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

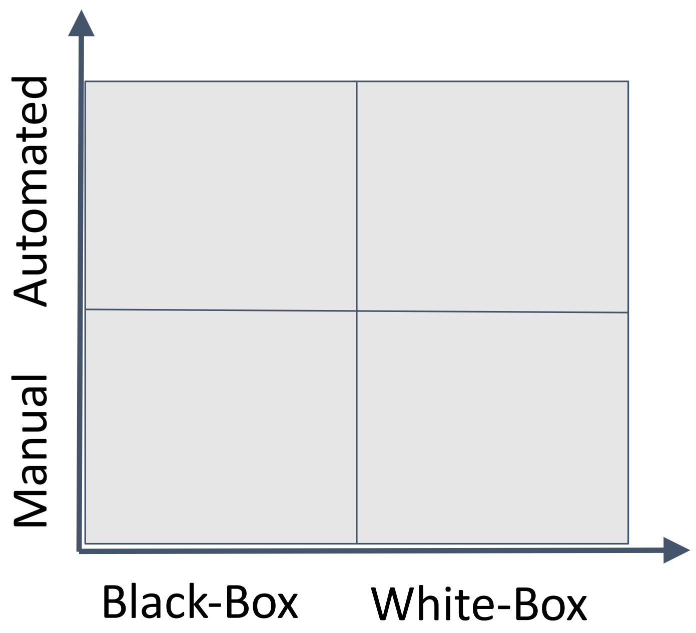
The Need for Specifications

- 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



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

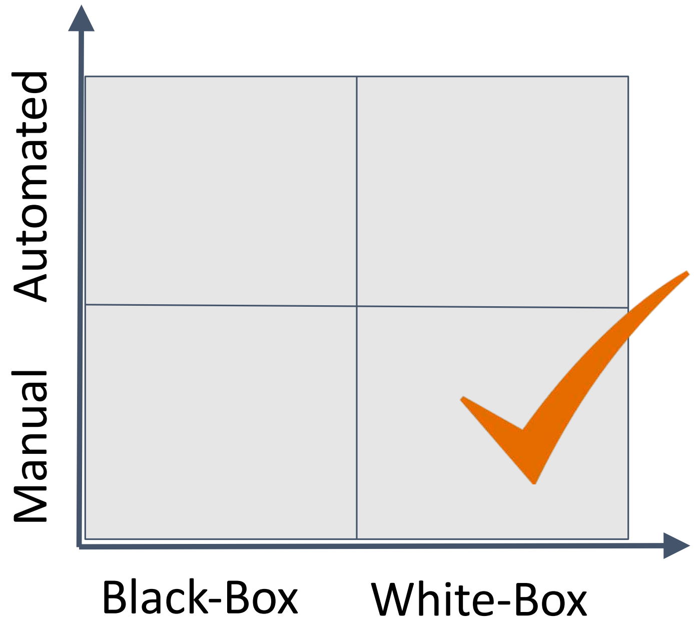
How Good Is Your Test Suite?

