Lecture 13 – Finding Vulnerabilities

Stephen Checkoway
University of Illinois at Chicago
CS 487 – Fall 2017
Slides based on Bailey’s ECE 422
Testing

• Testing Overview
• Automated White Box Tools
• Fuzzing
• Reverse Engineering
The Need for Specifications

• Testing checks whether program implementation agrees with program specification
• Without a specification, there is nothing to test!
• Testing is a form of consistency checking between implementation and specification
  • Recurring theme for software quality checking approaches
  • What if both implementation and specification are wrong?
Developer != Tester

• Developer writes implementation, tester writes specification
• Unlikely that both will independently make the same mistake
• Specifications useful even if written by developer itself
  • Much simpler than implementation
  • Specification unlikely to have same mistake as implementation
Classification of Testing Approaches

![Diagram showing a classification of testing approaches with axes labeled Manual vs. Automated and Black-Box vs. White-Box. The diagram is a 2x2 grid with no labels in the quadrants.]
Automated vs. Manual Testing

• Automated Testing:
  • Find bugs more quickly
  • No need to write tests
  • If software changes, no need to maintain tests
• Manual Testing:
  • Efficient test suite
  • Potentially better coverage
Black-Box vs. White-Box Testing

• Black-Box Testing:
  • Can work with code that cannot be modified
  • Does not need to analyze or study code
  • Code can be in any format (managed, binary, obfuscated)

• White-Box Testing:
  • Efficient test suite
  • Potentially better coverage
How Good Is Your Test Suite?

• How do we know that our test suite is good?
  • Too few tests: may miss bugs
  • Too many tests: costly to run, bloat and redundancy, harder to maintain

• Example: SQLite
  “As of version 3.20.0 (2017-08-01), the SQLite library consists of approximately **125.4 KSLOC of C code**. (KSLOC means thousands of ‘Source Lines Of Code’ or, in other words, lines of code excluding blank lines and comments.) By comparison, the project has 730 times as much test code and test scripts - **91616.0 KSLOC**.”
Code Coverage

• Metric to quantify extent to which a program’s code is tested by a given test suite
  • Function coverage: which functions were called?
  • Statement coverage: which statements were executed?
  • Branch coverage: which branches were taken?

• Given as percentage of some aspect of the program executed in the tests

• 100% coverage rare in practice: e.g., inaccessible code
  • Often required for safety-critical applications
  • Example: SQLite has 100% branch coverage
Classification of Testing Approaches

- Manual
- Automated
- Black-Box
- White-Box

[Diagram showing the classification of testing approaches with a checkmark in the White-Box category]
Manual white-box testing

• Tests written by hand
• Full knowledge of source code/deployment/infrastructure
• Can test all parts
• Test *running* can be automated (e.g., on commits/deployment)
Test Driven Security
Classification of Testing Approaches
Automated white-box testing

• Tests created automatically/dynamically

• Godefroid et al. “Automated Whitebox Fuzz Testing”
  • Record trace of program on well-formed inputs
  • Symbolic execution to capture constraints on input
  • Negate a constraint, use a constraint solver to derive new input, run on that input

• American fuzzy lop
  • Compile-time instrumentation
  • Genetic algorithms guided by the instrumentation

