Introduction to Software Testing

Lecture 14

Fuzz Testing

Instructor: Morteza Zakeri

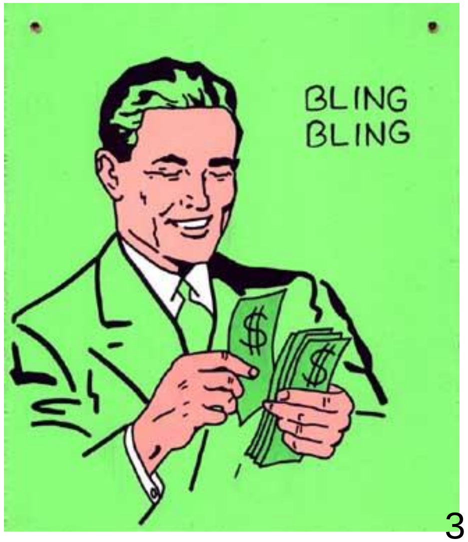
Some Slides by: **Tal Garfinkel, Charlie Miller, *et al*.**

March 2024

Fuzzing: (Semi)Automated Methods for Security Bug Detection

Vulnerability Finding Today

- **Security bugs (Vulnerabilities)** can bring $500-$100,000 on the open market
- Good bug finders make $180-$250/hr consulting
- Few companies can find **good people**, many do not even realize this is possible.
- Still largely a black art



Security Vulnerabilities

- What can **Security bugs** **(Vulnerabilities)** an attacker do?
  - avoid authentication
  - privilege escalation
  - bypass security check
  - deny service (crash/hose configuration)
  - run code remotely

Why not eliminate bugs all together?

- Impractical in general
  - Formal verification is hard in general, impossible for big things.
- Why do not you just program in Java, Haskell, <your favorite safe language>
  - Does not eliminate all problems
  - Performance, existing code base, flexibility, programmer competence, etc.
- Not cost effective
  - Only really need to catch same bugs as bad guys
- Incremental solutions beget incremental solutions
  - Better bug finding tools and mitigations make radical but complete solutions less economical

Bug Patterns

- Most bugs fit into just a few classes
  - See Mike Howards "19 Deadly Sins"
  - Some lend themselves to automatic detection, others don't
- Which classes varies primarily by language and application domain.
  - (C and C++) - Memory safety: Buffer overflows/ integer overflow/ double free()/ format strings.
  - Web Apps - Cross-Site Scripting (XSS), SQL Injection, etc.

More Bug Patterns

- Programmers repeat bugs
  - Copy/paste
  - Confusion over API
    - e.g., linux kernel drivers, Vista exploit, unsafe string functions
  - Individuals repeat bugs
- Bugs come from broken assumptions
  - Trusted inputs become untrusted
- Others bugs are often yours
  - Open source, third party code

Bug Finding **Arsenal**

- Threat Modeling: Look at design, write out/diagram what could go wrong.
- Manual code auditing
  - Code reviews
- Automated Tools

- Techniques are complementary
  - Few turn key solutions, no silver bullets

What this talk is about

- Using tools to find bugs
  - Major techniques
  - Some tips on how to use them
- Static Analysis
  - **Compile time**/ source code level
  - Compare code with abstract model
- Dynamic Analysis
  - Run Program/Feed it inputs/See what happens

Static Analysis

Two Types of Static Analysis

- The type you write in 100 lines of python.
  - Look for known unsafe string functions `strncpy()`, `sprintf()`, `gets()`
  - Look for unsafe functions in your source base
  - Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)
- The type you get a PhD for
  - Buy this from coverity, fortify, etc.
  - Built into visual studio
  - Roll your own on top of LLVM or Pheonix if your hardcore

Static Analysis Basics

- Model program properties abstractly, look for problems
- Tools come from program analysis
  - Type inference, data flow analysis, theorem proving
- Usually on source code, can be on byte code or disassembly
- Strengths
  - Complete code coverage (in theory)
  - Potentially verify absence/report all instances of whole class of bugs
  - Catches different bugs than dynamic analysis
- Weaknesses
  - High false positive rates
  - Many properties cannot be easily modeled
  - Difficult to build
  - Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)

Example: Where is the bug?

```
int read_packet(int fd)
{
    char header[50];
    char body[100];
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, bound_b);
    read(fd, body, bound_b);

    return 0;
 }
```

Example: Where is the bug?

```
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100);
    read(fd, body, 100);

    return 0;
 }
```