- 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

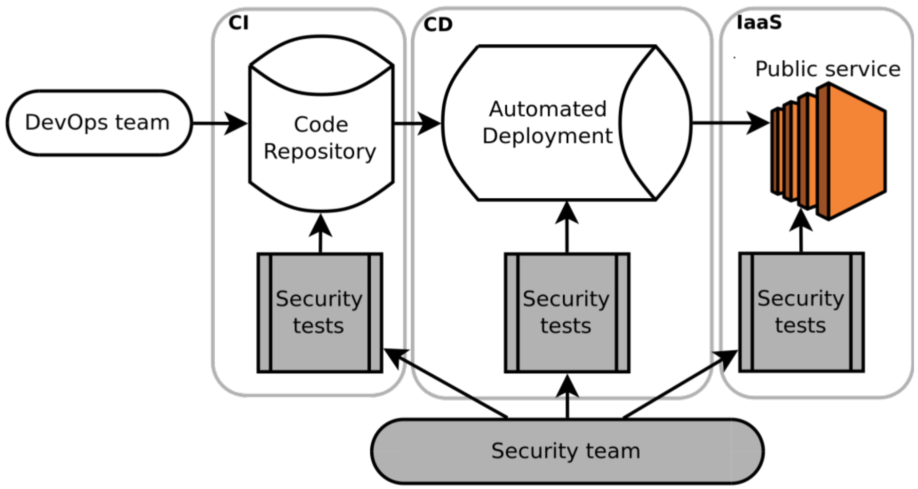
Classification of Testing Approaches

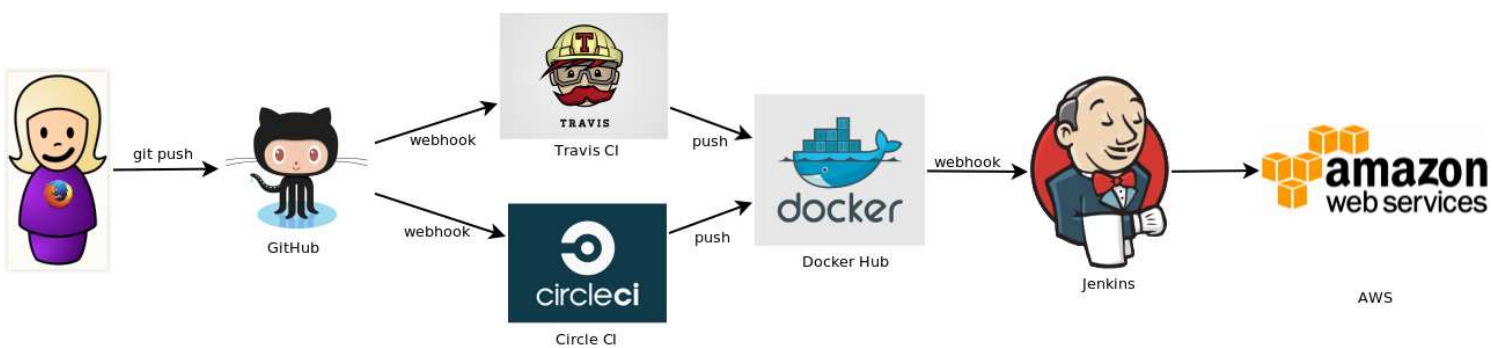


Manual white-box testing

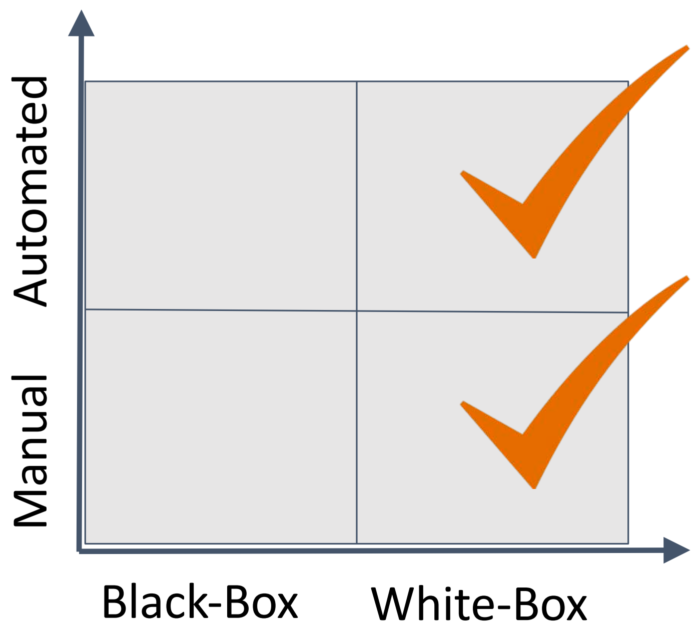
- Tests written by hand
- Full knowledge of source code/deployment/infrastructure
- Can test all parts
- Test *running* can be automated (e.g., on commits/deployment)

