CS 453/698 Assignment 2

TA: Adeola Tijani (atijani@uwaterloo.ca)
Office hours: Wednesdays 1pm to 2pm, online via BBB (access code `aofh7n`)

Due date: February 28, 2025

Bugs That Are Hard to Catch

Fuzzing is used by both attackers and defenders to hunt for potential issues in software. However, there exists code patterns that pose unique challenges for fuzzers to analyze in general. The goal of this assignment is to help you understand some of the *limitations* of a state-of-the-art fuzzer (AFL++) and to get a feeling of its advantages and disadvantages with some hands-on experience. We provide details on how to run AFL++ in later part of the assignment.

In short, your goal is to craft **five (5) different programs**, **each with a single bug intentionally embedded** and check whether AFL++ can find the bug (evident by a program crash). If AFL++ fails to find the bug *within a computation bound*, you have found a limitation (i.e., a weakness) of AFL++. In this assignment, the computation bound is **15 minutes of fuzzing per each program**.

Each of the **five** programs you crafted needs to expose **a different type of weaknesses** of AFL++ and you need to write a **one-sentence** (less than 25 words) summary for the weakness your program exploits. This summary should be put as the first line of a `README` file (see package structure below).

- While the definition of a "weakness" is hard to be precise, a rule of thumb is — if two programs are simple tweaks of each other (e.g., iterate a loop twice vs iterate a loop three times), they are considered as exploiting the same weakness of AFL++. This should be evident in the textual description in the `README` file.
- Each crafted program should exploit one weakness only, if your program exploits two or more weaknesses, we only count it as a program that exploits the weakness you put in the `README` file.
- **Tips of advice**: you can consult search engines or large-language models for a list of limitations or weaknesses of AFL++ (or fuzzing in general).

To confine the scope of the analysis and standardize the auto-grading process, we will NOT accept arbitrary programs for this assignment. Instead, you are encouraged to produce *minimal* programs and prepare a package for each program. The package should contain not only the program code but also information that bootstraps analysis and shows evidence that a bug indeed exists.

Specifically, each package MUST be prepared according to the **requirements and restrictions** below. Failure to do so will result in an invalid package that can't be used to score this assignment component.

- Each submitted package MUST follow the directory structure below:

```
<package>/
    |-- main.c    // mandatory: the only code file to submit
    |-- input/    // mandatory: the test suite with 100% gcov coverage
    |    |-- <*>  //   test case name can be arbitrary valid filename
    |-- crash/    // mandatory: sample inputs that crash the program
    |    |-- <*>  //   sample name can be arbitrary valid filename
    |-- README    // mandatory: an explanatory note on the weakness of AFL++
```

- Include all source code of the program in **one and only one** `main.c` file. This `main.c` is the only code file you need to include in the package. DO NOT include `interface.h`.
  - Only valid **C** programs are allowed. DO NOT code in C++.
  - The size of `main.c` should NOT exceed 8KB (i.e., 8×2^{10} bytes).
- Your program can only invoke three library calls, all provided in `interface.h` available here.
  - `ssize_t in(void *buffer, size_t count)`
    which reads in at max `count` bytes from `stdin` and store then in `buffer`. The return value indicates the actual number of bytes read in or a negative number indicating failure.
  - `int out(const char *buffer)`
    which prints the `buffer` string to `stdout`. The return value indicates the actual number of bytes written out.
  - `void abort(void)`
    which forces a crash of program. Note that this is NOT the only way to crash a program.
- Your program can only take input from `stdin` using the provided `in()` function in `interface.h`. It should NOT take input from command line arguments nor environment variables. The size of input acquired from `stdin` should NOT exceed 1024 bytes.
- Your program MUST be *compatible* with the version of AFL++ used in this assignment without special modification to the tool. In other words, your program can be analyzed using the tool invocation command provided in later part of the assignment.
- Your program should have **one and only one bug** that is intentionally planted. If AFL++ finds a crash in your program, even the crash is not caused by the intended bug, AFL++ is considered successful in analyzing your program and this package cannot be used to claim victory in exploiting a weakness of AFL++.
- You need to provide a set of test cases that achieves 100% coverage (see details on `gcov` below).
  - Each test case should NOT crash the program, i.e., exiting with a non-zero status code.
  - Each test case should complete its execution in 10 seconds.
  - Each test case should NOT exceed 1024 bytes in size.
- You need to provide **at least one** sample input that can cause the program to crash by triggering the planted bug — this is to provide evidence on the existence of the bug.
- Avoid using the following features that are known limitations for most program analysis tools:
  - DO NOT use any other library functions (including `libc` functions). The only permitted library routines are given in the header file `interface.h`.
  - DO NOT use floating-point operations in your program.
  - DO NOT use inline assemblies in your program.
  - DO NOT use multi-threading. The entire program logic must be able to execute end-to-end in a single process and a single thread.

