Fuzzing

Fuzzing Techniques

What is a fuzzing?

- A kind of random testing
    - **Goal**: make sure certain **bad things don’t happen,** **no matter what**
    - **Crashes, thrown exceptions, non-termination**
- **Complements functional testing**
    - Test features (and lack of misfeatures) directly
    - Normal tests can be starting points for fuzz tests

Kinds of fuzzing

- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky
- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space
- **White box**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed
  - Often **easy to use**, but **computationally expensive**

Fuzzing inputs

- **Mutation**
  - Take a **legal input and mutate it**, using that as input
  - Legal input may be human-produced e.g., from a grammar or SMT solver query (e.g., from a grammar or SMT solver query)
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
  - Generate initial input, mutate n times, generate new inputs, ...
  - Generate mutations according to grammar

Dealing with Crashes

- You **fuzz**. A **crash** occurs. **Questions**:
- What is the **root cause** (so it can be fixed)**?**
    - Is there a way to **make the input smaller**, so it is more understandable?
    - Are **two or more crashes signaling the same bug?!**
        - Yes, if they “minimize” to the same input
- Does the crash signal an **exploitable vulnerability**?
    - Dereferencing NULL is rarely exploitable
    - Buffer overruns often are positioned in a different place than the crash

Finding memory errors

- **Compile** the program with **Address Sanitizer** (ASAN)
  - Instrument accesses to arrays to check for overflows and use after free errors: https://code.google.com/p/address-sanitizer/
- **Fuzz it!**
- Did the program **crash with an ASAN-signaled error**? Then worry about exploitability
- Similarly, you can compile with other sorts of error checkers for the purposes of testing E.g., valgrind memcheck http://valgrind.org/

Fuzzing vs. Symbolic Execution

- (Dynamic) symbolic execution is sometimes called whitebox fuzzing, see, e.g., SAGE.
  - Let consider a program with an input x, having the following condition: `if (x > 5 && x < 10) abort();`
  - Suppose x is an integer, i.e. its domain is 2^{32}. The probability to randomly generate an input x that triggers the error is $4/2^{32}$. This probability is a bit higher in clever greybox fuzzing, but in general they suck for this kind of condition.
  - On the other hand, whitebox fuzzing executes the program with a symbol x = X, and use a constraint solver, e.g. Z3, to solve the constraint. Z3 can solve $X > 5 \land X < 10$ in mili or nano second.

Fuzzing vs. Symbolic Execution

- However, this power analysis comes with an expensive overhead.
  - A recent paper shows that for just one path, i.e. no constraint solving, an execution in KLEE is 3000 times slower than native execution, while angr is 300,000 times slower. Since it is very slow, it is unrealistic to achieve 100% coverage.
  - Constraint solver is only good for linear arithmetic. There is no efficient solver for the theory of string, floating point arithmetic
  - It is unrealistic to know all information about APIs, environment etc.

Hybrid Fuzzing

- The current state-of-the-art is **hybrid fuzzing**, combining both greybox and whitebox. The idea is to use greybox fuzzer as *global search* to quickly sample the state space. When it gets stuck, then use the heavyweight whitebox as *local search.*
- This is the technique used by all the team in the Cyber Grand Challenge. The hybrid fuzzer QSym , running on a modest machine, discovered some CVE that Google couldn't find with greybox fuzzers running on the cloud.
- *hybrid fuzzing*, was recently proposed. It combines both fuzzing and concolic execution, with the hope that the fuzzer will quickly explore trivial input spaces (i.e., loose conditions) and the concolic execution will solve the complex branches (i.e., tight conditions)