Test Driven Security





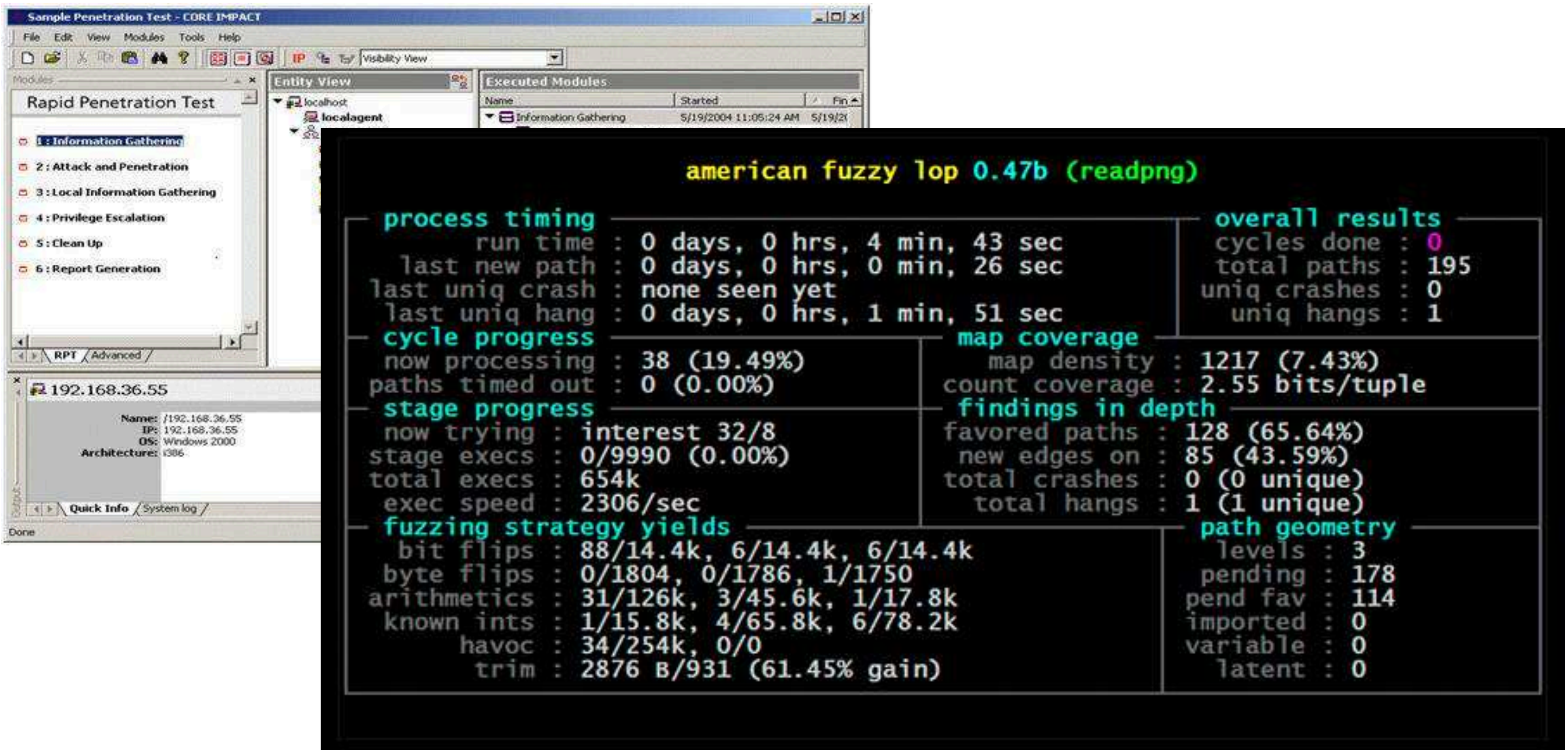
Classification of Testing Approaches



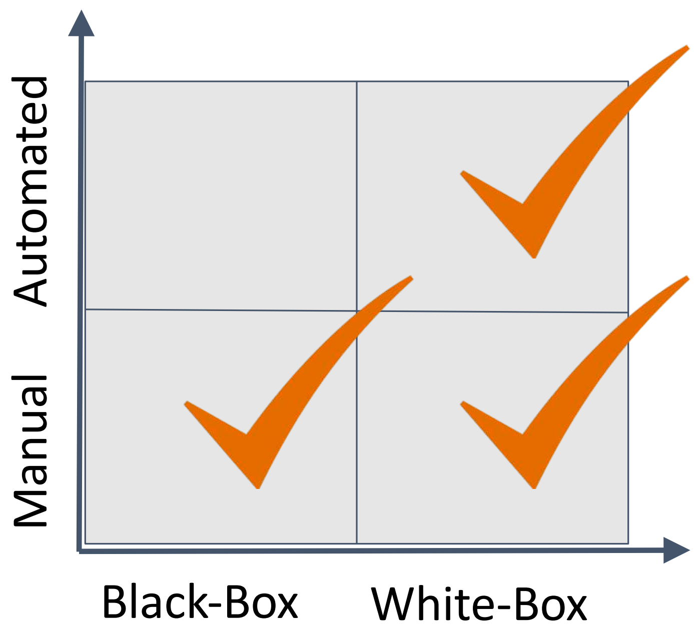
Automated white-box testing

- 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



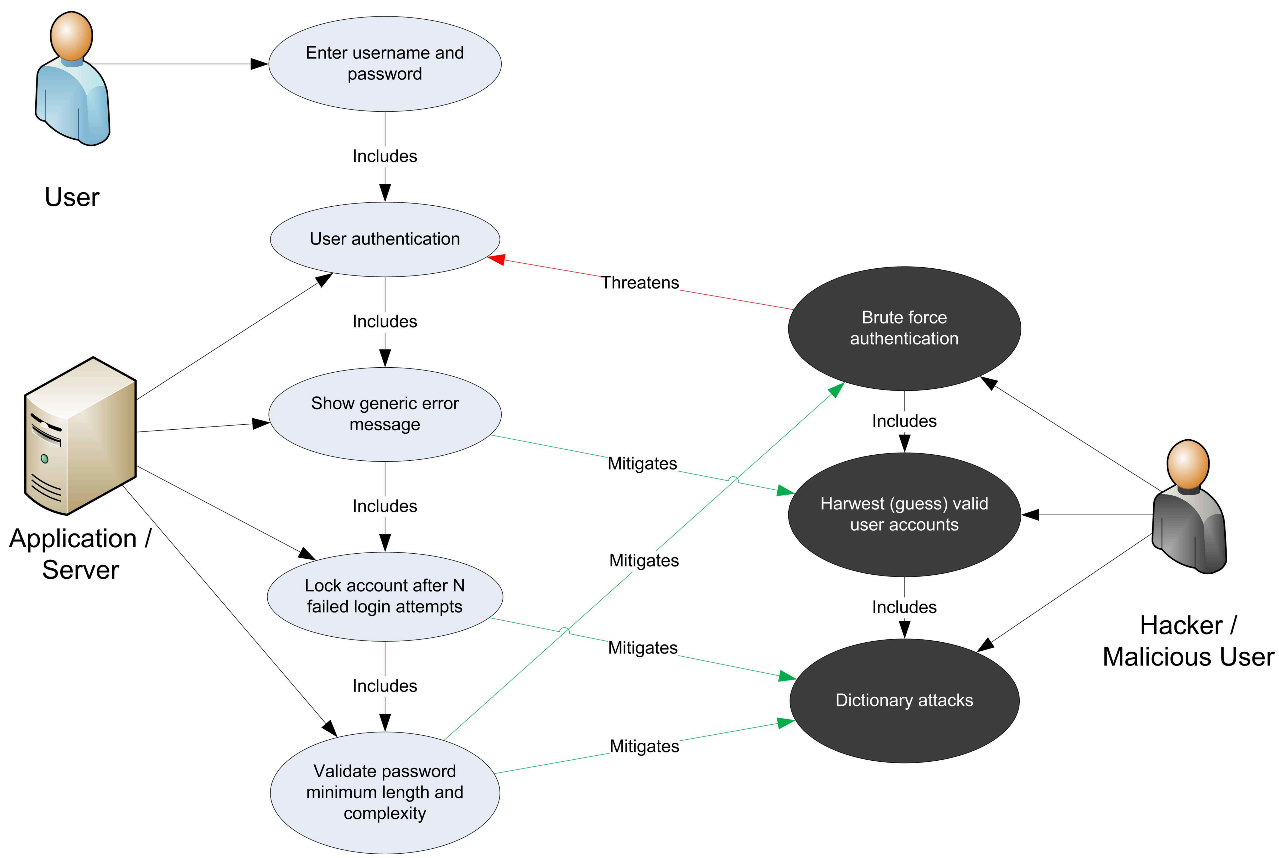
Classification of Testing Approaches



