Assignment 1– Fuzzer Playground (Part 1)

HSS
Fall 2022

In this part, you will be building a coverage-guided random input generator a.k.a. "fuzzer" for testing C programs.
Logistics:

- **LLVM Primer:** Please make sure that you have skimmed the LLVM Primer presentation (access it from the course webpage) to know the capabilities of LLVM.
- **Setup Repo:** I have created a `github` repo with all the necessary scripts to install LLVM, Z3 and starter code to write a pass. You can access it at: `https://github.com/HolisticSoftwareSecurity/hssllvmsetup`. The repo has examples of analysis (i.e., the passes that do not modify the IR) and instrumentation (i.e., the passes that modify the IR) passes.
- **Development Environment:** I use CLion (`https://www.jetbrains.com/clion/`) while working with LLVM and strongly suggest you to use it. You can get unlimited access using your `@purdue.edu` email.

In this lab, you will develop a fuzzer for testing C programs. Fuzzing is a popular software testing technique wherein the program under test is fed randomly generated inputs. Such inputs help uncover a wide range of security-critical and crashing bugs in programs. For this purpose, your fuzzer will begin with seed inputs, and generate new inputs by mutating previous inputs. It will use output from previous rounds of test as feedback to direct future test generation. You will use the code coverage metrics implemented in previous assignment to help select interesting inputs for your fuzzer to mutate.

Setup

The skeleton code for this part is located under `LLVMBasedFuzzer` folder of the following repo. We will frequently refer to this top level directory as `fuzzer` when describing file locations for the lab.

Repo

`https://github.com/HolisticSoftwareSecurity/FuzzerPlayground`

Step 1

Clone the above repository to a folder:

```
$ cd
$ git clone https://github.com/HolisticSoftwareSecurity/FuzzerPlayground.git
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 14 (delta 11), reused 11 (delta 8), pack-reused 0
Unpacking objects: 100% (14/14), done.
```

This lab builds off the work you did in llvm-playground assignment part 4. Start out by copying your solution from `Instrument.cpp` to `Fuzzer/src/Instrument.cpp`.

Then, run the following commands setup the lab:

```
$ cd ~/LLVMBasedFuzzer
$ mkdir build && cd build
$ cmake ..
$ make
```

Running make will build the `fuzzer` tool, which we will use to feed randomized input (that you will generate) into compiled C programs that will run with your sanitizer from instrumentation lab (i.e., llvm-playground assignment part 4). In particular, you should notice the `InstrumentPass.so` and fuzzer files in your `build` directory after running make.

Step 2

Next, create `Fuzzer/test/fuzz_output` if it does not exist. Then, prepare the programs under `Fuzzer/test` for fuzzing. This will be done by first instrumenting the `.c` files as you did in instrumentation lab. Here, we are instrumenting `sanity.c`:

```
$ cd ~/LLVMBasedFuzzer/Fuzzer/test
$ mkdir fuzz_output
$ clang -emit-llvm -S -fno-discard-value-names -c -o sanity.ll sanity.c -g
$ opt -load ../../build/Fuzzer/libInstrumentPass.so -Instrument -S sanity.ll > sanity.instrumented.ll
$ clang -o sanity -L${PWD}/../../build/Fuzzer -lruntime -lm sanity.instrumented.ll
```

Alternatively, you can run `make` in the `Fuzzer/test/` directory to instrument all `.c` files in the test directory. You can now run your fuzzer on the `sanity` executable:

```
$ timeout 1 ../build/fuzzer ./sanity fuzz_input fuzz_output
```

Notice the `./` before `sanity` as the fuzzer will not run without it. Since we can theoretically generate random input forever, we wrap the fuzzer in a `timeout` that will stop program execution after a specified interval. In the above case, we run for 1 second.

Expect `fuzz_output` to get populated with several files that contain the random inputs that cause crashes, e.g,

```
fuzz_output/failure/input1
fuzz_output/failure/input2
  ...
fuzz_output/failure/inputN
```

with N being the last case that caused a crash before the timeout. We will also output test cases that resulted in a successful program run under `fuzz_output/success/`; however, since it's likely you will get many successful program runs before a failure, we write every 1000th successful run.

