Exploitations in Linux

Micheal Wetherbee (z93r546), Michael Utt (p41r185), Cory Lagor (v89h865), and Emilia Bourgeois (n37n792)

Introduction

Our group decided to study and implement vulnerabilities that have existed in past versions of the Linux operating system. The version we specifically chose was Ubuntu 12.04. Our goals were to each choose an exploit that we found interesting and create a version of the attack ourselves. Micheal Wetherbee decided to focus on the Dirty Cow exploit, Michael Utt did an implementation of a buffer overflow vulnerability, and Cory Lagor and Emilia Bourgeois worked together on the Spectre attack. We worked from virtual machines in order to prevent ourselves from destroying our own systems while attempting to implement these attacks. Specifically we used SeedUbuntu12.04, which is commonly used to teach students about system vulnerabilities. Due to the complexity of these attacks we were unable to create patches of our own, so we instead researched how other, more qualified, people have created patches for each of our attacks. Throughout this technical report we go into more detail about how each of us performed our tasks and what our findings were.

Background and Related Work

Dirty Cow

Dirty COW (Copy on Write) is a Linux based exploit that has been formally patched. This exploit was usable on Linux based systems for 9 years spanning from 2007 to 2016. There has also been documentation on this exploit being used on android devices. Linus Torvald patched the exploit for the kernel back on October 18th of 2016. The exploit uses a race condition that can be brought around by a flaw in the memory system of the operating system.

Spectre

For the Spectre Attack, a good place to start is through the lessons taught by Wenliang Du, called Seed Labs. They are an easily to follow and informative set of explanations followed by hands-on activities. To put the concepts used into a short and concise way, Spectre is not an attack on the OS itself, moreover it exploits a problem within the hardware of most modern CPUs. As a means to increase speed and productivity, CPUs are able to run multiple things in parallel within a thread. What this means is that a CPU can try to “predict” the outcome of a given command by running it’s instructions before checking the logic behind them (this is called Branch Prediction & Out-Of-Order Execution). These results are placed on the CPU’s cache until it can get a chance to check the logic. If everything is good, logic-wise, then the CPU will output the result. If the logic does not work out, surprisingly, they just get left on the cache.

Buffer Overflow

This attack required the least amount of external research because it relates to what we have learned in operating systems the most. This attack happens entirely in the stack layer of our program, which is visually shown in the slides attached to this document. The area that took the most time to research was on assembly code instructions and how to read and write them. We touched upon how c programs call assembly instructions, but this attack required more in depth knowledge. The sources used were the Seed lab for buffer overflow vulnerabilities and videos that came from the YouTube channel Computerphile. Both of these are cited in the reference page at the bottom of this document.

Approach and Methods

Dirty Cow

The Dirty COW attack is rather simple in nature. To get into a read only file such as root access this exploit attacks the memory of the system to cause a race condition that gives an opening for the copy on write (COW) to happen. The race condition is caused by creating a thread pulling at the read only file to get a copy of it mapped into memory. This string is called using Madv_dontneed which frees up the resources for the process but leaves it alone beyond that for all functions after it to be done. While this is going on the user creates another string targeting /proc/self/mem to access the file we mapped to memory and it will presume to change its text. If we do these processes over and over we can confuse the machine into flip flopping these processes so that the string will write to the original read only file instead of the duplicate. For this exploit I followed the lessons taught by Wenliang until I felt confident to look at other resources. I then found a youtube video online that broke down and explained to me the exploit in great detail.(LiveOverFlow) To patch this exploit Linus changed how madv_dontneed works. It now dropes the mapped file from memory if it's dirty which can be found by checking a bit to see if the file has been modified or not. With the modified file being dropped we will never get the race conditioning by flip flopping the processes if one gets dropped every time it's stalled.

Spectre

For our attack to work, we need a way to take advantage of the CPU's temporary cache, this can be done by using something called a "Side Channel Attack". But, before we delve into this technique, it first needs to be understood what we hope to gain by performing this attack. While many exploits into an OS are hoping to gain root access, in this circumstance, we are trying to gain useful information on the system that the OS does not want us to find. Because of this, for our example, we create a string, called Secret, and then use our attack to find it within the CPU cache. The Side Channel Attack will attempt to grab our Secret from the CPU cache instead of from main memory within the OS, hence the title, "Side Channel" because we are using a means within the hardware instead of software to attack. A necessary technique for us to use is called Flush+Reload. What this means is we will flush the contents of the cache's memory array to ensure only newer stuff put on the cache will come back when we probe the cache. We can then add the Secret to the cache. From there we reload the array. We can measure the time it takes for the elements within the array to come back. Things on the cache (as opposed to main memory) will come back quite fast, from there we can inspect the faster items for what we are looking for.