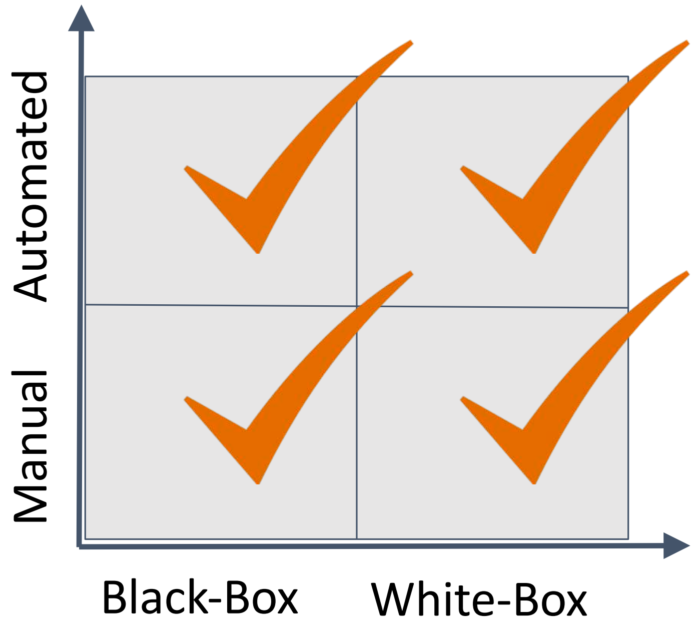
Manual black-box testing

- 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

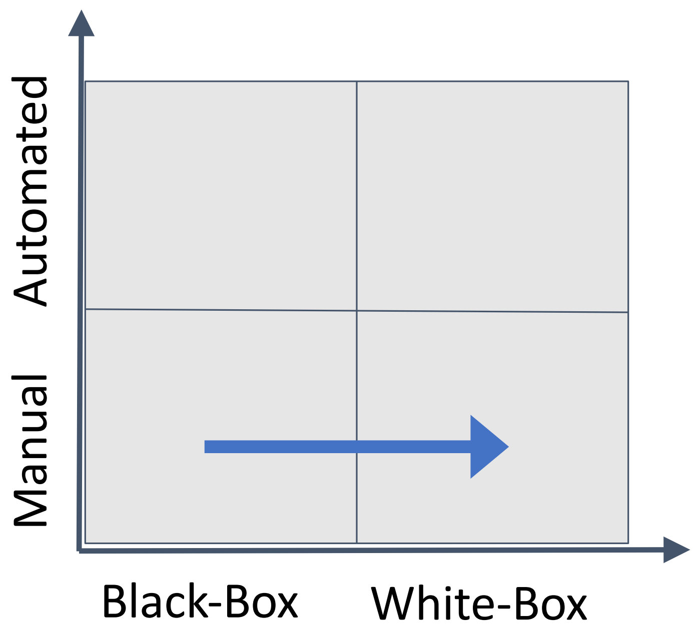
- Random Fuzzing
- “Dumb” (mutation-based) Fuzzing
  - Mutate an existing input
- “Smart” (generation-based) Fuzzing
  - Generate an input based on a model (grammar)

