Bug Finding

6.1600 Course Staff

Fall 2024

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

We are going to continue our treatment of security and bugs with a discussion of how to find bugs. As we already have seen, bugs are a big deal in terms of security problems. We have discussed how to architect a system using privilege separation so that bugs do not matter so much. But even with a very good privilege-separated design, we still want to make sure that our system is as bug-free as possible.

The topic of this chapter is then: *How do we find bugs?*

Step 1: Define what is a bug. Before we even begin talking about how to find bugs, we need to answer the question: *What is a bug?* There are a number of application-independent ways to determine when we have hit a bug:

- the program crashes,
- the program makes an out-of-bounds memory access, or
- the program jumps to an unknown or undefined point in the program.

More difficult types of bug to detect are ones that we can only detect with knowledge of what the application is supposed to do:

- the program's output is incorrect, or
- the program allows an attacker to access data it should not be able to access.

Step 2: Find bugs. To find a bug, we just need to identify a possible execution of the program that leads to one of the buggy outcomes we defined in Step 1. The reason this is difficult is:

- there are typically exponentially many possible inputs to the program and it may be difficult to find one the triggers the bug,
- concurrent systems exhibit non-deterministic behavior, and
- inputs from the environment (time, network state, etc.) can change the behavior of the program.

1 Bug finding: A concrete example

To frame our discussion of bugs, we will consider one specific example of a C program that parses a binary packet that has the format depicted in Fig. 1. The packet starts with a header byte and a length byte and then has a list of IDs.

```
-----------------------------------------
| header | len | id | id | id | ... | id |
-----------------------------------------
```

Figure 1: An example packet format.

```c
// Parse header
char in[64];
int hdr = in[0];
if (hdr != s) returnl;
int n = in[1];
if (n > 64) return;

// Read ID fields
int ctr[32];
char *next = &in[2];
for (int i=0; i < n; i++) {
  ctr[*next]++;
  next++;
}
```

Figure 2: An example of some buggy parsing code.

2 Manual testing

When we manually test code, we consider a *specific execution* of the program and predefine an *expected result*. For the example of Fig. 2, we might tests to check that the following two inputs give the following behavior:

```
in = {6};         // Should have no effect.
in = {5, 1, 2};   // Should cause ctr[2] == 1
```

Benefits. The main advantage of manual testing is that tests can be targeted and can exercise application-specific logic. If you have a precise correctness condition that your program should ensure, it is often easiest to test it with manual testing.

An additional advantage is that manual tests are useful in *regression testing*: If you find a bug in your program today, you can write a manual test that tests that the buggy condition does not occur. As you update your program later on, this regression test can determine whether the same bug occurs again.

Downsides. The downsides of manual tests are that they are expensive to write, the test cases can themselves be buggy (especially when the program is complicated), it requires a lot of program-specific understanding, and it is difficult to write enough tests to cover a large fraction of the program's behavior.

3 *Fuzzing*

Fuzzing is the process of running a program on a very large number of *randomly generated inputs*. As soon as a random input causes the program to crash, we know that we have detected a bug.

When using a fuzzer to test a piece of code, we will typically ask the compiler to instrument the code with extra instructions to test whether the code behaved improperly. In C, for example, we will instruct the compiler to check for out-of-bounds memory accesses.

With the GCC compiler, you can compile your code with the `-fsanitize=address` flag to insert extra checks for memory-access errors.

To test for application bugs with a fuzzer, we can instrument our code with assertions that will crash the program if the program ever violates certain programmer-specified invariants.

Fuzzers may have to test a large number of inputs before finding one that triggers a bug. In the code of Fig. 2, if a well-formed packet has `n=64`, the program will write off the end of the `in` array. (The check in Line 6 should test whether `n >= 64` instead of `n > 64`.) For a fuzzer to hit this bug, it will need to choose a random input value of the form:

```
in = {5, 64, ... 64 arbitrary values ...};
```

The probability that a uniform randomly bitstring of the appropriate length hits this bug is $2^{-8} \cdot 2^{-8} = 2^{-16}$, which is quite small.

Coverage-guided fuzzing. Real fuzzers do not just feed uniform random bitstrings to programs that they are trying to fuzz. Instead, real fuzzers try to pick inputs in a way that maximizes the fuzzer's *code coverage*: the fraction of the lines of the program's code that the fuzzed programs have executed.

Most random bitstrings will probably not cause the program to exercise a large fraction of the program's code, since the program will reject them early most of the time.

To implement this strategy, the fuzzer maintains a corpus of bitstrings. Each time the fuzzer runs the program, it picks an input from the corpus and randomly mutates it in some way (e.g., by changing or adding a byte). If running the new string on the program causes the program to execute some new lines of code (i.e., the coverage increases), the fuzzer adds the newly mutated string to the corpus.

While this strategy does not have a robust theory to support it, coverage-guided fuzzing works shockingly well in practice. Many major software projects use fuzzing extensively to find bugs.