Each submitted package will be analyzed by AFL++ independently. If AFL++ finds a bug (even not the planted one), it is considered a success in analyzing your program. Otherwise, the bug has managed to "evade" the detection from AFL++ and the package will be counted towards the scoring.

A maximum of ten (10) packages can be submitted. Each package that exploits a unique weakness of AFL++ will be awarded **10pts**, up to a total of **50pts**. In other words, a minimum of five (5) packages are needed to receive full marks, provided that they all exploit different weaknesses of AFL++.

1.0 Environment preparation

Although we provide two modes of preparation, we highly recommend you to setup a local VM (i.e., Option 1) for this assignment as long as your machine can support it. There are two reasons for this recommendation: 1) the ugster platform is limited in resources and cannot accommodate everyone, 2) a local VM enables more agile development and a more controllable environment.

Option 1: local VM

You need to create a fresh VM with Ubuntu version 22.04.5 LTS. You can choose your favorite VM management tool for this task, including but not limited to VirtualBox, VMware, Hyper-V, etc. Once your Ubuntu VM is ready, log / SSH into your account and follow the common provision steps. You will need `sudo` passwords for the provision.

Option 2: ugster VM

If option 1 is not feasible on your machine, you can opt to use the ugster VM, as you experienced with Assignment 1. If you go with this option, DO NOT reset or destroy your VM and DO NOT re-register your account. Keep what you have with A1 (as the TAs might occasionally need to access your VM for A1 grading) and do A2 in a new directory. To be more specific, SSH into your VM and follow the common provision steps below.

Common provision step for both options

After logging in / SSH into your VM, you can provision your VM for A2 with the following commands:

```
git clone https://github.com/meng-xu-cs/cs453-program-analysis-platform.git
cd cs453-program-analysis-platform
./scripts/ugster-up.sh
```

Upon successful completion, you will see an `==== END OF PROVISION ===` mark in the terminal.

- The `./ugster-up.sh` script will take a couple of minutes to finish so you might find utilities such as `tmux` or `screen` useful in case of unreliable SSH connections.
- During the process, you *might* be prompted to upgrade or restart system services. Simply hit "Enter" to go with the default choices suggested by `apt`.
- After provision, the entire VM will take about 20GB storage on disk.

1.1 Coverage tracking with `gcov`

gcov is a tool you can use in conjunction with `gcc` to test code coverage in your programs. In a nutshell, it tracks the the portion of code that is "covered" by a concrete execution at runtime and aggregates the coverage results from multiple runs to produce a final coverage report.

You can use the `run-gcov.sh` script to check the coverage of your package. The script is available to you, as a reference implementation, under `scripts/run-gcov.sh` in the cloned repository. In general, the script performs the following steps:

```
// Step 1: compile your code with gcov instrumentations
$ gcc -fprofile-arcs -ftest-coverage -g main.c -o main

// Step 2: execute each of your input test case
$ for test in input/*; do main < ${test}; done

// Step 3: collect and print detailed coverage
$ gcov -a -b -c -o ./ -t -H main.c
```

Check gcov documentation for details on how to interpret the coverage report.

```
// Step 4: optionally, pipe step 3 to grep for a summary of statistics
$ gcov -a -b -c -o ./ -t -H main.c | grep "blocks executed"
```

If you read 100% coverage for all functions in your console, it means your test suite (provided under the `input/` directory) has achieved complete coverage in `gcov`'s perspective.

