Lecture 35 – Finding Vulnerabilities

Ryan Cunningham
University of Illinois
ECE 422/CS 461 – Fall 2017
CS436 – Systems and Networking Lab

CRN: 67921 (CS); 67922 (ECE)

Course description: This course teaches an understanding of networks and systems design through hands-on construction and experimentation with real-world implementations, scenarios, and devices. Students will perform bi-weekly projects in building, analyzing, evaluating, and deploying the communication protocols and server software behind modern cloud/compute/network infrastructures. Students will gain hands-on implementation experience in operating system networking kernels, cloud application service code, and firewall and router configuration. Students will gain experience with widely-used and production-grade code and systems, such as Cisco IOS, the Linux networking stack, and Amazon Web Services. This class links theory with practice to prepare students to confidently carry out tasks they will commonly encounter in industry, such as building an enterprise network, deploying a large-scale cloud service, or implementing a new network protocol. This course builds upon computer networking courses such as CS 241 and CS 438 to cover practical and experimental aspects of networking.

Prerequisite: CS 241 (Systems Programming), or equivalent course on operating systems or networking.

When: MW 09:30am - 10:50am

Where: Siebel Center 1109

Course website: http://web.engr.illinois.edu/~caesar/courses/cs436.s18/

Contact information: caesar@illinois.edu
Security News

- MacOS High Sierra login bug
- Many HP printers found to have RCE vuln
- exim SMTP server has RCE vuln
- Krebs might have identified contractor responsible for Shadow Brokers leaks
TESTING
The Need for Specifications

• Testing checks whether program implementation agrees with program specification
• Without a specification, there is nothing to test!
• Testing 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 developers themselves
  - Much simpler than implementation
  - Specification unlikely to have same mistake as implementation
Classification of Testing Approaches

Manual | Automated
--- | ---
Black-Box | White-Box
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
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
Classification of Testing Approaches
Test Driven Security
Classification of Testing Approaches

- Black-Box (Manual, Automated)
- White-Box (Manual, Automated)

- Manual
- Automated
Automated White Box Testing
Classification of Testing Approaches

Manual vs. Automated

Black-Box  White-Box
Web Pen Testing Simple Example
Classification of Testing Approaches