You will also see the seed of the latest run saved in `fuzz_output/randomSeed.txt`. If you wish to provide your own seed, e.g. to reproduce a previous fuzzer run, you may do so by passing it as an additional argument to the above command:

```
$ timeout 1 ../build/fuzzer ./sanity fuzz_input fuzz_output 112358
```

For `sanity`, the fuzzer will find several crashes. However, to get your fuzzer to work on the other programs, you will need to complete `Fuzzer/src/Mutate.cpp`. The fuzzer you implement will randomly generate new inputs starting from the initial seed file that is located in `fuzz_input/seed.txt`. You will also use information about the program run, specifically, coverage information, to drive the fuzzing.

Lab Instructions

A full-fledged fuzzer consists of three key features: i) test case generation matching the grammar of the program input, ii) strategies to mutate test inputs to increase code coverage, iii) a feedback mechanism to help drive the types of mutations used.

Mutation-Fuzzing Primer. Consider the following code that reads some string input from the command line:

```
int main() {
  char input[65536];
  fgets(input, sizeof(input), stdin);
  int x = 13;
  int z = 21;

  if (strlen(input) % 13 == 0) {
    z = x / 0;
  }

  if (strlen(input) > 100 && input[25] == 'a') {
    z = x / 0;
  }

  return 0;
}
```

We have two very obvious cases that cause divide-by-zero errors in the program: (1) if the length of the program input is divisible by 13, and (2) if the length of the input is greater than 100 and the 25th character in the string is an 'a'. Now, let's imagine that this program is a black box, and we can only search for errors by running the code with different inputs.

We would likely try a random string, say "abcdef", which would give us a successful run. From there, we could take our first string as a starting point and add some new characters, "ghi", giving us "abcdefghi". Here we have mutated our original input string to generate a new test case. We might repeat this process, finally stumbling on "abcdefghijklm" which is divisible by 13 and causes the program to crash.

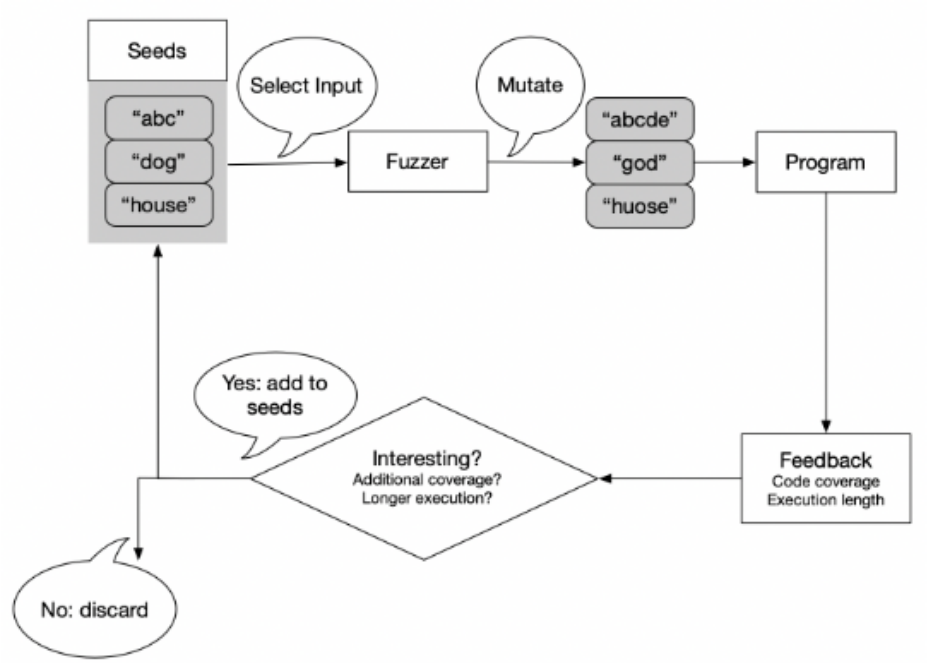
How about the second case? We could keep inserting characters onto the end of our string, which would eventually get us some large string that satisfies the first condition of the if statement (input length greater than 100), but we need to perform an additional type of mutation—randomly changing characters in the string—to eventually satisfy the second condition in the if statement.

Through various mutations on an input string, we ended up exhausting all program execution paths, i.e., more varied mutation in the input increased our code coverage. In its simplest form, this is exactly what a fuzzer does. You may take a look at the Mutation-Based Fuzzing chapter [1] in the Fuzzing Book.