Mutation Fuzzer

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

```
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



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

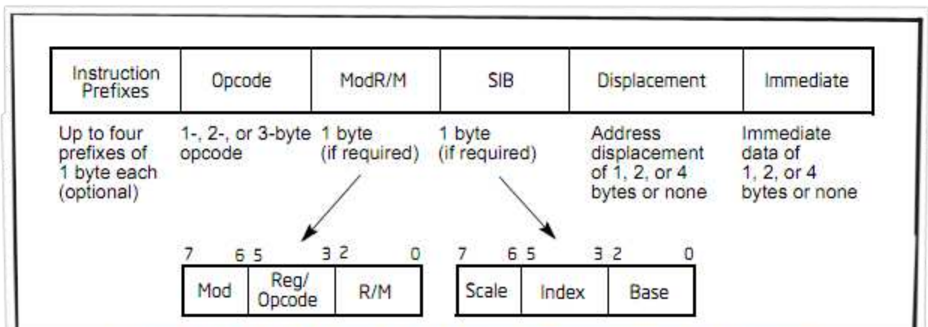
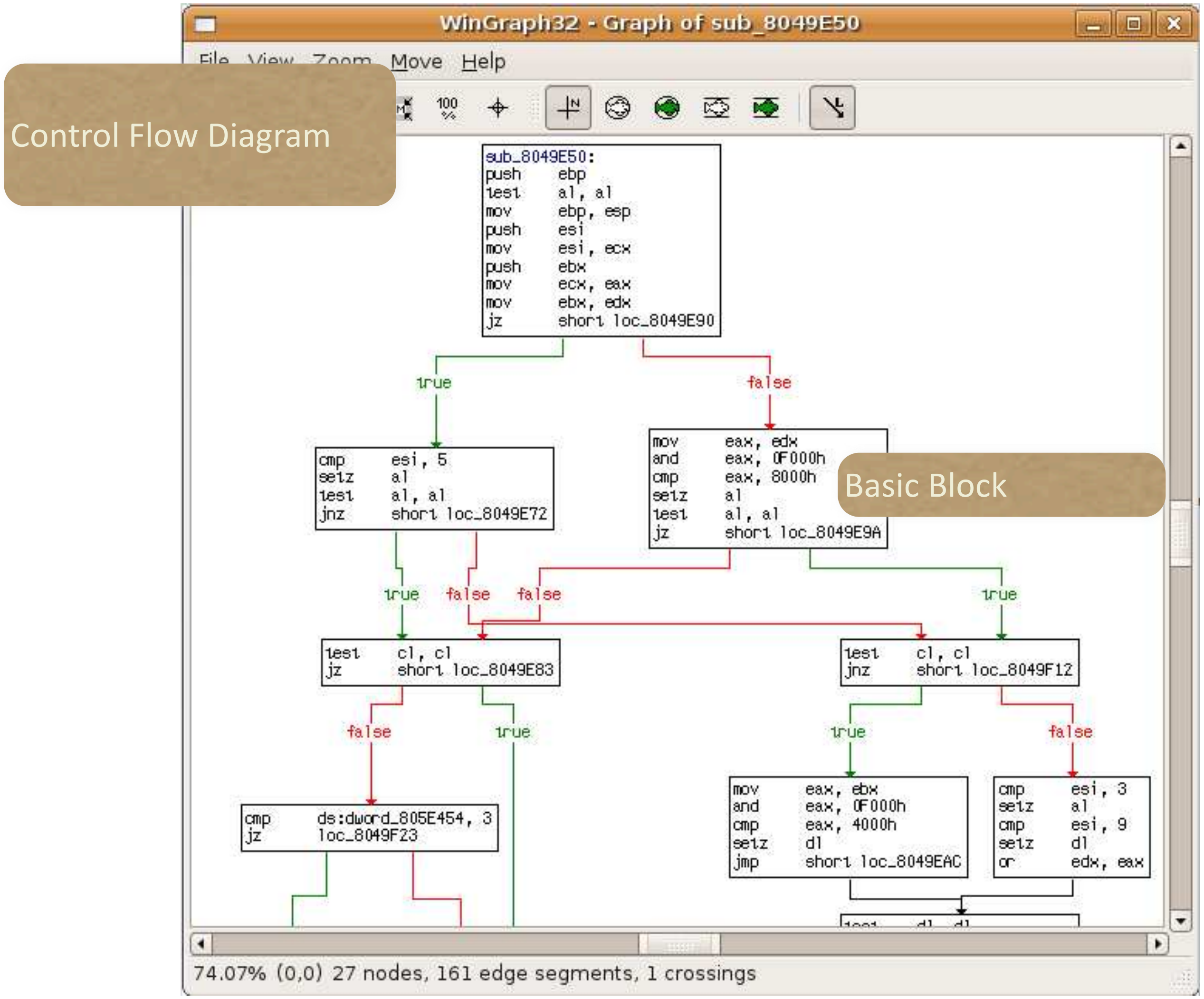


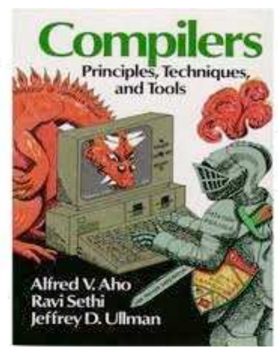
Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Bits	Hex Bytes	Instructions (human-readable)
11101011 00000110	0xEB 0x06	JMP +6
01010000	0x50	PUSH EAX


Control Flow Diagram
Basic Block
WinGraph32 - Graph of sub_8049E50
File View Zoom Move Help
sub_8049E50:
push ebp
test al, al
mov ebp, esp
push esi
mov esi, ecx
push ebx
mov ecx, eax
mov ebx, edx
