CS3235 AY2021/22
Memory Security Coding Assignment

[33 points]

Due Date : Friday, May 6th, 2022, 23:59 SGT

1 Assignment Summary

In this assignment we will launch 5 different attacks that exploit memory security vulnerabilities. The tasks are designed in a "Capture The Flag" (CTF) style, where the goal is to find a hidden flag (a number) by exploiting vulnerabilities in the provided programs.

While some are more challenging than others, all of the attacks are targeting vulnerabilities covered in the lectures and the tutorials (Buffer Overflow, Format String, ROP, etc.). Therefore, we suggest that you refer back to the lecture and tutorial materials while you are attempting this assignment.

We will be using Github Classroom for autograding this assignment. Please follow the instructions below to set it up.

1. Follow previous instructions to set up the Docker. Make sure that you can use the provided script to run the tutorial demos.
2. Download the assignment zip file (on Luminus) and extract it to a folder. We will use `ASSIGNMENT_FOLDER` in place of the folder you extract the files to.
3. In `ASSIGNMENT_FOLDER`, execute `./qemu-start`. This will start an emulated x86-64 Linux system. You can shut the emulated system down with `./qemu-shutdown`. You will need to repeat this step again when you want to work on the assignment later.
4. In `ASSIGNMENT_FOLDER`, execute `./init`. This may prompt you for your NUSNET ID. After finishing this step, you will be able to find the files for individual cases inside `ASSIGNMENT_FOLDER/student`. You will be working with those files for this assignment.
5. Visit `https://classroom.github.com/a/KyXz5CDG`. Choose your NUSNET ID from the list (unnecessary if you already did this for the previous assignment) and accept the assignment.
6. Once you have joined the Github classroom and accepted the assignment, you will automatically get a Github repository for this assignment. Clone that repository to your local environment. We use `REPO_FOLDER` to represent your local repository folder.

2 How to Submit

Normally, the individual case folders in `ASSIGNMENT_FOLDER/student` are where you prepare your exploits. When you wish to submit, execute `./prep-submit REPO_FOLDER` inside `ASSIGNMENT_FOLDER`. This will copy relevant files to `REPO_FOLDER` and arrange them properly. Then commit and push the changes in `REPO_FOLDER` to your Github repository. A Github Action workflow will run to evaluate your submission, the results of which should be available within minutes. To see the evaluation progress and results, you can then go to the "Actions" tab of your Github repository.

For more information, take a look at the official documentations of Github Classroom.

- *You may find a folder named* `.github` *inside* `REPO_FOLDER`. *Please do not modify anything inside this folder. Evaluation results for commits with modified* `.github` *will be considered as invalid.*
- *The file* `.studentid` *should be included in your commits. Make sure that it contains your correct NUSNET ID.*

3 Local Evaluation

You can also evaluate your answers locally. In `ASSIGNMENT_FOLDER`, execute `./run-eval REPO_FOLDER` or `./run-eval ASSIGNMENT_FOLDER/student`. It is expected that there should be no discrepancy between the local and the online evaluation results. In case you encounter a discrepancy, please ask the TAs for help.

NOTE: Local evaluation results are for your own reference only. The final marks you get for this assignment are decided by your submissions on Github.

4 Tasks

This assignment consists of 5 tasks, each requiring you to obtain a flag value (a decimal string). After setting this assignment up, you can find the files for the tasks in `ASSIGNMENT_FOLDER/student`. Each task is included in its own directory and has been compiled during the setup process.

Under the directory of each task, a file named `flag.txt` contains the flag value you need to print to the standard output in a successful exploit. In some tasks, an executable called `flag` is also supplied, which will print out the flag for you once executed. This allows you to target executing `flag` instead of reading and printing `flag.txt`.

The specific goal for each task is:

- Create exploit files (listed separately for each task) and place them under the task directory;
- In `ASSIGNMENT_FOLDER/student`, execute `./run <task-name>` where `task-name` is the name of the task (a through e), and the flag appears in the standard output (your terminal in this case).

You can use the scripts `gdb` and `peda` inside `ASSIGNMENT_FOLDER/student`. To run vanilla GDB on a certain task, execute `./gdb <task-name>`. Execute `./peda <task-name>` to run GDB-peda instead.

