CS 5435 – Computer Security

Homework 4

Name: Bar Kadosh, Ben Kadosh *Fall 2019*

For this assignment, exercises in sections 2.1, 2.2, 2.3, and 2.4 require code deliverables, which have been submitted along with this report. These scripts contain comments that explain our code and approach. This report captures the responses to short answer exercises 3.1, 3.2, and 3.3.

Exercise 3.1

Briefly explain the vulnerabilities in target1.c, target2.c, target3.c, and target4.c.

In targets 0-3, vulnerabilities are exploited using more traditional stack smashing buffer overflow attacks where the goal is to use some combination of a NOP sled, the AlephOne shellcode, and a return address (building an exploit sandwich in a sense) to overflow the buffer, whether through one arguement or several, as we will discuss below.

In target0 for example, the code in the greeting function uses the strcpy function, which does not check the size of the temp1 parameter ahead of time – therefore, we can exploit this vulnerability and pass in a parameter that overflows the stack and allows us to gain root access. In target1, an added complexity is that we have to deal with a small buffer – as a result, we cannot pass the entire exploit as an args parameter, but rather we exploit a vulnerability in the environment parameter and store the exploit buffer in an environment variable. In target2, we build on the vulnerabilities noted previously. We now have a larger buffer to work with, but also have more constraints with input size and number of arguments. To work around this, we add another arg and exploit an additional vulnerability with an integer overflow in args[2]. Similarly, for target3 we again are working with a small buffer – thus, we exploit the vulnerability that we can store the exploit buffer in an environment variable.

For target4, we exploit a different vulnerability, namely one to circumvent a non-executable stack (W^X). To do so, we take advantage of already existing code that is loaded in memory – in our case, we accessed the system() and exit() memory addresses within libc (as well as in bin/sh) to exploit the vulnerability and gain root access.

Exercise 3.2

Does ASLR help prevent return-into-libc attacks? If so, how? If not, why not?

ASLR (Address Space Layout Randomization) does help prevent return-into-libc attacks, but it does not necessarily guarantee prevention 100 percent of the time, as we will discuss below. First though, let us recall that return-into-libc attacks are effective because they do not require the use of shell code and work in a W^X (non-executable) stack. Return-into-libc attacks exploit already existing code that is loaded in memory – in our case, as we saw in exercise 2.4, we accessed the system() and exit() memory addresses within libc to exploit the vulnerability and gain root access.

ASLR works to prevent return-to-libc attacks in that it randomizes offsets in process memory such that when dynamically linked libraries (such as libc) are mapped into memory, their locations are randomized. By randomizing the memory addresses of system() and exit() functions, we would no longer be able to definitively exploit the vulnerability we saw in exercise 2.4 because the memory address space positions would be constantly changing.

However, it is important to note that while ASLR does help in reducing the likelihood that return-into-libc attacks will succeed, ASLR will not entirely prevent them depending in large part on the architecture of the computer. While exploring a Stanford Research group's paper, "On the Effectiveness of Address-Space Randomization", they note: "In case of Linux on 32-bit x86 machines, 16 of the 32 address bits are available for randomization. As our results show, 16 bits of address randomization can be defeated by a brute force attack in a matter of minutes. Any 64-bit machine, on the other hand, is unlikely to have fewer than 40 address bits available for randomization given that memory pages are usually between 4 kB and 4 MB in size. Online brute force attacks that need to guess at least 40 bits of randomness can be ruled out as a threat, since an attack of this magnitude is unlikely to go unnoticed." Therefore, we can conclude that the architecture and the degree of randomization possible will impact the effectiveness of ASLR, but ASLR certainly does reduce the success rate of return-into-libc attacks.

Link 1: https://web.stanford.edu/ blp/papers/asrandom.pdf
Link 2: https://eklitzke.org/memory-protection-and-aslr

Exercise 3.3

Explain the stack protector countermeasure, which we turned off via the "-fno-stack-protector" option to gcc. Which of your exploits would be prevented by turning on this stack protection?

The -fno-stack-protector option we include when compiling the sploits disables the -fstack-protector option, which is the default setting when compiling. As the gcc gnu documentation states, by raising the -fstack-protector flag when compiling, the following occurs: "Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits."

The -fstack-protector we disable is a countermeasure that uses extra code and guard variables to evaluate whether a stack smashing attack has occurred and altered the integrity of the stack by checking if the guard variable was written over, thus indicating that a buffer overflow has occurred. As the documentation above mentions, guard variables are used, and they are more commonly referred to as canaries.

In a Linux Weekly News article discussing stack protection for GCC and the strong stack protection, the author discusses canaries, noting, "The basic idea behind stack protection is to push a "canary" (a randomly chosen integer) on the stack just after the function return pointer has been pushed. The canary value is then checked before the function returns; if it has changed, the program will abort. Generally, stack buffer overflow (aka "stack smashing") attacks will have to change the value of the canary as they write beyond the end of the buffer before they can get to the return pointer. Since the value of the canary is unknown to the attacker, it cannot be replaced by the attack." We now know the power of canaries and can see it is difficult to work around for attackers.

We do want to mention, however, that using canaries requires additional overhead to store the canary values and check if they have been overwritten, and as a result there are different levels of protection that can be enabled in gcc. In the gcc gnu documentation we noted earlier, there is language discussing the fstack-protector-all and fstack-protector-strong flags. As we noted, -fstack-protector adds a guard to functions that call alloca and functions with buffers larger than 8 bytes, whereas fstack-protector-all protects all functions and strong includes additional functions that have local array definitions or have references to local frame addresses.

At this point, we have highlighted the necessary information to evaluate how these protections would impact our exploits. Because each of the sploits from sploit0 - sploit4 are writing beyond the end of the buffer, they would all trigger a change in the canary variables and thus be prevented if the stack protection were turned on. We do have to note, however, that this assumes the stack protection would be fstack-protector-all or fstack-protector-strong. Because target1 and target4 have functions where the buffer is less than 8 bytes, we would advise to use one of the two options above to ensure stronger protection mechanisms.

Link 1: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
Link 2: https://lwn.net/Articles/584225/