EXE: Automatically Generating Inputs of Death

Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, Dawson Engler

Computer Systems Laboratory -- Stanford University

Lisa Johansen
February 13, 2006
Code Bugs

• Do you code safe?
  ‣ use assertions
  ‣ catch errors
  ‣ check inputs

• Are you human?

• Code is, by nature, buggy
  ‣ size, human error, C

• How do we solve this?
Execution generated Executions

- Goal: dynamically (and quickly) find vulnerabilities in code
- Challenge: enumerating every bug possible in code that could enable attacks or errors
- Technique: find all of the locations for possible bugs, enumerate all possible execution paths over those locations, test the code with these paths
- What does this tell us?
How it works?

• Compile code with EXE
  ‣ finds symbolic operands, determines constraints, inserts code to fork at branches, inserts tests for correctness

• Execute EXE
  ‣ “constraint solver” checks for satisfaction of constraints in execution
    ‣ finds bugs and the inputs necessary to exploit them

• Execute
  ‣ run the code with the given input and watch it fail
Symbolic Operands

• Symbolic operands allow EXE to determine all of the values that a given operand may be.
  ‣ These operands must be specifically set

• EXE has separate memory stores for concrete and symbolic. After execution, the concrete store will have a solution (if one exists)
What’s in a test?

• Universal checks:
  ‣ divide or mod by zero
  ‣ dereferenced null pointer
  ‣ validity of a dereferenced pointer

• Generalized universal checks:
  ‣ asserts
  ‣ inverse functions
  ‣ equivalent functions
CVCL and STP

• CVCL and STP are both constraint solvers
  ‣ CVCL - cooperating validity checker lite
  ‣ STP - (for a dollar)

• Given the EXE running code and the constraints determined at compilation time, is there a solution that fits the constraints?

• C code is translated into bits for simple (global) execution
Example

```c
int main(void) {
    unsigned i, t, a[4] = { 1, 0, 5, 2 };
    make symbolic(&i);

    t = a[i];

    t = t / a[i];

    return a[a[i]];  
}
```
Optimizations

• Array optimizations in STP
• Constraint caching (results from satisfiability queries)
• Constraint independence optimization (making the queries simpler)
• Search heuristics
  ‣ DFS by default is not always the best option
Testing and Results

• Packet Filtering
  ‣ buffer overflows in FreeBSD BPF

• Complete Server
  ‣ read overflows in udchpd

• Device Driver
  ‣ Nothing yet, but were able to run it on a variety of drivers (including a joystick driver which is just funny)

• File System Implementations
  ‣ found that malicious code can cause buffer overflows, kernel panic, or perform reads and writes to random memory locations (ext2, ext3, JFS)
Testing and Results

• Oh yeah, the performance of the optimizations in STP over CVCL was ridiculous
  ‣ 20 times faster
  ‣ code is smaller (and what have we learned about small code)
Authors’ Issues

• What if you call uninstrumented code
• What if the STP does not terminate (or not?)
• The tests should be configurable
So what?

• Dynamic
• Performed at the bit level
• Related work
  ‣ Random inputs
  ‣ Complete symbolic representation
  ‣ Static analysis
  ‣ Exclusion of many of the tests
Implications

• “We had to add only 10 lines of code to all the device drivers to get them to run in our system.”

• “…and restructured one loop that interacted badly with our current system.” (server test)

• You still have to fix it and, unfortunately, you are still human