<table>
<thead>
<tr>
<th>Manual</th>
<th>Automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Box</td>
<td>White-Box</td>
</tr>
</tbody>
</table>
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.randint(0, math.ceil(len(buf) / FuzzFactor)) + 1
for j in range(numwrites):
    rbyte = random.randint(0, 255)
    rn = random.randint(0, len(buf))
    buf[rn] = '%c' % (rbyte);
```
Classification of Testing Approaches

[Diagram showing a 2x2 matrix with axes labeled 'Manual' and 'Automated', and categories 'Black-Box' and 'White-Box'. An arrow points from 'Manual Black-Box' to 'Automated White-Box'.]
Reverse Engineering

- Reverse Engineering (RC), Reverse Code Engineering (RCE)
- reverse engineering -- 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
• IT’S FUN!
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
Disassembly

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

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg/Opcode</th>
<th>R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

11101011 00000110

0xEB 0x06

JMP +6

Bits

01010000

0x50

PUSH EAX

Hex Bytes

Instructions (human-readable)
Control Flow Diagram

Basic Block
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

Identified: 59,070 (60%)
Top 10: 33.3%

Packer identification
98,801 malware samples

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

- Unidentified have: high entropy, small IATs
- Overall: > 90% packed
Anti-disassembly

- Attackers use clever tricks to confuse the disassembler
JNZ XOR JZ CALL DEC ADD

75 52 33 C0 74 01 E8 8B 45 0C 8B 48 04 0F

.text:0040100A jnz short loc_40105E
.text:0040100C xor eax, eax
.text:0040100E jz short near ptr loc_401010+1
.text:00401010
.text:00401010 loc_401010:
.text:00401010 call near ptr 8B4C55A0h
.text:00401015 dec eax
.text:00401016 add al, 0Fh
JNZ XOR JZ MOV MOV MOVSX

75 52 33 C0 74 01 E8 8B 45 0C 8B 48 04 0F

.text:0040100A jnz short loc_40105E
.text:0040100C xor eax, eax
.text:0040100E jz short loc_401011
.text:00401010 db 0E8h
.text:00401011 mov eax, [ebp+0Ch]
.text:00401014 mov ecx, [eax+4]
.text:00401017 movsx edx, byte ptr [ecx]
Rogue False Branch

• Possible to create two contradictory disassembly interpretations of a binary.
• Disassembler takes false branch first (linear sweep).
• Make false branch bogus with useless conditional.
  – Use back-to-back jump instructions (e.g., JZ and JNZ) to always jump to a location.
  – Jump to a location with a constant condition (e.g., XOR eax,eax followed by JZ).
ADD  MOV  JMP
C4 08  C6 00 00  EB FF C0  48 E8 F1

.text:0040120F    add esp, 8
.text:00401212    mov byte ptr [eax], 0
.text:00401215    jmp short near ptr loc_401215+1
.text:00401217    db 0C0h ; +
.text:00401218    db 48h ; H
.text:00401219    db 0E8h ; F
.text:0040121A    db 0F1h ; ±
ADD       MOV       INC       DEC       CALL
C4 08  C6 00 00  EB  FF  C0  48  E8  F1

.text:0040120F add esp, 8
.text:00401212 mov byte ptr [eax], 0
.text:00401215 db 0EBh
.text:00401216 inc eax
.text:00401218 dec eax
.text:00401219 call sub_40130F

NOT VALID DISASSEMBLY!
Obfuscating Control Flow

- Recursive descent disassembler cannot disassemble instructions it cannot find.
- Manipulate function pointers:
  lea eax,[ebp+14]; add eax,14; call [eax];
- Manipulate return instructions:
  call $+5; add [esp],5; retn;
- Manipulate structured exception handlers (SEH):
  push EH; push fs:[0]; mov fs:[0], esp; xor ecx,ecx; div ecx;
Thwarting Stack Analysis

• IDA identifies stack variables by checking ESP math.
• Easy to throw off that analysis by abusing, or not following, calling conventions (e.g., compute variable offsets using ESP and strange math).
• That will absolutely wreck decompilers.
Dynamic Analysis

• A couple techniques available:
  1. Tracing / Hooking
  2. Debugging
**Tracing with Procmon**

<table>
<thead>
<tr>
<th>Time</th>
<th>Process Name</th>
<th>PID</th>
<th>Operation</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:46:00</td>
<td>calc.exe</td>
<td>5400</td>
<td>Process Start</td>
<td>C:\WINDOWS\system32\calc.exe</td>
</tr>
<tr>
<td>12:46:01</td>
<td>calc.exe</td>
<td>5400</td>
<td>Thread Create</td>
<td></td>
</tr>
<tr>
<td>12:46:02</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryNameInfo</td>
<td>C:\WINDOWS\system32\calc.exe</td>
</tr>
<tr>
<td>12:46:03</td>
<td>calc.exe</td>
<td>5400</td>
<td>Load Image</td>
<td>C:\WINDOWS\system32\ntdll.dll</td>
</tr>
<tr>
<td>12:46:04</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryNameInfo</td>
<td>C:\WINDOWS\system32\calc.exe</td>
</tr>
<tr>
<td>12:46:05</td>
<td>calc.exe</td>
<td>5400</td>
<td>Load Image</td>
<td>C:\WINDOWS\system32\ntdll.dll</td>
</tr>
<tr>
<td>12:46:06</td>
<td>calc.exe</td>
<td>5400</td>
<td>CreateFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:07</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryStandardInfo</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:08</td>
<td>calc.exe</td>
<td>5400</td>
<td>ReadFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:09</td>
<td>calc.exe</td>
<td>5400</td>
<td>CloseFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:10</td>
<td>calc.exe</td>
<td>5400</td>
<td>CreateFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:11</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryInformation</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:12</td>
<td>calc.exe</td>
<td>5400</td>
<td>FileSystemService</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:13</td>
<td>calc.exe</td>
<td>5400</td>
<td>CreateFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:14</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:15</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:16</td>
<td>calc.exe</td>
<td>5400</td>
<td>CloseFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:17</td>
<td>calc.exe</td>
<td>5400</td>
<td>CreateFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:18</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:19</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:20</td>
<td>calc.exe</td>
<td>5400</td>
<td>CloseFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:21</td>
<td>calc.exe</td>
<td>5400</td>
<td>CreateFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:22</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:23</td>
<td>calc.exe</td>
<td>5400</td>
<td>QueryDirectory</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
<tr>
<td>12:46:24</td>
<td>calc.exe</td>
<td>5400</td>
<td>CloseFile</td>
<td>C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl</td>
</tr>
</tbody>
</table>

**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 intent 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>

The table compares different layers of abstraction in malware evasion, from application to behavior, with examples for each layer. The layers are ordered from easier (Application) to harder (Behavior).
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
    - Exception raised

Under Plain Machine:
- Write Magic values to EAX, EBX...
  - Read Port ‘VX’
    - Exception raised
    - VMware Not detected
Prevalence of evasion

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