• Tools exist
Automated white-box testing tools

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td>Cycles Done: 0</td>
</tr>
<tr>
<td>Last New Path</td>
<td>Total Paths: 195</td>
</tr>
<tr>
<td>Last Unique Crash</td>
<td>Uniq Crashes: 0</td>
</tr>
<tr>
<td>Last Unique Hang</td>
<td>Uniq Hangs: 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now Processing</td>
<td>Map Density: 1217 (7.43%)</td>
</tr>
<tr>
<td>Paths Timed Out</td>
<td>Count Coverage: 2.55 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now Trying</td>
<td>Favored Paths: 128 (65.64%)</td>
</tr>
<tr>
<td>Stage Execs</td>
<td>New Edges On: 85 (43.59%)</td>
</tr>
<tr>
<td>Exec Speed</td>
<td>Total Crashes: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Flips: 88/14.4k, 6/14.4k</td>
<td>Levels: 3</td>
</tr>
<tr>
<td>Byte Flips: 0/1804, 0/1786, 1/1750</td>
<td>Pending: 178</td>
</tr>
<tr>
<td>Arithmetic: 31/126k, 3/45.6k, 1/17.8k</td>
<td>Pending Fav: 114</td>
</tr>
<tr>
<td>Known Ints: 1/15.8k, 4/65.8k, 6/78.2k</td>
<td>Imported: 0</td>
</tr>
<tr>
<td>Havoc: 34/254k, 0/0</td>
<td>Variable: 0</td>
</tr>
<tr>
<td>Trim: 2876 B/931 (61.45% gain)</td>
<td>Latent: 0</td>
</tr>
</tbody>
</table>
Classification of Testing Approaches

- Manual - Black-Box
- Automated - White-Box

- Manual - Black-Box
- Automated - White-Box
Manual black-box testing

• Tester interacts with the system in a black-box fashion
• Crafts ill-formed inputs, tests them, and records how the system reacts
Web Pen Testing Simple Example
Classification of Testing Approaches
Automated black-box testing

- Fuzzing components
  - Test case generation
  - Application execution
  - Exception detection and logging
Test Case Generation

• Random Fuzzing

• “Dumb” (mutation-based) Fuzzing
  • Mutate an existing input

• “Smart” (generation-based) Fuzzing
  • Generate an input based on a model (grammar)
Mutation Fuzzer

- Charlie Miller’s “5 lines of Python” fuzzer
- Found bugs in PDF and PowerPoint readers

```python
numwrites=random.randrange(
    math.ceil((float(len(buf)) / FuzzFactor)))+1
for j in range(numwrites):
    rbyte = random.randrange(256)
    rn = random.randrange(len(buf))
    buf[rn] = "%c"%(rbyte);
```
Classification of Testing Approaches

A diagram illustrates the classification of testing approaches into manual and automated categories, with black-box and white-box testing. The diagram points towards an increase in automated testing.
Reverse Engineering

• Reverse Engineering (RE) -- process of discovering the technological principles of a [insert noun] through analysis of its structure, function, and operation.

• The development cycle ... backwards
Why Reverse Engineer?

• Malware analysis
• Vulnerability or exploit research
• Check for copyright/patent violations
• Interoperability (e.g. understanding a file/protocol format)
• Copy protection removal
Legality

• Gray Area (a common theme)
• Usually breaches the EULA contract of software
• Additionally -- DMCA law governs reversing in U.S.
  • “may circumvent a technological measure ... solely for the purpose of enabling interoperability of an independently created computer program”
Two Techniques

• Static Code Analysis (structure)
  • Disassemblers

• Dynamic Code Analysis (operation)
  • Tracing / Hooking
  • Debuggers

• Combination of the two works best in my experience
Disassembly

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Bits: 11101011 00000110
Hex Bytes: 0xEB 0x06
Instructions: JMP +6

Bits: 01010000
Hex Bytes: 0x50
Instructions: PUSH EAX
Difficulties

• Imperfect disassembly

• Benign Optimizations
  • Constant folding
  • Dead code elimination
  • Inline expansion
  • etc...