jz short loc_8049E90
true
false
cmp esi, 5
setz al
test al, al
jnz short loc_8049E72
mov eax, edx
and eax, 0F000h
cmp eax, 8000h
setz al
test al, al
jz short loc_8049E9A
true
false
false
true
test cl, cl
jz short loc_8049E83
test cl, cl
jnz short loc_8049F12
false
true
true
false
cmp ds:dword_805E454, 3
jz loc_8049F23
mov eax, ebx
and eax, 0F000h
cmp eax, 4000h
setz dl
jmp short loc_8049EAC
cmp esi, 3
setz al
cmp esi, 9
setz dl
or edx, eax
74.07% (0,0) 27 nodes, 161 edge segments, 1 crossings

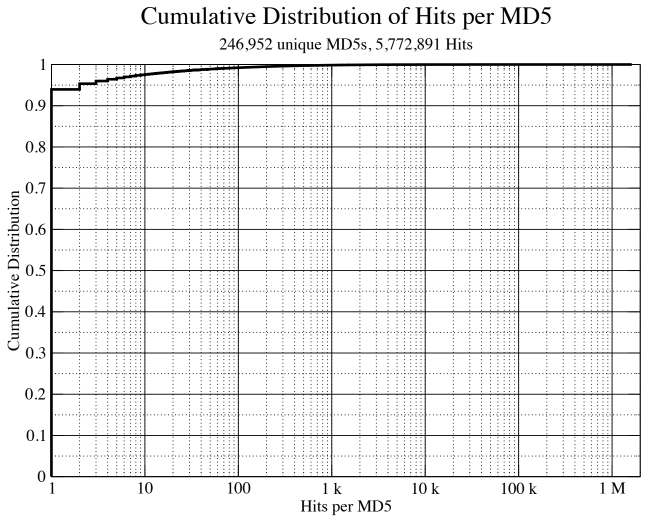
Difficulties

- Imperfect disassembly
- Benign Optimizations
  - Constant folding
  - Dead code elimination
