CS 6324: Information Security **Spring 2021**

Project #2 - Software Security

In this project, you will perform a series of software vulnerability exploits. You will explore unsafe and insecure programming techniques and will evaluate the efficacy of operating system defenses against them.

Instructions

Due date & time 11:59pm CST on April 16, 2021. Submit your report to eLearning by the due time.

- For these questions, you will need to access a virtual machine that is set up on `edgar.utdallas.edu`. Each student will use their account on `edgar.utdallas.edu`. You will receive an email with this information.
- Use the UT Dallas VPN if you connect to the server from outside the campus: `https://oit.utdallas.edu/howto/vpn/`.
- After logging into `edgar`, you can access the virtual machine via ssh: `ssh attackme`. The username/password will be the same as that of `edgar.utdallas.edu`.
- The source files you need can be found in your home directory. The targets will be in a tar archive in the directory `~/targets/`.
- The source code of your answers needs to be in the directory `~/exploits/` by the deadline. You need to hardcode relative paths in your source code to execute the targets.
- Any code you write should run in the virtual machine in `edgar` with no errors.
- The written portion of the project must be typed. Using Latex is recommended, but not required. The submitted document must be a PDF (no doc or docx are allowed).
- Most of the points for each question will be for a correct exploit. If you answer a question without correctly exploiting the target, no credit will be given.
- **When writing an exploit in C, you should use a function like *execve* to launch the target, not a function like system. Passing in *null* for the environmental variables so that it will be consistent and repeatable from run to run.**
- You are **NOT** allowed to modify the source code for any of the targets.
- To compile the source code you can run `make` within the target directory. You can create a similar make file in the exploit directory or compile them individually using gcc.
- If you want to use extra days in this project, please notify TA (dongpeng.liu@utdallas.edu) by the due time. Otherwise, your account on `edgar` will be locked and you will not be able to modify your code.

1 (20 pts) ~~Simple Command Line Buffer Overflow~~

`target1` is a program that takes a directory as input, and tells the user how to use the command `ls` to list the contents of the directory. Suppose that this program is setuid root. You will login as a normal user, and your goal is to pass an argument to the program so it will start a root shell.

- In the exploits directory, write a shell script `exploit1.sh` that passes the attack string to the target and performs the attack.
- Identify the exact vulnerability in the program that you exploited (i.e. function name and line number)
- Explain your attack strategy. That is, explain how you determined the correct input to pass and what commands are executed.

2 (20 pts) ~~Buffer Overflow To Rewrite a Return~~

1. **The attack**: `target2` is a program that takes a customer's name as the input, and prints a coupon. Assume that each customer can only execute the program once, so he/she can only get one coupon. Your goal is to pass some argument to the program so it will repeatedly print coupons. In other words, the argument will make the program execute the function *coupon* repeatedly. Note: To get full credit, the function `coupon` has to execute an *infinite* number of times. If it only executes twice, then you will get half the points.
   - In the exploits directory, write a C program `exploit2.c` that passes the attack string to the target and performs the attack.
   - Identify the specific bug/vulnerability that made your attack possible.
   - Describe your attack strategy. That is, describe the memory addresses involved in your attack, and explain how the attack made the program print an unlimited number of coupons.
2. **The defense**: The machine `edgar` has an updated operating system with some stack defenses activated.
   - Repeat the attack on `target2` outside the virtual machine. Did the attack work? Comment on your results (i.e. explain why)
   - Propose **two** different operating system and/or compiler/programming language defenses that can be used to prevent this attack from working. Discuss the advantages, disadvantages, and feasibility of the proposed defenses.

3 (20 pts) ~~Return to libc~~

1. **The attack**: `target3` is a program that scans several network packets and checks if the traffic (concatenation of the packets) matches any virus signatures. Suppose `target3` is setuid root. You will login as a normal user, and the goal is to pass argument(s) to the program to start a root shell. You need to assume that the stack is **not** executable. Therefore, you **cannot** change the return address to the shellcode in the stack.

- Draw the layout of the stack frame corresponding to the function `is_virus` directly after the local variables are initialized. For each element on the stack, provide its size (assuming a 32-bit architecture).
- In the exploits directory, write a C program `exploit3.c` that performs the attack.
- Identify the specific bug in the program and vulnerability in the operating system that made your attack possible.
- Describe your attack strategy. That is, explain what memory addresses you used and how you figured out those addresses.

2. **The defense**: Try repeating the above attack on `edgar` (outside the virtual machine). The attack should become more difficult now.
   - Are you able to get the attack to work? If so, explain your method. Otherwise, explain what prevented you from completing the attack.
   - What specific mechanism(s) make the attack more difficult?

4 (40 pts) ~~Format String Attacks~~

In this section, you are given a program with a format-string vulnerability; your task is to develop a scheme to exploit the vulnerability. You can find the source code for the program `target4.c`.

In `target4.c`, you will be asked to provide an input, which will be saved in a buffer called `user_input`. The program then prints out the buffer using `printf`. Unfortunately, there is a format-string vulnerability in the way the `printf` is called on the user inputs. We want to exploit this vulnerability and see how much damage we can achieve.

The program has two secret values stored in its memory, and you are interested in these secret values. However, the secret values are unknown to you, nor can you find them from reading the binary code (for the sake of simplicity, we hardcode the secrets using constants 0x44 and 0x55, but you can pretend that you don't have the source code or the secrets). Although you do not know the secret values, in practice, it is not so difficult to find out the memory address (the range or the exact value) of them (they are in consecutive addresses), because for many operating systems, the addresses are exactly the same anytime you run the program.

- Draw the layout of the stack frame corresponding to the main function directly after the local variables are initialized. For each element on the stack, provide its size (assuming a 32bit architecture).
- Provide the specific inputs (i.e. both the integer and the string) that you need in order to crash the program. Explain why the program crashes with your input.
- Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the *address* of the variable `secret[0]`. Explain why you think this is the correct address. Hint: you can use `gdb` to verify that your answer is correct.
- Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the *value* of `secret[0]`. Explain your strategy.

- Based on your knowledge of how arrays are stored on the heap, calculate the address of `secret[1]`.
- Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the value of `secret[1]`. Explain your strategy.
- Provide the specific inputs (i.e. both the integer and the string) that you need in order to modify the values of *both* `secret[0]` AND `secret[1]`. Explain your strategy.
- Does address space randomization make this attack more difficult? Explain.
- What other operating system defenses can be used to prevent this attack? Explain.