CMPSC 497: Dynamic Analysis

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Our Goal

• In this course, we want to develop techniques to detect vulnerabilities automatically before they are exploited
  ‣ What’s a vulnerability?
  ‣ How to find them?
Vulnerability

• How do you define computer ‘vulnerability’?
  ‣ **Flaw**
  ‣ **Accessible to adversary**
  ‣ **Adversary has ability to exploit**
Vulnerability

• How do you define computer ‘vulnerability’?
  ‣ Flaw – Can we find flaws in source code?
  ‣ Accessible to adversary – Can we find what is accessible?
  ‣ Adversary has ability to exploit – Can we find how to exploit?
One Approach

• Run the program on various inputs
  ‣ See what happens
  ‣ Maybe you will find a flaw

• How should you choose inputs?
Dynamic Analysis Options

• Regression Testing
  ‣ Run program on many normal inputs and look for bad behavior in the responses
  • Typically looking for behavior that differs from expected – e.g., a previous version of the program

• Fuzz Testing
  ‣ Run program on many abnormal inputs and look for bad behavior in the responses
  • Looking for behaviors that may be triggered by adversaries
    ‣ Bad behaviors are typically crashes caused by memory errors
Dynamic Analysis Options

• Why do you think fuzz testing is more appropriate for finding vulnerabilities?
Fuzz Testing

• Fuzz Testing
  ‣ Idea proposed by Bart Miller at Wisconsin in 1988

• Problem: People assumed that utility programs could correctly process any input values
  ‣ Available to all

• Found that they could crash 25-33% of UNIX utility programs
Fuzz Testing

- **Fuzz Testing**
  - Idea proposed by Bart Miller at Wisconsin in 1988

- **Approach**
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate with the random inputs that caused the crashes

- **Problems**: Not checking returns, Array indices…
Example Found

• Fuzz Testing
  ‣ Idea proposed by Bart Miller at Wisconsin in 1988

format.c (line 276):
... 
while (lastc != '\n') {
  rdc();
}
... 

input.c (line 27):
rdc()
{ do { readchar(); } 
  while (lastc == ' ' || lastc == '\t'); return (lastc);
}
Challenges

• **Idea**: Search for possibly accessible and exploitable flaws in a program by running the program under a variety of inputs

• **Challenge**: Selecting input values for the program
  ‣ What should be the goals in choosing input values for dynamic analysis?
Challenges

- **Idea**: Search for possibility exploitable flaws in a program by running the program under a variety of inputs.

- **Challenge**: Selecting input values for the program
  - What should be the goals in choosing input values for dynamic analysis?
  - *Find all exploitable flaws*
  - *With the fewest possible input values*

- How should these goals impact input choices?
Black Box Fuzzing

• Like Miller – Feed the program random inputs and see if it crashes

• **Pros**: Easy to configure

• **Cons**: May not search efficiently
  ‣ May re-run the same path over again (low coverage)
  ‣ May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
  ‣ May cause the program to terminate for logical reasons – fail format checks and stop
Black Box Fuzzing

• Example

```c
function( char *name, char *passwd, char *buf )
{
    if ( authenticate_user( name, passwd ) ) {
        if ( check_format( buf ) ) {
            update( buf );
        }
    }
}
```
Mutation-Based Fuzzing

• Supply a well-formed input
  ‣ Generate random changes to that input

• No assumptions about input
  ‣ Only assumes that variants of well-formed input may problematic

• Example: zzuf
  ‣ Reading: The Fuzzing Project Tutorial
Mutation-Based Fuzzing

• Example: zzuf
  ‣ http://sam.zoy.org/zzuf/

• The Fuzzing Project Tutorial
  ‣ zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe
  ‣ Fuzzes the program objdump using the sample input win9x.exe
  ‣ Try 1M seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)
Mutation-Based Fuzzing

• Easy to setup, and not dependent on program details
• But may be strongly biased by the initial input
• Still prone to some problems
  ‣ May re-run the same path over again (same test)
  ‣ May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
Generation-Based Fuzzing

• Generational fuzzer generate inputs “from scratch” rather than using an initial input and mutating

• However, to overcome problems of naïve fuzzers they often need a format or protocol spec to start

• Examples include
  ‣ SPIKE, Peach Fuzz

• However format-aware fuzzing is cumbersome, because you'll need a fuzzer specification for every input format you are fuzzing
Generation-Based Fuzzing

• Can be more accurate, but at a cost

• **Pros**: More complete search
  ‣ Values more specific to the program operation
  ‣ Can account for dependencies between inputs

• **Cons**: More work
  ‣ Get the specification
  ‣ Write the generator – ad hoc

• Need to do for each program
Grey Box Fuzzing

• Rather than treating the program as a black box, instrument the program to track the paths run