  - Inline expansion
  - etc...
- Intentional Obfuscation
  - Packing
  - No-op instructions



Packing

- “Tons” of malware



Packer identification
98,801 malware samples

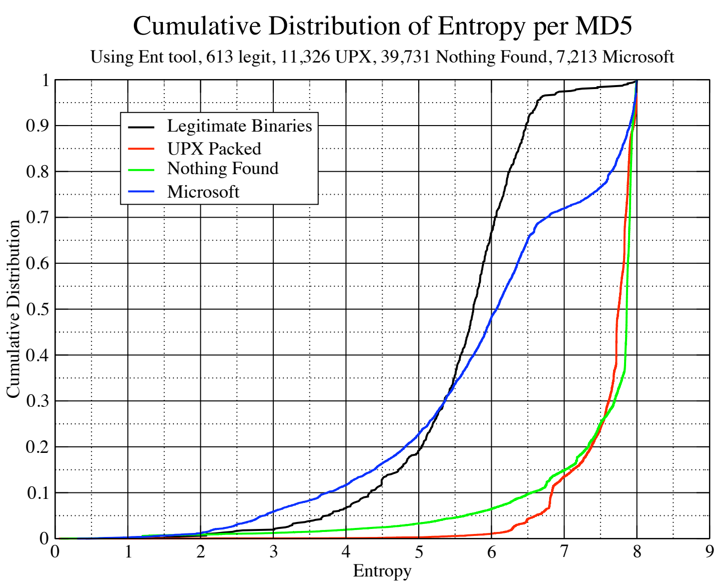
PEiD	Count
UPX	11244
Upack	6079
PECompact	4672
Nullsoft	2295
Themida	1688
FSG	1633
tElock	1398
NsPack	1375
ASpack	1283
WinUpack	1234

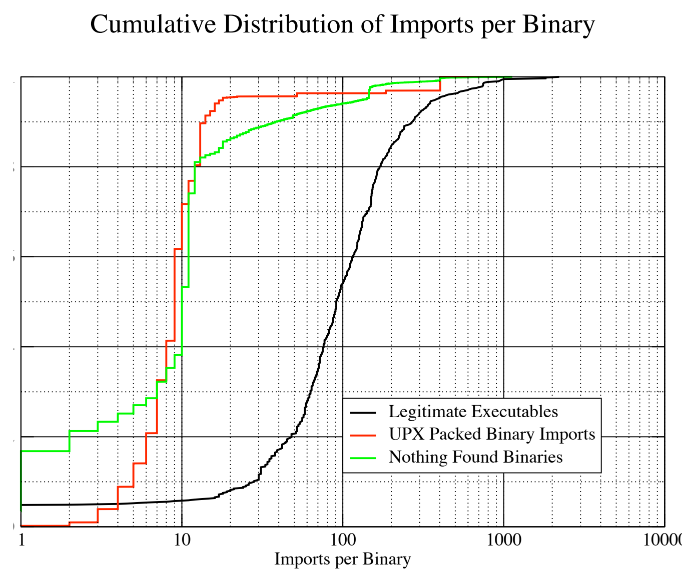
Identified: 59,070 (60%)
Top 10: 33.3%

SigBuster	Count
Allaple	22050
UPX	11324
PECompact	5278
FSG	5080
Upack	3639
Themida	1679
NsPack	1645
ASpack	1505
tElock	1332
Nullsoft	1058

Identified: 69,974 (71%)
Top 10: 55.3%

How about the unidentified?



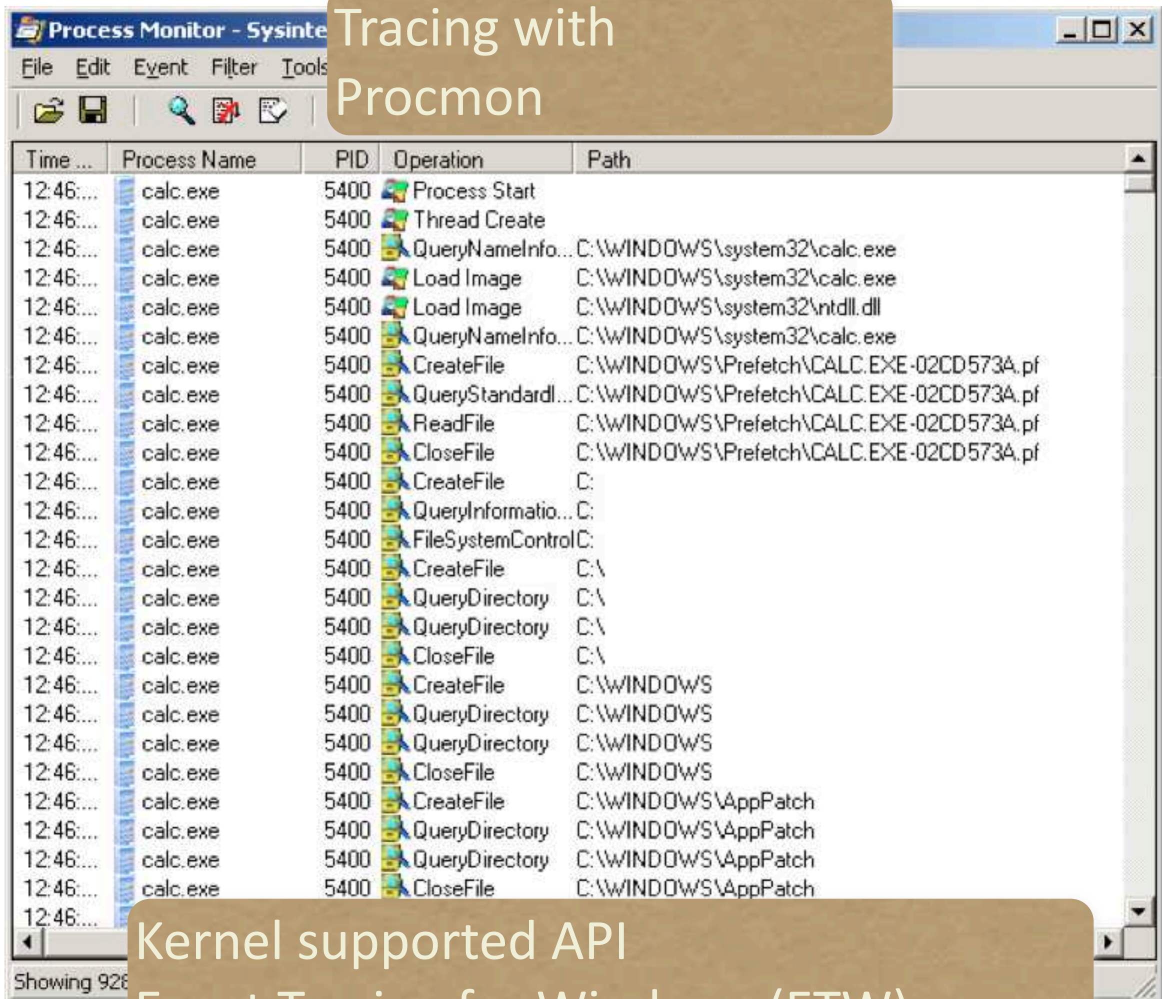


