CS 489/698 Assignment 3

TA: Liyi Zhang (l392zhang@uwaterloo.ca)
Office hours: to be announced on Piazza posts

Due date: March 22, 2024

Bugs That Are Hard to Catch

Program analysis is used both by attackers and defenders to hunt for potential issues in the software. However, there are always code patterns that pose unique challenges for these tools to analyze. The goal of this part of the assignment is to help you understand the *limitations* of state-of-the-art program analysis tools and get a feeling on their advantages and disadvantages with some hands-on experience.

In short, your goal is to craft programs **with a single bug intentionally embedded** and check whether different program analysis tools can find the bug (evident by a program crash). If the program analysis tool fails to find the bug *within a computation bound*, you have found a potential limitation in this program analysis tool and will be awarded full marks for that component.

In this assignment, the computation bound is **15 minutes per analyzer $\times$ 3 analyzers**:

- AFL++: as a representative fuzzer
- KLEE: as a representative symbolic executor
- SymCC: as a representative concolic executor (and hybrid fuzzer)

We will provide details on how to run these tools in later part of the assignment.

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. Failing 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.md // optional: an explanatory note on the code and embedded bug
```

- 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 256KB (i.e., 256×2^{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 **all** program analysis tools used in this assignment without special modification to these tools. 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 any of the tool finds a crash in your program, even the crash is not caused by the intended bug, the tool is considered successful and this package cannot be used to claim victory over that tool.
- 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.

- 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 features that are known limitations for most program analysis tools, including:

  - 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 package will be analyzed by all program analysis tools. For any tool X, if the tool finds a bug (even not the planted one), it is considered a success. Otherwise, the bug has managed to "evade" the detection from tool X and the package will be counted towards the scoring of tool X evasion.

A maximum of 10 packages can be submitted. You will have full points for tool X evasion as long as you have one package that "evaded" the detection from tool X. Below is the more verbose criteria:

- [**10 pts**] at least one package that evades AFL++
- [**10 pts**] at least one package that evades KLEE
- [**10 pts**] at least one package that evades SymCC
- [**10 pts**] one package that evades them all

If you are confident that one package can be used to score all four components, feel free to submit one package only. However, to be on the safe side, it is highly advised to submit multiple packages with different strategies to confuse these program analysis tools.

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 22.04.4 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 (e.g., Mac computers with Apple silicon chips), you can opt to use a similar VM setup as you experienced with Assignment 1. To be specific, you can use the portal.sh script to create a new VM, destroy an existing VM, or ssh into a running VM. To learn more about portal.sh and how to use it, please visit http://ugster72d.student.cs.uwaterloo.ca:8000/. Unfortunately, in this assignment, we cannot re-use the VMs you have in A1 due to different resource requirements. This means that you will need to **register again for the new platform with a new pair of keys**. However, in this time, take care of your private key!

- The way to wipe out and reset your VM is through `./portal.sh destroy`. Please do not use `./portal.sh register` for this purpose.
- NOTE: if you lose your private key after successful registration, you will be locked out of the system completely. There is no way back. You can request a key reset on Piazza, and every reset means a **1 pt** deduction on the whole assignment.

After you register to the `ugster` platform and have your VM launched, please SSH into your VM and follow the common provision steps.

Common provision step for both options

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

```
git clone --recurse-submodules --shallow-submodules \
   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.

- Pay attention to the `--recurse-submodules` flag in the `git clone` command, it is important.
- The `./ugster-up.sh` script will take quite some time to finish (about an hour or even two hours) 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 50GB – 60GB 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 coverage
$ gcov -o ./ -n main.c
```

If you read a 100% coverage 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 `4.10c` which is the most up-to-date stable release of AFL++. 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 Symbolic reasoning with KLEE

We use the open-source KLEE symbolic executor, in particular, version `v3.1` which is the most up-to-date stable release of KLEE. You may want to read a bit of details on the project page about KLEE and symbolic execution in general.

KLEE is provisioned into the VM as a Docker image tagged as `klee`. 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 klee \
   bash
```

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

```
// Step 1: compile your code into LLVM IR
$ clang -emit-llvm -g -O0 -c main.c -o main.bc

// Step 2: symbolically explore all paths in the program
// NOTE: it caps the symbolic input from stdin to at max 1024 bytes
$ klee --libc=klee --posix-runtime --output-dir=output main.bc -sym-stdin 1024
```

The output of the KLEE execution results are stored in the `output` directory. The test cases generated by KLEE can be parsed by the ktest-tool.

1.4 Concolic execution with SymCC

We use the open-source SymCC concolic executor, in particular, commit 8978760 which is the most up-to-date stable commit of SymCC. You may want to read a bit of details on the project page about SymCC, concolic execution, and hybrid fuzzing in general.

SymCC is provisioned into the VM as a Docker image tagged as `symcc`. 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 symcc \
    bash
```

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

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

// Step 2: compile your code with symcc instrumentations
$ symcc main.c main.c -o main-sym

// Step 3: start fuzzing your code with naive afl in the background
$ screen -dmS afl -- \
    /afl/afl-fuzz -M afl-0 -i input -o output -- main-afl

// Step 4: wait a while for the fuzzer to initialize the directories
$ sleep 5

// Step 5: invoke the concolic executor to find more seeds
$ symcc_fuzzing_helper -v -o output -a afl-0 -n symcc -- main-sym
```

The output of the SymCC hybrid fuzzing results are stored in the `output` directory.

1.5 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
$ ./run-klee.sh pkg-sample
$ ./run-symcc.sh pkg-sample
```

The output should obviously show that 1) this package does not provide 100% code coverage and 2) it has a bug that is found by all three tools.

Despite that this package sample cannot "evade" any program analysis tool, feel free to duplicate this template package to bootstrap your package preparation that can eventually "evade" the tools.

1.6 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,

- to submit a package to the server, ZIP the package directory, send a HTTP `POST` request to `http://ugster72c.student.cs.uwaterloo.ca:8000/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://ugster72c.student.cs.uwaterloo.ca:8000/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., 3 analyzers x 15 minutes each). 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 3 Dropbox on LEARN.

Write-up. We use an auto-grader to check the code submitted for this assignment for both parts. 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 `README.md` 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).