Buffer Overflow

Our vulnerable code uses the strcpy method in c, and it is given a buffer of size 500. This means that if the string we input is longer than 500 it will begin to overflow into parts of the stack that should not be changed. The goal of our attack is to change where the method returns to. We run the attack by giving the vulnerable file the following argument, which is condensed using python. $(python -c 'print "\x90"*425 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x14\xee\xff\xbf" * 10'). The first part is adding 425 instances of x90's which are no-ops that tell the computer to continue to the next instruction. The 425 value comes from taking 508, the buffer size plus 8 to get to the return section of the stack, then subtracting the size of the size of the rest of the text. The next part of the input is shellcode that opens a terminal with root privileges, the final part of the input, the

"\x14\xee\xff\xbf" * 10, is what the return statement will see. This is an address from within the buffer, the no-ops will then push that instruction forward until it hits the shellcode and then it will execute it. That is how the attack is performed. Some basic solutions that can prevent these attacks are address randomization, non-executable stacks, and stack guards.

Lessons Learned

Dirty Cow

This is my first exploration into an exploit of any sort. I found this to be fun and exciting. Because of this I am going to take computer security next semester. I learned that this exploit is a rather simple concept that flew under the radar for a long time that had major consequences. This shows me that sometimes the simple parts of an operating system are the ones that get overlooked and can have severe consequences. Along with this the patch against Dirty cow was as simple as disconnecting a part of the metaphorical circuit that made the exploit run. With this reasoning one can deduce that the simple solution is sometimes the answer versus over thinking the problem leading you down the rabbit hole of complexity. It’s a lesson many can use when they program anything.

Spectre

The attack itself is quite simple to understand, but the real learning lied in the preventive measures taken against Spectre. The annoying part about the exploit is that it is intrinsically a hardware fault. Unfortunately, 32 and 64-bit systems are laid out in a way where the address-space and user-space addresses are visible to the user at pretty much all times. Kernel address randomization is used to prevent the user from really being able to see the kernel-address space, but this has been shown in the last few years to not be the case. KAISER (kernel address isolation to have side-channels effectively removed) is an attempt to fix Spectre (and a similar exploit called Meltdown), but it doesn’t seem to work particularly well. An evolution of KAISER called kernel page-table isolation (KPTI) was implemented in Linux to hopefully address both but it unfortunately only covers Meltdown. The catch is that there have been now real-world exploits of either vulnerability, just proof of concept, but it is still very worrisome that the patch for Spectre seems so out of reach. In addition, we learned that the CPU manufacturers errors led to the supposed “patches” to decrease performance by a 5% to 30% margin, which is quite dangerous for someone with supposed shiny new hardware. We say supposed because Intel & Microsoft have said they have added a software mitigation to windows and the cpu software respectively. However there are many variations of Spectre and old CPUs seem to be affected the most. “[Spectre] will haunt us for a long time” according to Dell, as the name itself suggests. There is only one way to prevent the attack, develop better CPUs.

Buffer Overflow

Beyond learning how to perform a buffer overflow attack, the actual learning came from researching how the attack can be prevented. Address randomization is fairly simple to work around because you can repeatedly run the attack until the randomization gives addresses that allow your attack to work. So this prevention method really just slows down the attack without really preventing it from occurring. Non-executable stacks are more difficult to get around, and actually stop this specific version of the attack from working all together. If non-executable stack is turned on, or in Linux’s case if it is not turned off, then shellcode cannot be run from the stack. There are other ways to perform buffer overflow attacks, such as going into lib-c, but we did not do any research for that kind of attack. Finally, stack guards completely stopped the attack from working and would always force a segmentation fault in the program. This detects if overflow is occurring and throws an error if it is. There seemed to be no easy way of getting around this protection, at least nothing that could still be considered a buffer-overflow attack.

Conclusion

After completing this assignment, it was interesting to see a small taste of the spectrum of exploits that have arisen due to creative malicious thinking. We as a group felt confident with each of our exploits that we took on. We felt that our choice of operating system was a good one as it gave us a basic playground to try to be creative with our exploits without including anything complex that we would find in more secure operating system builds. With the collective mastery of four exploits we feel confident in pursuing more complicated exploits in the future.

References

Computerphile. “Buffer Overflow Attack - Computerphile.” *YouTube*, Computerphile, 2016, www.youtube.com/watch?v=1S0aBV-Waeo.

Du, Wenliang. Spectre Attack Lab. Seed Labs, 2018, https://seedsecuritylabs.org/Labs_16.04/PDF/Spectre_Attack.pdf

LIveOverFlow. (2016). Explaining Dirty COW local root exploit - CVE-2016-5195. Retrieved 2020, from https://www.youtube.com/watch?v=kEsshExn7aE

Seed “Buffer-Overflow Vulnerability Lab.” *Seed Labs*, 2014, www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/.

Corbet, Jonathan. “KAISER: Hiding the Kernel from User Space.” [LWN.net], 15 Nov. 2017, lwn.net/Articles/738975/.

Gruss, Daniel, et al. “KASLR Is Dead: Long Live KASLR.” SpringerLink, Springer, Cham, 3 July 2017, link.springer.com/chapter/10.1007/978-3-319-62105-0_11.