions must be present for the attacker to exploit:

1. Load shellcode in memory at a predictable address
2. Force an object to be freed and then overwrite it with one that includes a virtual table pointer (VPTR) linked to a fake vtable (virtual table) pointing to the shellcode loaded in step 1
3. Trigger a vulnerability in the browser to reuse the freed object (which now has malicious pointers inserted by the attacker) and redirect execution flow to the shellcode loaded in memory

Below is a diagram showing the flow of a use-after-free exploit:

[INSERT PICTURE HERE]

Modern browsers, like Microsoft’s Edge in Windows 10, have functions in place to mitigate use after free exploits. Recently, Microsoft introduced MemGC which prevents the freeing of memory chunks if references to them are found. MemGC checks the registers and the stack for chunk references, and in addition, scans the contents of MemGC managed chunks for references.

Below is flowchart of how MemGC works:

[INSERT MEMGC PIC HERE]

For more information you can click on the link below. It’s an article from LinkedIn on a BlackHat session in which the presenter discusses MemGC.

https://www.linkedin.com/pulse/memgc-use-after-free-exploit-mitigation-edge-ie-10-mayur-agnihotri

Tools

Also, in the lecture we discussed how WinDbg is a great tool for establishing breakpoints in applications and examining the applications assembly code. WinDbg is also extremely useful when attempting to review the stack and heap starting points, strings in the applications, and register contents, including the instruction pointer.

WinDbg can also be used to install breakpoints in running process/applications. Allowing a user to observe behavior incrementally in real time. Below is an article discussing debugging malware with WinDbg

https://www.ixiacom.com/company/blog/debugging-malware-windbg