[1] `https://www.fuzzingbook.org/html/MutationFuzzer.html`

Feedback-Directed Fuzzing. We've seen how randomized testing can find bugs and is a useful software analysis tool. The previous section describes a brute force generation of test cases; we simply perform random mutations hoping that we find a bug. This results in a lot of test cases being redundant, and therefore unnecessary.

We can gather additional information about a program's execution and use it as *feedback* to our fuzzer. The following figure shows at a high level what this process looks like:



Generating new, interesting seeds is the goal of feedback directed fuzzing. What does interesting mean? We might consider whether a test increases code coverage. If so, we have found new execution pathways that we want to continue to explore. Another test might significantly increase program runtime, in which case we might discover some latent performance bugs. In both cases, the tests increased our knowledge of the program; hence, we insert these tests into our set of seeds and use them as a starting point for future test generation.

Building the Fuzzer. In this lab, you will implement a mutation function mutate in `Fuzzer/src/Mutate.cpp`, which will perform some form of mutation on the input. You will decide which mutation strategy or strategies to use to perform the mutation. Feel free to explore the source code of the programs under `Fuzzer/test` like `easy.c` or `path.c` to get a better idea of how to implement the mutation function.

The fuzzer will start by mutating the seed values based on the mutation you've selected on the command line. The mutated value will be run on the input program, and feedback will be provided based on the coverage of that seed. You will then decide if this is an interesting seed and insert it as an additional seed if so in the SeedInput vector. This causes the mutated input to again be mutated. This process continues until the fuzzer gets interrupted (via timeout, or on the terminal by Ctrl+C). The following code snippet illustrates this main loop:

```
initialize(OutDir);

if (readSeedInputs(SeedInputDir)) {
  fprintf(stderr, "Cannot read seed input directory\n");
  return 1;
}
```

```
while (true) {
        std::string SC = selectInput();
        auto Mutant = mutate(SC);
        test(Target, Mutant, OutDir);
        feedBack(Target, Mutant);
}
```

Fuzzing the Binaries. We have provided several binary programs `hidden1`, `hidden2`, etc. These programs serve as more challenging test cases for your fuzzer. We will not provide the raw source; instead, your mutations should be versatile enough to find test cases that crash these programs.

Possible Mutations. The following is a list of potential suggestions for your mutations:

- Replace bytes with random values
- Swap adjacent bytes
- Cycle through all values for each byte
- Remove a random byte
- Insert a random byte

Feel free to play around with additional mutations, and see if you can speed up the search for bugs on the binaries. You may use the C++ function `rand()` to generate a random integer.

You will notice that different programs will require different strategies, or that in some cases you may even have to switch between different mutation strategies in the middle of the fuzzing process. You are expected to include a mechanism that will choose the best strategy for the input program based on the coverage feedback.

In short, the lab consists of the following tasks in `Mutate.cpp`:

1. Complete the `selectInput` function, which selects a mutant string from the `SeedInputs` vector.
2. Complete the `mutate` function so that it returns a `Mutant` of the original string. Take inspiration from the aforementioned list of mutations.
3. Decide whether the mutation was interesting based on success or failure of the program and the code coverage. Again, you may follow our groundwork and fill in `feedBack`.
4. Insert the `Mutant` into the pool of `SeedInput` to drive further mutation.

Code Coverage Metric. Recall that you have a way of checking how much of a particular program gets executed with the coverage information output by your sanitizer. A .cov file will get generated from your sanitizer in the working directory for the program that is getting fuzzed. For example, running:

```
$ ../../build/Fuzzer/fuzzer ./sanity fuzz_input fuzz_output
```

will create and update `sanity.cov` in the working directory that your `feedBack` function should read and use between each test.

A Few More Hints. Start small. Implement one mutation strategy in the `mutate` function to try and crash some of the easier test cases. Once successful, you can move on to implementing multiple strategies and choosing between them based on the feedback you get.

Do not be afraid to keep state between rounds of the fuzzing. You may want to try each of your mutation strategies initially to see which one generates a test that increases code coverage, and then exploit that strategy. We expect this fuzzer to be versatile; it should generate crashing tests on all programs in `Fuzzer/test`.

Submission

Submit file `Mutate.cpp`. Optionally, you may submit `Instrument.cpp`, and/or `randomSeed.txt` if you would like us to use a specific seed in grading the provided binary programs.