1.2 Fuzzing with AFL++

We use the open-source AFL++ fuzzer, in particular, version `v4.31c` which is the most up-to-date stable release of AFL++ as the assignment is developed (released February 10, 2025). You may want to read a bit of details on the project page about AFL++ and fuzzing in general.

AFL++ is provisioned into the VM as a Docker image tagged as `afl`. Once in the VM, you can use the following command to run the Docker image, get an interactive shell, and explore around.

```
docker run \
    --tty --interactive \
    --volume <path-to-your-package>:/test \
    --workdir /test \
    --rm afl \
    bash
```

You can use the `run-afl.sh` script to fuzz your package. The script is available to you, as a reference implementation, under `scripts/run-afl.sh` in the cloned repository. In general, the script runs-off the `afl` Docker image and performs the following steps:

```
// Step 1: compile your code with afl instrumentations
$ afl-cc main.c -o main

// Step 2: start fuzzing your code
$ afl-fuzz -i input -o output -- main
```

The output of the AFL++ fuzzing results are stored in the `output` directory.

1.3 Example

The provided repository contains a sample package under directory `scripts/pkg-sample` to illustrate how a package should look like, with the addition of `interface.h` and `.gitignore` which shouldn't be submitted. The code can also be found on GitHub as well.

You can test out the sample package inside the VM via:

```
$ cd cs453-program-analysis-platform/scripts
$ ./run-gcov.sh pkg-sample
$ ./run-afl.sh pkg-sample
```

The output should obviously show that 1) this package does not provide 100% code coverage and 2) it has a bug (a crash) that is found by AFL++ quickly.

Despite that this package sample cannot "evade" AFL++ and does not reveal any of its weaknesses, feel free to duplicate this template package to bootstrap your package preparation that can eventually "evade" AFL++.

1.4 Grading platform

We also open the grading platform for this assignment to you, which can be accessed through this link. You can find detailed instructions on the landing page. NOTE: you will need to use the campus VPN if you can't access it from outside.

In short (and see the landing page for details),

- to submit a package to the server, ZIP the package directory, send a HTTP `POST` request to `http://ugster72d.student.cs.uwaterloo.ca:9000/submit` with the ZIP content as body. You will get a hash string as the package identifier.
- to check the analysis result (after the server has processed the package), send a HTTP `GET` request to `http://ugster72d.student.cs.uwaterloo.ca:9000/status/<hash>` where the `<hash>` is the hash value you get from the package submission phase.

Some important things to note regarding the submission system:

- Submitting to the evaluation platform DOES NOT count towards assignment submission. To make the final submission, please follow the instructions in the Assignment instruction file and make the final submission on LEARN. We DO NOT accept package hashes as proof-of-submission.
- This evaluation server is NOT well tested, meaning, there might be bugs. If you observe weird behaviors, please make a Piazza post and we will try to investigate as soon as possible.
- Please DO NOT rely on the submission server for iteration. The server is NOT designed to be a system for quick feedback. For each package, the server tries to analyze it to the fullest extent (i.e., a full 15-minute fuzzing per each package). A better strategy is to develop the package using a local platform and only use the server for final confirmation.

Deliverables

You are required to hand in a single compressed `.zip` file for this assignment: Unzipping the file should yield up to 10 packages where each package is a directory itself. We will count the first 10 packages only (by alphabetical order) if more than 10 packages are found.

Submit your files using Assignment 2 Dropbox on LEARN.

Write-up. We use an auto-grader to check the code submitted for this assignment and the TA will also read the `README` file in each package to determine whether you are indeed exploiting different weaknesses of AFL++ in each package. While efficient, the auto-grader can only provide a binary pass/fail result, which rules out the possibility of awarding partial marks for each task. As a result, we also solicit a write-up submissions.

The write-ups can be optionally included in each package as `WRITEUP` files. Typical things to be put in the write-up include:

- How the code you submitted is expected to work
- How you manage to get certain hardcoded information in the code
- Explanation on critical steps / algorithms in the code
- Any special situations the TAs need to be aware when running the code
- If you do not complete the full task, how far you have explored

On the other hand, if you are confident that all your code will work out of the box and can tolerate a zero score for any tasks on which the auto-grader fails to execute your code, you do not need to submit a write-up (or you can omit certain tasks in the write-up).