• Save inputs that lead to new paths
  ‣ Associated with the paths they exercise

• Example
  ‣ American Fuzzy Lop (AFL)

• “State of the practice” at this time
AFL

• Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
AFL

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- See
  - http://lcamtuf.coredump.cx/afl/
AFL Build

• Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

• Replace the gcc compiler in your build process with afl-gcc

• For example, in the Makefile (for Project 1)
  ‣ CC=path-to/afl-gcc

• Then build your target program with afl-gcc
  ‣ Generates a binary instrumented for AFL fuzzing
AFL Use

• Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

• Run the fuzzer using afl-fuzz

  path-to/afl-fuzz –i <input-dir> –o <output-dir> <path-to-bin> [args]

• For example

  path-to/afl-fuzz –i input/ –o output/ ./cmpsc497-p1 set user passwd @@

• Where

  ‣ input/input-x.txt is the input file supplied in project 1
  ‣ output is the directory where the AFL results will be placed
AFL Use

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- Run the fuzzer using afl-fuzz
  
  ```
  path-to/afl-fuzz -i <input-dir> -o <output-dir> <path-to-bin> [args]
  ```

- For example
  
  ```
  path-to/afl-fuzz -i input/ -o output/ ./cmpsc497-p1 set user passwd @@
  ```

- Where
  - @@ shows that the last arg (input file) will be fuzzed
  - Can also do “user” and “passwd”
AFL Issues

• Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

• After you install AFL but before you can use it effectively, you must set the following environment variables to prevent aborts

  setenv AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES

  setenv AFL_SKIP_CPUFREQ

• The former speeds up response from crashes

• The latter suppresses AFL complaint about missing some short-lived processes
AFL Display

• Tracks the execution of the fuzzer

- Key information are
  - “total paths” – number of different execution paths tried
  - “unique crashes” – number of unique crash locations
AFL Output

• Shows the results of the fuzzer
  ‣ E.g., provides inputs that will cause the crash
• File “fuzzer_stats” provides summary of stats – UI
• File “plot_data” shows the progress of fuzzer
• Directory “queue” shows inputs that led to paths
• Directory “crashes” contains input that caused crash
• Directory “hangs” contains input that caused hang
AFL Results

• Shows the results of the fuzzer
  ‣ E.g., provides inputs that will cause the crash

• Crashes
  ‣ May be caused by failed assertions – as they abort
    • Had several assertions caught as crashes, but format violated my checks
  ‣ I had a bug in authenticate_user – not zeroing hash_preimage data before writing
    • Slowed down the fuzzer – fixed this and the fuzzer generated unique paths more quickly
AFL Operation

• How does AFL work?
  ‣ [http://lcamtuf.coredump.cx/afl/technical_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt)

• The instrumentation captures branch (edge) coverage, along with coarse branch-taken hit counts.
  ‣ `cur_location = <COMPILE_TIME_RANDOM>;
  ‣ `shared_mem[cur_location ^ prev_location]++;
  ‣ `prev_location = cur_location >> 1;

• Record branches taken with low collision rate

• Enables distinguishing unique paths
AFL Operation

• How does AFL work?
  ‣ [http://lcamtuf.coredump.cx/afl/technical_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt)

• When a mutated input produces an execution trace containing new tuples, the corresponding input file is preserved and routed for additional processing
  ‣ Otherwise, input is discarded

• Mutated test cases that produced new state transitions are added to the input queue and used as a starting point for future rounds of fuzzing
AFL Operation

- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

- Fuzzing strategies
  - Highly deterministic at first – bit flips, add/sub integer values, and choose interesting integer values
  - Then, non-deterministic choices – insertions, deletions, and combinations of test cases
Grey Box Fuzzing

• Finds flaws, but still does not understand the program

• **Pros**: Much better than black box testing
  ‣ Essentially no configuration
  ‣ Lots of crashes have been identified

• **Cons**: Still a bit of a stab in the dark
  ‣ May not be able to execute some paths
  ‣ Searches for inputs independently from the program

• Need to improve the effectiveness further
White Box Fuzzing

• Combines test generation with fuzzing
  ‣ Test generation based on static analysis and/or symbolic execution – more later
  ‣ Rather than generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
    • And use them as fuzzing inputs
  
• **Goal**: Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage
White Box Fuzzing

- We will come back to white box testing when we have the tools to perform automated test generation
Take Away

• Goal is to detect vulnerabilities in our programs before adversaries exploit them

• One approach is dynamic testing of the program
  ‣ Fuzz testing aims to achieve good program coverage with little effort for the programmer
  ‣ Challenge is to generate the right inputs

• Black box (Mutational and generation), Grey box, and White box approaches are being investigated
  ‣ AFL (Grey box) is now commonly used