Software Security: Vulnerability Analysis
Finding Bugs/Vulnerabilities

• Attackers:
  – Find vulnerabilities
  – Weaponize them (exploit the vulnerabilities)
  – Use exploits to compromise machines & systems
  – Exploits are worth money
Market for 0days

- Sell for $10K-1M
Finding Bugs/Vulnerabilities

• Defenders:
  – Find vulnerabilities & eliminate them
    • Improve security of software
    • Easier and cheaper to fix a vulnerability before software deployed
    • After deployed: patching is expensive
  – Ideally prove a program is free of vulnerabilities
Example: Static Device Verifier

- Verifies that drivers are not making illegal function calls or causing system corruption
  - SLAM project at Microsoft

- “The requirements for the Windows logo program (now Windows Hardware Certification Program) state that a driver must not fail while running under Driver Verifier.”
Techniques & Approaches

Automatic test case generation
- Fuzzing
  - Lower coverage
  - Lower false positive
  - Higher false negative

Static analysis
- Dynamic Symbolic Execution

Program verification
- Higher coverage
- Lower false negative
- Higher false positive
Fuzzing
Finding bugs in PDF viewer

PDF viewer

?
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
Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, .mpg)
- Or network based...
  - http, SNMP, SOAP
## Regression vs. Fuzzing

<table>
<thead>
<tr>
<th></th>
<th>Regression</th>
<th>Fuzzing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td>Run program on many <strong>normal</strong> inputs, look for badness.</td>
<td>Run program on many <strong>abnormal</strong> inputs, look for badness.</td>
</tr>
<tr>
<td><strong>Goals</strong></td>
<td>Prevent <strong>normal users</strong> from encountering errors (e.g. assertion failures are bad).</td>
<td>Prevent <strong>attackers</strong> from encountering <strong>exploitable</strong> errors (e.g. assertion failures are often ok).</td>
</tr>
</tbody>
</table>
Enhancement I: Mutation-Based Fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- 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:
  - E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/
  - 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)
  1. Grab a file
  2. Mutate that file
  3. Feed it to the program
  4. Record if it crashed (and input that crashed it)
## Mutation-based Fuzzing In Short

<table>
<thead>
<tr>
<th>Mutation-based</th>
<th>Super easy to setup and automate</th>
<th>Little to no protocol knowledge required</th>
<th>Limited by initial corpus</th>
<th>May fail for protocols with checksums, those which depend on challenge</th>
</tr>
</thead>
</table>
Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
    [http://www.immunitysec.com/resources-freesoftware.shtml](http://www.immunitysec.com/resources-freesoftware.shtml)

- Anomalies are added to each possible spot in the inputs

- Knowledge of protocol should give better results than random fuzzing

![Diagram](Dawn Song)
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=ONEBYTE
    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");
...

Dawn Song
# Generation-Based Fuzzing In Short

<table>
<thead>
<tr>
<th>Method</th>
<th>Setup and Automation</th>
<th>Knowledge Required</th>
<th>Initial Corpus</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation-based</td>
<td>Super easy to setup and automate</td>
<td>Little to no protocol knowledge required</td>
<td>Limited by initial corpus</td>
<td>May fail for protocols with checksums, those which depend on challenge</td>
</tr>
<tr>
<td>Generation-based</td>
<td>Writing generator can be labor intensive for complex protocols</td>
<td>Have to have spec of protocol (Often can find good tools for existing protocols e.g. http, SNMP)</td>
<td>Completeness</td>
<td>Can deal with complex dependencies e.g. checksums</td>
</tr>
</tbody>
</table>

Dawn Song
Fuzzing Tools & Frameworks

Input generation → Input injection → Bug detection
Input Generation

• Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  – Mu Dynamics, Codenomicon, PROTOS, FTPFuzz, WebScarab
• Fuzzing Frameworks: providing a fuzz set with a given spec
  – SPIKE, Peach, Sulley
• Mutation-based fuzzers
  – Taof, GPF, ProxyFuzz, PeachShark
• Special purpose fuzzers
  – ActiveX (AxMan), regular expressions, etc.
Input Injection

• 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
Bug Detection

• See if program crashed
  – Type of crash can tell a lot (SEGV vs. assert fail)
• Run program under dynamic memory error detector (valgrind/purify)
  – Catch more bugs, but more expensive per run.
• See if program locks up
• Write your own checker: e.g. valgrind skins
Workflow Automation

• Sulley, Peach, Mu-4000
  – Provide tools to aid setup, running, recording, etc.
• Virtual machines: help create reproducible workload
How Much Fuzzing Is Enough?

• Mutation based fuzzers may generate an infinite number of test cases... When has the fuzzer run long enough?

• Generation based fuzzers may generate a finite number of test cases. What happens when they’re all run and no bugs are found?
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
Line Coverage

Line/block coverage: Measures how many lines of source code have been executed.

For the code on the right, how many test cases (values of pair \((a,b)\)) needed for full(100%) line coverage?

```plaintext
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```
Branch Coverage

Branch coverage: Measures how many branches in code have been taken (conditional jmps)

For the code on the right, how many test cases needed for full branch coverage?

if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
Path Coverage

Path coverage: Measures how many paths have been taken.

For the code on the right, how many test cases needed for full path coverage?

```cpp
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```
Code Coverage

• Benefits:
  – 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?
Problems of code coverage

• For:  
  ```c
  mySafeCpy(char *dst, char* src){
    if(dst && src)
      strcpy(dst, src);
  }
  ```

• Does full line coverage guarantee finding the bug?
  ○ Yes  ○ No
Problems of code coverage

• For:  

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

• Does full line coverage guarantee finding the bug?
  ○ Yes  ○ No

• Does full branch coverage guarantee finding the bug?
  ○ Yes  ○ No
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 may 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: AFL
- Can we do better?
Symbolic Execution
(Next Lecture)