In both vanilla GDB and GDB-peda, two additional commands `runwithinput` and `startwithinput` are available. Those are the same as `run` and `start` except that they start the program with the right input and argument files. You should use them instead of `run` or `start` most of the time for this assignment.

IMPORTANT NOTE:

- When you use the scripts `run`, `gdb` and `peda` to run a vulnerable program, as well as during evaluation, the relevant files will be relocated to `/assignment/<task-name>/` regardless of what your `ASSIGNMENT_FOLDER` is. For example, when running case A, the `flag` executable will be located at `/assignment/a/flag`.
- ASLR is turned off for all of the tasks (the setup has already taken care of this). For all tasks except for cases B and C, the stack is executable as well (see `Makefile` under each task directory).
- During evaluation the flags are randomly generated. Never assume that the values will be the same as what you find in the files generated inside `ASSIGNMENT_FOLDER/student`.
- Do not change the source code, as your answer will be evaluated only with regards to the source code as it is provided to you.
- Do not use externally available automatic exploit generation toolkits to create your exploits. Please discuss only within your own team. You can, however, consult online resources such as Stack Overflow.

Hints:

- If the placement of the bytes is not clear immediately, you can fill exploit files with random characters and observe how the stack is manipulated in GDB.
- It is recommended that you create the exploit files using scripts, so that you can include non-printable characters in your exploits.
- Overwriting certain local variables in the program might cause it to fall in infinite loops.

4.1 Case A (5 points)

Exploit files: `exploit1`, `exploit2`

The vulnerable program reads data from two files `exploit1` and `exploit2` which are controlled by you. It then constructs a buffer based on the data, and something wrong could happen here (find out what it is!). You need to exploit this vulnerability to execute `./flag`, which will print out the flag for you.

4.2 Case B (6 points)

Exploit files: `input`

The vulnerable program will print out the flag whenever it finds that `jackpot` equals a magic value. Normally `jackpot` is generated randomly and you cannot guarantee that it is set to the magic value. Your goal is to exploit a vulnerability to set `jackpot` to the magic value by constructing the right data for the standard input.

4.3 Case C (6 points)

Exploit files: `exploit`, `input` (containing the data for standard input to the vulnerable program)

The vulnerable program reads a specified number of bytes from the file `exploit`. Here the program does not do things properly, giving you the possibility of hijacking its control flow. Since **the stack is not executable**, you must reuse existing code in the address space of the program. Note that **the program `./flag` is not supplied and therefore you need to control the vulnerable program to open `./flag.txt`, read data from it, and print the data out to the standard output**.

NOTE: Do not execute any external program such as `cat` to access `./flag.txt`.

4.4 Case D (7 points)

Exploit files: `input` (containing the data for standard input to the vulnerable program)

The vulnerable program sets specified locations inside a buffer based on data read from standard input. There are bound checks but they are not done properly, allowing you to manipulate the vulnerable program to execute `./flag` and get your flag in the standard output.

4.5 Case E (9 points)

Exploit files: `args` (containing the argument to be passed to the vulnerable program)

The vulnerable program uses a custom memory allocation library. It is however done in an incorrect way, and you can exploit this and supply the right argument to the vulnerable program to achieve the goal of executing `./flag` and printing out the flag.

5 Submission Format

You need to place the exploit payloads under the directories for their respective tasks. The `prep-submit` script can help you with this (see § 2). The files you need to submit are the "exploit files" listed above. Carefully follow the instructions in § 4.

Important Note. The file names or locations should not be changed, otherwise the evaluation scripts will not be able to recognize your solution.

Time Limit. We limit the execution time of each of your exploits to 5 seconds.

6 Report

Aside from submitting the required files for each task to your Github repository, you also need to write a short report to describe your methods and submit it on Luminus before the deadline.

For every task you have finished, write a few sentences to briefly explain the sequence of events that lead to the appearance of the flag, including the part that your answer plays.

In the report, please also indicate the hash of the Github commit you want to use for the final grading.

7 Grading

This assignment is worth 33 points. We will use the evaluation results from Github Classroom.