• Intentional Obfuscation
  • Packing
  • No-op instructions
Packing

• “Tons” of malware

Cumulative Distribution of Hits per MD5
246,952 unique MD5s, 5,772,891 Hits

<table>
<thead>
<tr>
<th>PEID</th>
<th>Count</th>
<th>SigBuster</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX</td>
<td>11244</td>
<td>Allaple</td>
<td>22050</td>
</tr>
<tr>
<td>Upack</td>
<td>6079</td>
<td>UPX</td>
<td>11324</td>
</tr>
<tr>
<td>PECompact</td>
<td>4872</td>
<td>PECompact</td>
<td>5276</td>
</tr>
<tr>
<td>Nullsoft</td>
<td>2295</td>
<td>FSG</td>
<td>5080</td>
</tr>
<tr>
<td>Themida</td>
<td>1688</td>
<td>Upack</td>
<td>3639</td>
</tr>
<tr>
<td>FSG</td>
<td>1633</td>
<td>Themida</td>
<td>1679</td>
</tr>
<tr>
<td>IElock</td>
<td>1398</td>
<td>FSG</td>
<td>5080</td>
</tr>
<tr>
<td>NsPack</td>
<td>1375</td>
<td>NsPack</td>
<td>1645</td>
</tr>
<tr>
<td>ASpack</td>
<td>1283</td>
<td>ASpack</td>
<td>1505</td>
</tr>
<tr>
<td>WinUpack</td>
<td>1234</td>
<td>IElock</td>
<td>1332</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nullsoft</td>
<td>1058</td>
</tr>
</tbody>
</table>

Identified: 59,070 (60%) Identified: 69,974 (71%)
Top 10: 33.3% Top 10: 55.3%
How about the unidentified?

- Unidentified have: high entropy, small IATs
- Overall: > 90% packed
Dynamic Analysis

• A couple techniques available:
  • Tracing / Hooking
  • Debugging
Tracing with Procmon

Kernel supported API
Event Tracing for Windows (ETW)
Debugger Features

• Trace every instruction a program executes -- single step
• Or, let program execute normally until an exception
• At every step or exception, can observe / modify:
  • Instructions, stack, heap, and register set
• May inject exceptions at arbitrary code locations
• INT 3 instruction generates a breakpoint exception
Debugging Benefits

• Sometimes easier to just see what code does
• Unpacking
  • just let the code unpack itself and debug as normal
• Most debuggers have in-built disassemblers anyway
• Can always combine static and dynamic analysis
Difficulties

• We are now executing potentially malicious code
  • use an isolated virtual machine

• Anti-Debugging
  • detect debugger and [exit | crash | modify behavior ]
  • IsDebuggerPresent(), INT3 scanning, timing, VM-detection, pop ss trick, etc., etc., etc.
  • Anti-Anti-Debugging can be tedious
Commonality of evasion

• Detect evidence of monitoring systems
  • Fingerprint a machine/look for fingerprints

• Hide real malicious intents if necessary

  • IF VM_PRESENT() or DEBUGGER_PRESENT()
    • Terminate()  // hide real intents
  • ELSE
    • Malicious_Behavior()  // real intents
## Taxonomy of malware evasion

<table>
<thead>
<tr>
<th>Layer of abstraction</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Installation, execution</td>
</tr>
<tr>
<td>Hardware</td>
<td>Device name, drivers</td>
</tr>
<tr>
<td>Environment</td>
<td>Memory and execution artifacts</td>
</tr>
<tr>
<td>Behavior</td>
<td>Timing</td>
</tr>
</tbody>
</table>
Example 1

- Device driver strings
  - Network cards
Example 2

- VMWare CommChannel (hooks)

  Under VMware
  
  Write Magic values to EAX, EBX...
  
  Read Port ‘VX’
  
  Useful information returned
  
  VMware detected

  VMware Detection
  
  Write Magic values to EAX, EBX...
  
  Read Port ‘VX’
  
  No exception
  
  VMware Not detected

  Under Plain Machine
  
  Write Magic values to EAX, EBX...
  
  Read Port ‘VX’
  
  Exception raised
  
  VMware Not detected
VMware detection code

MOV    EAX, 0x564D5868 ; 'VMXh'
MOV    EBX, 0        ; Any value but not the MAGIC VALUE
MOV    ECX, 0x0A     ; Get VMWare version
MOV    EDX, 0x5658   ; 'VX' (port number)
IN     EAX, DX      ; Read port
CMP    EBX, 0x564D5868 ; Is there a reply from VMWare? 'VMXh'

https://www.aldeid.com/wiki/VMXh-Magic-Value
Prevalence of evasion

- **40%** of malware samples exhibit fewer malicious events with debugger attached
- **4.0%** exhibit fewer malicious events under VMware execution