Automatic Tools for Finding Bugs

Dawn Song

dawnsong@cs.berkeley.edu

Important to Develop Techniques to Discover Bugs/Vulnerabilities in Programs

- **Programs tend to have bugs**
- **Ideally, prove programs correct/secure**
  - **E.g., using pre/post condition & invariants as discussed in earlier lecture**
  - **However, automated proofs hard to scale to large programs**
- **One alternative, find as many bugs as we can**
- **Key question: how to find bugs in programs?**

Approach I: Black-box Fuzz Testing

- **Given a program, simply feed it random inputs, see whether it crashes**
- **Advantage: really easy**
- **Disadvantage: inefficient**
  - **Input often requires structures, random inputs are likely to be malformed**
  - **Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low**

Enhancement: With Protocol/Format Info

- **Mutation-based fuzzing:**
  - Take a well-formed input, randomly perturb (flipping bit, etc.)
  - E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/
    - » Try out your own tool there
- **Generation-based fuzzing**
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity http://www.immunitysec.com/resources-freesoftware.shtml
- **Shortcomings:**
  - Still hard to find the rare cases that would trigger the bug

Approach II: Constraint-based Automatic Test Case Generation

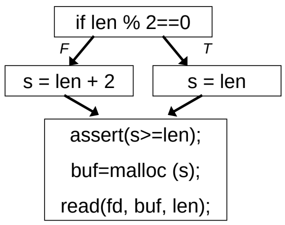
- **Look inside the box**
  - Use the code itself to guide the fuzzing
- **Assert security/safety properties**
- **Explore different program execution paths to check for security properties**
- **Challenge:**
  1. For a given path, need to check whether an input can trigger the bug, i.e., violate security property
  2. Find inputs that will go down different program execution paths

Running Example

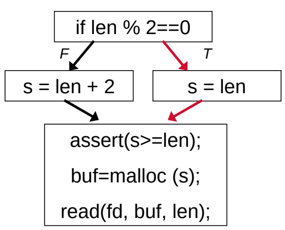
```
f(unsigned int len){
 unsigned int s;
 char *buf;
  if len % 2==0;
  then s = len;
  else s = len + 2;
 buf = malloc(s);
 read(fd, buf, len);
  …
}
```

- Where's the bug?
- What's the security/safety property?
  - s>=len
- What inputs will cause violation of the security property?
  - len = 2^{32} - 1
- How likely will random testing find the bug?

Running Example

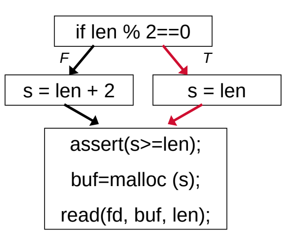


Symbolic Execution



- **Test input len=6**
- **No assertion failure**
- **What about all inputs that takes the same path as len=6?**

Symbolic Execution

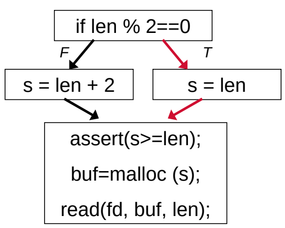


- **What about all inputs that takes the same path as len=6?**
- **Represent len as symbolic variable**

Symbolic Execution

- Reprenset inputs as symbolic variables
- Perform each operation on symbolic variables symbolically
  - x = y + 5;
- Registers and memory values dependent on inputs become symoblic expressions
- Certain conditions for conditional jump become symbolic expressions as well

Symbolic Execution



len % 2 = 0 (path constraint)

s = len

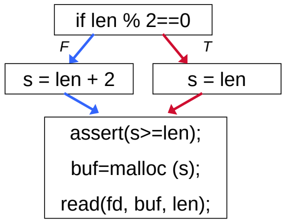
s<len?

- What about all inputs that takes the same path as len=6?
- Represent len as symbolic variable

Using a Solver

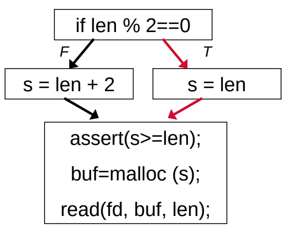
- Is there a value for len s.t.
  len % 2 = 0 ^ s = len ^ s < len?
- Give the symbolic formula to a solver
- In this case, the solver returns No
  - The formula is not satisfiable
- What does this mean?
  - For any len that follows the same path as len = 6, the execution will be safe
  - Symbolic execution can check many inputs at the same time for the same path
- What to do next?
  - Try to explore different path

How to Explore Different Paths?



