Fuzzing and gcov example

vulnerable source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void vulnerable_function(char *input) {
   char buffer[64];
   if (strlen(input) > 64) {
       printf("Buffer overflow detected!\n");
       return;
   }
   strcpy(buffer, input); // Vulnerable to buffer overflow
   printf("You entered: %s\n", buffer);
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
       printf("Usage: %s <input>\n", argv[0]);
       return 1;
   }
   vulnerable_function(argv[1]);
   return 0;
}
```

Compile using afl-gcc or afl-clang

```
afl-gcc -o vulnerable_program vulnerable_program.c
```

```
afl-clang-fast -o vulnerable_program vulnerable_program.c
```

input corpus

AFL++ takes a directory of seed inputs, known as the *input corpus*. This can be any old string, that the target program can accept as input. AFL will mutate this seed input during fuzzing to explore different execution paths.

```
 mkdir input_corpus
 echo "hello, world!" > input_corpus/input.txt
```

Run AFL++ against the vulnerable binary

run the following:

```
afl-fuzz -i input_corpus -o output_corpus ./vulnerable_binary @@
```

- `-i input_corpus/`: Specifies where to find seed inputs.
- `-o output_corpus/`: Directory where the results will be saved.

- `./vulnerable_binary @@`: The program to be fuzzed. @@ is replaced by the fuzzed input file.

Generate code coverage report

generate a coverage report by using `afl-showmap`. This shows the basic blocks covered by a particular input.

to analyze the coverage of a given input file:

```
afl-showmap -o coverage_output -- ./vulnerable_program < output_corpus/crashes/id:000001,sig
```

- `coverage_output`: Output for coverage information
- `output_corpus/crashes/...`: Example path to a crash file. (replace with a valid input generated through fuzzing.)

Use gcov for detailed code coverage reports.

For a more detailed code coverage report, you can compile the binary with coverage flags (-fprofile-arcs and -ftest-coverage) and use gcov to produce a human-readable report.

Recompile the binary with coverage instrumentation:

```
gcc -fprofile-arcs -ftest-coverage -o vulnerable_program_cov vulnerable_program.c
```

Run the fuzzed input against the instrumented binary:

```
./vulnerable_program_cov < output_corpus/crashes/id:000001,sig:11,src:000000+000001,op:havoc
```

Generate the coverage report using gcov:

```
gcov vulnerable_program.c
```

This will produce a vulnerable_program.c.gcov file, which contains a detailed line-by-line code coverage report. Each line of the source code will be annotated with the number of times it was executed.

Analyze fuzzing results

Step 7: Analyze Fuzzing Results

Crashes: AFL++ saves crash-inducing inputs in the output_corpus/crashes/ directory. Analyze these inputs to understand how the program crashes.

Coverage: Use afl-showmap and gcov to understand which parts of the binary were exercised during fuzzing.

Hangs: AFL++ also saves inputs that cause the program to hang in the output_corpus/hangs/ directory. These can indicate infinite loops or other issues.

Example: Summarizing Workflow

```
# Step 1: Compile with AFL++ instrumentation
afl-gcc -o vulnerable_program vulnerable_program.c

# Step 2: Create an initial seed input
mkdir input_corpus
echo "hello" > input_corpus/input.txt

# Step 3: Start fuzzing with AFL++
afl-fuzz -i input_corpus -o output_corpus ./vulnerable_program @@

# Step 4: After fuzzing, generate a coverage report for a crash input
afl-showmap -o coverage_output -- ./vulnerable_program < output_corpus/crashes/id:000001,sig

# Optional: Step 5: Use gcov for more detailed code coverage
gcc -fprofile-arcs -ftest-coverage -o vulnerable_program_cov vulnerable_program.c
./vulnerable_program_cov < output_corpus/crashes/id:000001,sig:11,src:000000+000001,op:havo
gcov vulnerable_program.c
```