ELEC 377 Lab 5: Software Security – Description of Solution

Section 003, Group 16 – Thursday Lab

Erhowvosere Otubu, #20293052

Ivan Samardzic, #20296563

Date of Submission: December 5, 2023

Problem Solution

The purpose of Lab 5 is to examine a common security vulnerability, the stack overflow. Evaluating the security of applications often involves deliberately testing their robustness in a controlled environment. In this context, various organizations employ tiger teams compromising both the defensive “blue team” and the offensive “red team”. The blue team focuses on defense, while the read team actively engages in simulated attacks, aiming to identify potential security vulnerabilities within a network. The two main objectives of this lab are to: (1) compromise a server program using a standard buffer overflow, (2) brute force guess the secret password that is revealed.

Solution Approach

The following sections will explain in-depth the process of the group’s solution. The problem involves exploiting a buffer vulnerability in a server program to gain unauthorized access to its environment variables. The solution includes developing a precise exploit that allows this access.

Shell Code

The shell code was intricately designed to exploit the server’s memory vulnerability. It involved writing assembly code to execute within the server’s memory space, comprising ‘no-operation’ (NOP) instructions and the payload. The payload hijacks servers control flow to access environment variables. Size and structure were crucial to avoid adjacent memory corruption.

Self Compromise

The self-compromise process used the ‘selfcomp.c’ code file for simulating a controlled buffer overflow. This is a file that executes the exploit internally. The group modified the program where we applied buffer overflow principles to exploit vulnerabilities in the server’s memory. The approach was to find the exact input length to overflow the buffer and overwrite the return address, then manipulate it to point to the shell code, diverting execution flow. This step was pivotal for understanding buffer overflow mechanics. The length of the compromise string was padded with NOPs to be 159 after undergoing iterative adjustments through trail and error. This specific length was identified through numerous executions that induced core dumps and segmentation faults, ensuring the integrity of the return address. Notably, the determined the return address was 0x7FFFFFFFE028.

Client and Server

In the client-server model, the ‘client.c’ code file was altered to send a string that would trigger the server’s buffer overflow. This a file that a client will use to compromise the server program. The client’s role was to transmit the malicious payload, while the sever, upon receiving this input, experience a buffer overflow, executing the shell code. This required precise control and debugging to ensure successful payload execution. Just like in the Self-Compromise section, the length of the compromise string was padded with NOPs to be 224 after several attempts of trial and error. This specific length was identified through numerous executions that induced core dumps and segmentation faults, ensuring the integrity of the return address. As a result, the return address was 0x7FFFFFFFDEB0.