Example: Where is the bug?

```
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

    read(fd, header, 100); //constant propagation
    read(fd, body, 100);   //constant propagation

    return 0;
 }
```

Example: Where is the bug?

```
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

   //check read(fd, dest.size >= len)
    read(fd, header, 100); //constant propagation
    read(fd, body, 100);   //constant propagation

    return 0;
 }
```

Example: Where is the bug?

```
int read_packet(int fd)
{
    char header[50]; //model (header, 50)
    char body[100];  //model (body, 100)
    size_t bound_a = 50;
    size_t bound_b = 100;

   //check read(fd, 50 >= 100) // SIZE MISMATCH!!
    read(fd, header, 100); //constant propagation
    read(fd, body, 100);   //constant propagation

    return 0;
 }
```

Rarely are Things This Clean

- Need information across functions
- Ambiguity due to pointers
- Lack of association between size and data type…
- Lack of information about program inputs/runtime state…

Rarely are Things This Clean

- Need information across functions
- Ambiguity due to pointers
- Lack of association between size and data type…
- Lack of information about program inputs/runtime state…

- Static Analysis is not a **panacea (نوش دارو)**, still its very helpful especially when used properly.

Care and Feeding of Static Analysis Tools

- Run and Fix Errors Early and Often
  - otherwise false positives can be overwhelming.
- Use Annotations
  - Will catch more bugs with few false positives e.g. SAL
- Write custom rules!
  - Static analysis tools provide institutional memory
- Take advantage of what your **compiler** provides
  - `gcc -Wall, /analyze   //in visual studio`
- Bake it into your build or source control

Dynamic Analysis

Normal Dynamic Analysis

- Run program in **instrumented** execution environment
  - Binary translator, Static instrumentation, emulator
- Look for bad stuff
  - Use of invalid memory, race conditions, null pointer deref, etc.
- Examples: Purify, Valgrind, Normal OS exception handlers (crashes)
- Most well-known vulnerability testing: **Fuzz testing** or **Fuzzing**

Regression *vs.* Fuzzing

- Regression: Run program on many normal inputs, look for badness.
  - Goal: Prevent normal users from encountering errors (e.g. assertions bad).

- Fuzzing: Run program on many **abnormal (Negative) inputs**, look for badness.
  - Goal: Prevent attackers from encountering exploitable errors (e.g., assertions often ok)

Fuzzing Basics

- **Automatically** generate test data
- Many slightly **anomalous test data** are input into a target interface
- Application is **monitored** for errors
- Inputs are generally either **file based** (.pdf, .png, .wav, .mpg)
- Or network based…
  - http, SNMP, SOAP
- Or other…
  - Command line apps
  - e.g., `crashme()`

Trivial Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- **Anomalous** requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1

Different Ways To Generate Inputs

- Mutation Based - “Dumb Fuzzing”
- Generation Based - “Smart Fuzzing”

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)

- Examples:
  - Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.

Example: fuzzing a pdf viewer

- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script to)

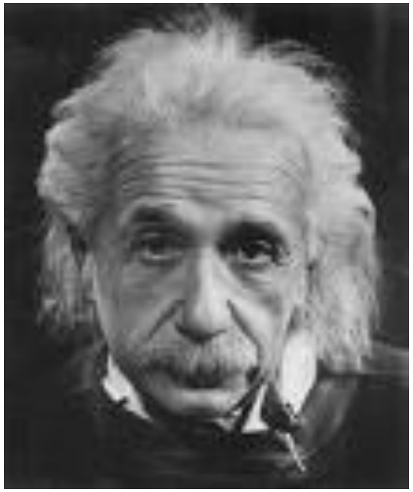
1. Grab a file
2. Mutate that file
3. Feed it to the program
4. Record if it crashed (and input that crashed it)

Dumb Fuzzing In Short

- Strengths
  - Super easy to setup and automate
  - Little to no protocol knowledge required
- Weaknesses
  - Limited by initial corpus
  - May fail for protocols with checksums, those which depend on challenge response, etc.

Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Example: Protocol Description

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation Based Fuzzing In Short

- Strengths
  - completeness
  - Can deal with complex dependencies e.g. checksums
- Weaknesses
  - Have to have spec of protocol
    - Often can find good tools for existing protocols e.g. http, SNMP
  - Writing generator can be labor intensive for complex protocols
  - The spec is not the code

Fuzzing Tools

Input Generation

- Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  - Mu-4000, Codenomicon, PROTOS, FTPFuzz
- Fuzzing Frameworks: You provide a spec, they provide a fuzz set
  - SPIKE, Peach, Sulley
- Dumb Fuzzing automated: you provide the files or packet traces, they provide the fuzz sets
  - Filep, Taof, GPF, ProxyFuzz, PeachShark
- Many special purpose fuzzers already exist as well
  - ActiveX (AxMan), regular expressions, etc.

Input Inject

- Simplest
  - Run program on fuzzed file
  - Replay fuzzed packet trace
- Modify existing program/ client
  - Invoke fuzzer at appropriate point
- Use fuzzing framework
  - e.g., **Peach** automates generating COM interface fuzzers

Problem Detection

- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (**valgrind, ASAN,** purify)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own checker, e.g., valgrind skins

Workflow Automation

- Sulley, Peach, Mu-4000 all provide tools to aid setup, running, recording, etc.
- Virtual machines can help create reproducible workload
- Some assembly still required

How Much Fuzz Is Enough?

- Mutation based fuzzers can generate an **infinite number** of test cases...
  - When has the fuzzer run long enough?
- Generation based fuzzers generate a **finite number** of test cases.
  - What happens when they're all run and no bugs are found?

Example: PDF

- I have a PDF file with 248,000 bytes
- There is one byte that, if changed to particular values, causes a crash
  - This byte is 94% of the way through the file
- Any single random mutation to the file has a probability of .00000392 of finding the crash
- On average, need 127,512 test cases to find it
- At 2 seconds a test case, thats just under 3 days...
- It could take a week or more...

Code Coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of **profiling tools.** e.g., `gcov`

Types of Code Coverage

- Line coverage
  - Measures how many lines of source code have been executed.
- Branch coverage
  - Measures how many branches in code have been taken (conditional jmps)
- Path coverage
  - Measures how many paths have been taken

Example

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- Requires
  - 1 test case for line coverage
  - 2 test cases for branch coverage
  - 4 test cases for path coverage
    - i.e. `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`

Problems with Code Coverage

- Code can be covered without revealing bugs
- Error checking code mostly missed (and we don't particularly care about it)

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

- Only "attack surface" reachable
  - **i.e.,** the code processing user controlled data
  - No easy way to measure the attack surface
    - Interesting use of static analysis?

```
ptr = malloc(sizeof(blah));
if(!ptr)
     ran_out_of_memory();
```

Code Coverage Good For Lots of Things

- How good is this initial file?
- Am I getting stuck somewhere?

```
if(packet[0x10] < 7) { //hot path
} else { //cold path

}
```

- How good is fuzzer X vs. fuzzer Y
- Am I getting benefits from running a different fuzzer?

See Charlie Miller's work for more!

Fuzzing Rules of Thumb

- Protocol specific knowledge very helpful
  - Generational tends to beat random, better spec's make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you find
- Best results come from guiding the process
  - Notice where your getting stuck, use profiling!
- Code coverage can be very useful for guiding the process

The Future of Fuzz

Outstanding Problems

- What if we don't have a spec for our protocol/How can we avoid writing a spec.
- How do we select which possible test cases to generate

Whitebox Fuzzing

- Infer protocol spec from observing program execution,then do generational fuzzing
- Potentially best of both worlds
- Bleeding edge

How do we generate constraints?

- Observe running program
  - – Instrument source code (EXE)
  - – Binary Translation (SAGE, Catchconv)
- Treat inputs as symbolic
- Infer contraints

Example:

```
int test(x)
{
   if (x < 10) {
                     //X < 10 and X <= 0 gets us this path
         if (x > 0) {
                     //0 < X < 10 gets us this path
          return 1;
      }
   }
                  //X >= 10 gets us this path
  return 0;
}
```

- Constraints:

X >= 10

0 < X < 10

X <= 0

Solve Constraints -- we get test cases: {12,0,4}

- Provides maximal code coverage

Greybox Techniques

- **Evolutionary Fuzzing**
- Guided mutations based on fitness metrics
- Prefer mutations that give
  - Better code coverage
  - Modify inputs to potentially dangerous functions (e.g. memcpy)
- EFS, autodafe

Summary

- To find bugs, use the tools and tactics of an attacker
- Fuzzing and static analysis belong in every developers toolbox
- Field is rapidly evolving
- If you do not apply these tools to your code, someone else will ☺