- Previous path constraint: len % 2 = 0
- Flip the branch to go down a different path:
  - len % 2 != 0
- Using a solver for the formula
  - A satisfying assignment is a new input to go down the path

Checking Assertion in the Other Path



len % 2 != 0 (path constraint)

s = len + 2

s<len?

- **Is there a value for len s.t. len % 2 != 0 ^ s = len+2 ^ s < len?**
- **Give the symbolic formula to a solver**
  - **Solver returns satisfying assignment: len = 2^{32} -1**
  - **Found the bug!**

Summary: Symbolic Execution for Bug Finding

- **Symbolicly execution a path**
  - **Create the formula representing: path constraint ^ assertion failure**
  - **Give the solver the formula**
    - » If returns a satisfying assignment, a bug found
- **Reverse condition for a branch to go down a different path**
  - **Give the solver the new path constraint**
  - **If returns a satisfying assignment**
    - » The path is feasible
    - » Found a new input going down a different path
- **Pioneer work**
  - **EXE, DART**

Challenges

- **Too many paths to explore**
  - **Exponential or infinite # of paths**
- **How to address the challenge?**
  - **Prioritize for block/branch coverage**

Other Applicatoins to Symbolic Execution

- **Automatic signature generation**
- **Automatic patch-based exploit generation**

Administrivia

- **HW4 due today**
- **Project milestone #2 due Wed**

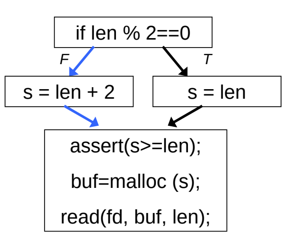
Other Applicatoins to Symbolic Execution

- **Automatic signature generation**
- **Automatic patch-based exploit generation**

Symbolic Execution for Signature Generation

- **Instead of bit patterns, use root cause**
  - **Generating signatures based on vulnerability**
- **As exploits morph, they need to trigger vulnerability**
- **So, vulnerability puts constraints on exploits**
- **Problem reduction:**
  - **Signature generation = constraints on inputs that trigger vulnerability**
- **Symbolic execution**

Identifying the Constraints



len % 2 != 0 (path constraint)

s = len + 2

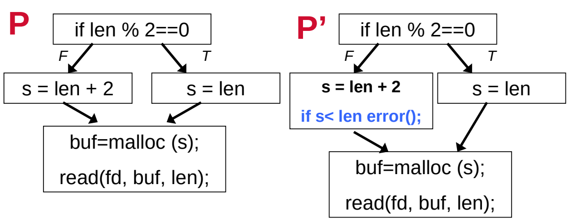
s<len?

- **Given exploit len = 2^{32} -1**
- **Constraint on len to trigger vulnerability: len % 2 != 0 ^ s = len+2 ^ s < len**
- **Use this constraint as the signature**

Signature Quality

- **False positive?**
  - **No**
- **False negative?**
  - **Depending on path coverage**
- **Challenge**
  - **Increase path coverage**

Automatic Patch-based Exploit Generation



- **Patch leaks**
  - **Vulnerability point (where in code)**
  - **Vulnerability condition (under what conditions)**
- **Exploits for P are inputs that fail vulnerability condition at vulnerability point**
  - **len % 2 != 0 ^ s = len+2 ^ s < len**

Procedure Summary

1. **Diff P and P' to identify candidate vuln point and condition**
2. **Create input that satisfy candidate vuln condition in P'**
   - **i.e., candidate exploits**
3. **Check candidate exploits on P**

Real-world Examples

- 5 Microsoft patches
  - Mostly 2007
  - Integer overflow, buffer overflow, information disclosure, DoS
- Automatically generated exploits for all 5 patches
  - In seconds to minutes
  - 3 out of 5 have no publicly available exploits
  - Automatically generated exploit variants for the other 2

Conclusion

- Automatic testing for bug finding
  - Symbolic execution
    - » check all inputs along the same path at the same time
    - » Automatically finding new inputs to go down different paths
- Other applications for symbolic execution
  - Automatic signature generation
  - Automatic patch-based exploit generation