CS 5435 – Security and Privacy Concepts in the Wild

Homework 4

Name: Arystan Tatishev(at855) , Nikhil Reddy(nc463) *Spring 2024*

Exercise 3.1

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

1. **target1.c**

   Buffer Overflow: The foo function declares a character array name with a size of 4 bytes, but it uses strncpy to copy up to 12 bytes from the temp argument into name. This can lead to a buffer overflow if the length of the input string (temp) is greater than or equal to 12 characters. Buffer overflows can potentially allow an attacker to overwrite adjacent memory locations, leading to various security issues, such as code execution or data corruption.

   Lack of Input Validation: The main function passes the second command-line argument (argv[1]) directly to the foo function without any validation. This means that the second input it could trigger the buffer overflow vulnerability.

2. **target2.c**

   Buffer Overflow: The greeting function uses the strcpy function to copy the contents of temp1 into the name buffer, which has a fixed size of 400 bytes. If the length of temp1 is greater than or equal to 400 characters, a buffer overflow can occur, potentially allowing an attacker to overwrite adjacent memory locations and potentially leading to code execution or data corruption.

   Integer Overflow/Underflow: The atoi function used to convert argv[2] to an integer value (input_size) does not perform any validation or error checking. If argv[2] contains a value that exceeds the range of a short integer type, an integer overflow or underflow can occur, leading to undefined behavior and potential security vulnerabilities.

   Lack of Input Validation: While the code attempts to validate the length of argv[1] against input_size, it does not validate the actual contents of argv[1]. An attacker could potentially craft a malicious input string that exploits vulnerabilities in the printf function or other string-handling functions used later in the code.

3. **target3.c**

   Buffer Overflow: The copyFourInts function uses the memcpy function to copy data from the values buffer into the intvalues array, which is declared to hold 4 integers (16 bytes on most systems). However, the memcpy operation copies sizeof(int) * 5 bytes from values into intvalues, which is 20 bytes on most systems. This means that the memcpy operation writes 4 bytes beyond the end of the intvalues array, potentially overwriting adjacent memory locations.

4. **target4.c**

   Buffer Overflow: The foo function declares a character array name with a size of 4 bytes, but it uses the strcpy function to copy the contents of the temp argument into name. The strcpy function does not perform any bounds checking and will continue copying characters until it encounters a null terminator (\0). If the length of the input string (temp) is greater than or equal to 4 characters, a buffer overflow will occur, overwriting adjacent memory locations.

Exercise 3.2

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

Yes, Address Space Layout Randomization (ASLR) helps prevent return-to-libc attacks by randomizing the memory layout of a running process, making it more difficult for an attacker to predict the addresses of library functions and other code sections. This prevents the attacker from exploiting a buffer overflow vulnerability to overwrite the return address on the stack.

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 stack protector is a security countermeasure implemented in modern compilers like GCC to mitigate buffer overflow attacks on the stack. It works by placing a guard variable on the stack and checking it before return from a function. If the variable has been modified, the program will terminate and alert the user that an attack has occurred.

1. **sploit0.c**

   This exploit would be prevented by the stack protector. It overflows the buffer str with NOP instructions and shellcode, followed by overwriting the return address. This is a stack-based buffer overflow attack, which the stack protector is designed to mitigate.

2. **sploit1.c**

   This exploit would not be prevented by the stack protector. It does not rely on overwriting the return address on the stack. Instead, it passes the shellcode as an environment variable.

3. **sploit2.c**

   This exploit would be prevented by the stack protector. It overflows the buffer str with NOP instructions and shellcode, followed by overwriting the return address. This is a stack-based buffer overflow attack, which the stack protector is designed to mitigate.

4. **sploit3.c**

   This exploit would be prevented by the stack protector. It overwrites the return address on the stack after setting up a buffer with NOP instructions.

5. **sploit4.c**

   This exploit would not be prevented by the stack protector. It does not rely on overwriting the return address on the stack. Instead, it constructs a sequence of instructions (system, exit, and /bin/sh) and passes them as an argument to the target program.