A coverage-guided fuzzer run on Fig. 2 might find the following input that causes the program to crash with an out-of-bounds write:

```
in = {5, 1, 100};
```

Benefits. A major benefit of fuzzers is that they are almost completely automated—they require very little input from the programmer. Since programmer time is more expensive than machine time, finding bugs using fuzzers is often much cheaper than finding bugs via manual test cases. Since fuzzers execute the program on billions of inputs, they will often find tricky bugs that a human might never find.

Drawbacks. A drawback of fuzzers is that they cannot find application-specific bugs: they are essentially limited to only finding violated assertions in a program. So while fuzzers are a useful tool for finding bugs, they are typically only useful in conjunction with manual tests.

Generalizations of fuzzing. The first fuzzers used random bitstrings as their initial pool of inputs. More recent fuzzers have application-specific logic for handling HTML, JSON, or other file formats—these fuzzers are better at catching higher-level logic errors. Some languages, such as Go, have support for fuzzing in their test infrastructure.

4 *Symbolic execution*

A weakness of fuzzing is that a fuzzer may not be able to trigger bugs that hide behind `if` conditions that are very very rarely true. Symbolic execution is a testing strategy that can find bugs in these difficult-to-fuzz programs.

For example, if a parser first checks a CRC32 checksum on a packet, a fuzzer will almost never find an input that causes the program to run past the checksum check. Another example of difficult-to-fuzz code might be some HTML parsing code that checks that every < symbol is followed by an > symbol.

The idea of symbolic execution is that we will run the program. But instead of running the program on actual concrete input values, we will run the program on *symbolic variables* that represent arbitrary values. For example, we might want to use symbolic execution to run the following simple snippet of code:

```
c = a + b;
e = d + c;
```

```
f = a * d;
```

When executing this code, the state of memory might look like this, where we replace the value of the variable `d` with a variable X:

```
--------------------------------------------------
...    |  5  |  7  |  12  |  X  |     |     |  ...
--------------------------------------------------
          a     b      c     d     e     f
```

The major headache when using symbolic execution is *control flow*. For example, we might have code that looks like this:

```
if (e == f) {
  BUG();
} else {
  // Something else
}
```

If running with the symbolic variable `d` $= X$, then we will have the condition: `(d+c == a*d)`, which simplifies to: `(X+12 == 5*X)`.

To handle this sort of case, we can use a program called a *SAT Solver* to search for inputs that cause the condition to be true. For example, a SAT solver run on the branch condition `(X+12 == 5*X)` will likely find the value `X==3` that causes the condition to be true. Then the symbolic-execution engine can continue executing the program down the `true` branch of the program with the constraint `X==3` on the symbolic variable X. In parallel, the engine can search for values of `X` that cause the program to go down the `false` branch of the program. A SAT solver might find `X==908234` as one input that causes the program to traverse the `false` branch.

An important thing to know is that there is *no guarantee* that a SAT solver will actually be able to find an input that causes the program to execute one branch or another. The reason is that we know of no efficient algorithm for finding an assignment of the variables for an arbitrary condition that causes the condition to be true. (This problem is NP complete.) At the same time, for finding branch conditions in "reasonable" programs, SAT solves work surprisingly well.

Running a symbolic-execution engine on the code of Fig. 2 might produce the following output:

```
ERROR: buggy.c:10 memory error
ERROR: buggy.c:12 memory error
```

The symbolic-execution engine may consider the following tree of program executions, branching on each condition:

```
// Input data
in = {i0, i1, i2, i3, i4, ...};

[i0 == 5]
  /
  | FALSE -> return
  | TRUE  -> [i1 > 64]
  \           /
```

```
| TRUE  -> return
| FALSE -> [0 < i1]
\           /
            | FALSE -> return
            | TRUE  -> [i2 < 0 || i2 >= 32]
            \             /
                          | TRUE  -> BUG!
                          | FALSE -> [1 < i1]
                          \           /
                                      | ...continue ...
                                      | ...execution...
                                      \
```

Benefits. Symbolic execution can find tricky bugs involving complicated branch conditions that fuzzers and manual tests may not find. In addition, symbolic execution may be able to find bugs that do not crash the program. In addition, a symbolic-execution engine can in principle consider *all possible inputs* to a program and can give a guarantee that the program has no bugs of a certain type (e.g., out-of-bounds read).

Drawbacks. Symbolic-execution engines can be very slow to run and often work poorly on very large pieces of code. As the length of an execution grows, the symbolic-execution engine accumulates more and more symbolic variables (representing values in memory) and the number of constraints on each symbolic variables grows as well. When there are many variables and many constraints, the SAT solver may not be able to determine—in a reasonable amount of time—whether there is or is not a satisfying assignment to the variables. For these reasons, symbolic execution can work well for small snippets of code; in large programs (such as a web browser), symbolic execution may not be able to progress very deep into the program.

To address these drawbacks in symbolic executions, one approach is to write a *scheduler* that guides the search that the symbolic execution makes through the program state. A second approach is to define *loop invariants* or *function invariants* that aim simplify the job that the SAT-solver must perform by giving it more information about the program's expected behavior.