- Unidentified have: high entropy, small IATs
- Overall: > 90% packed

Dynamic Analysis

- A couple techniques available:
  - Tracing / Hooking
  - Debugging

Tracing with Procmon

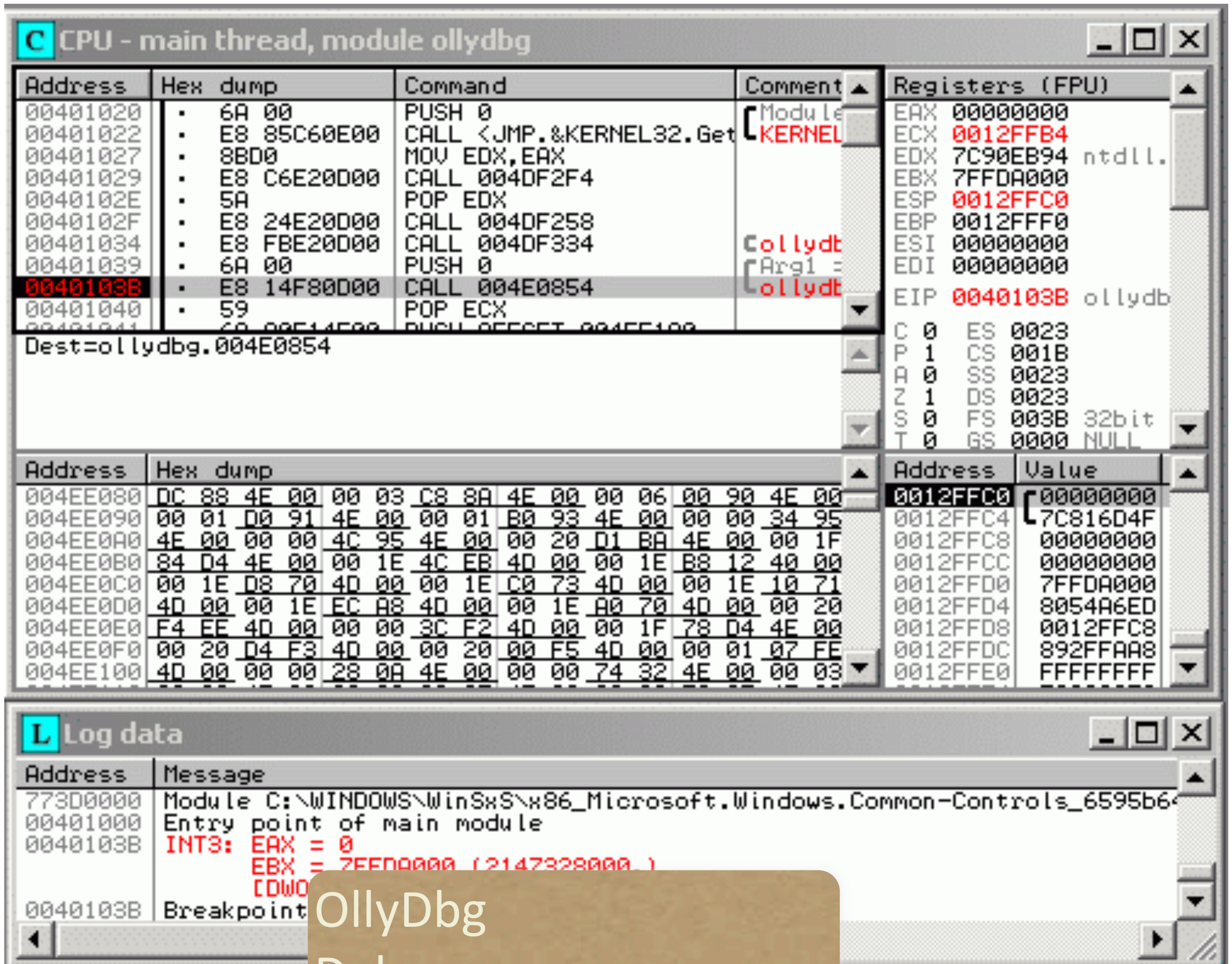


Kernel supported API

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



OllyDbg Debugger

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 malicous 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

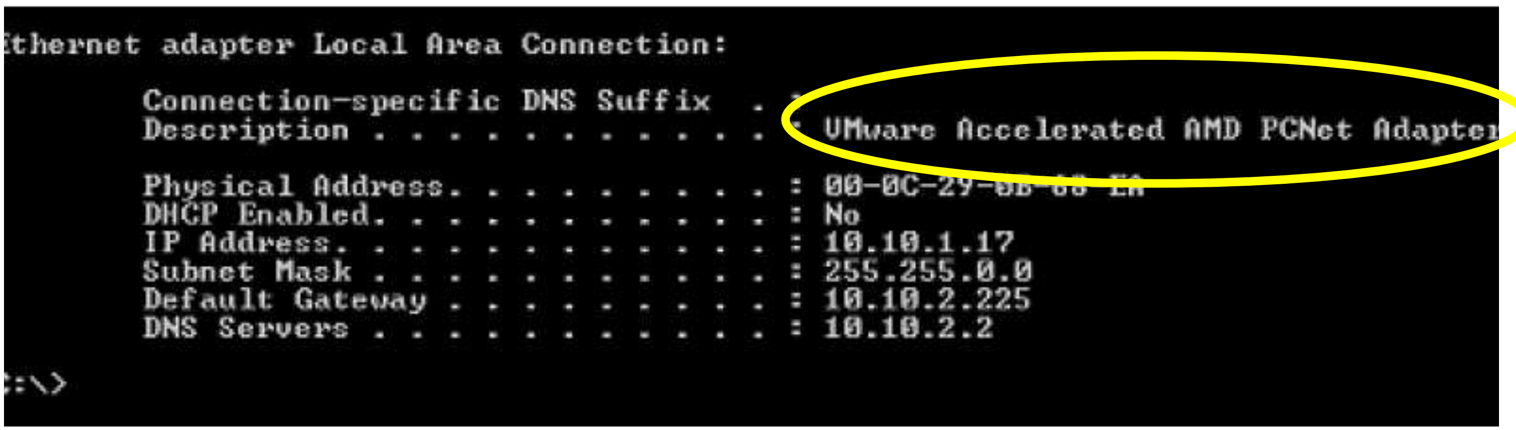
Easier

Harder

Layer of abstraction	Examples
Application	Installation, execution
Hardware	Device name, drivers
Environment	Memory and execution artifacts
Behavior	Timing

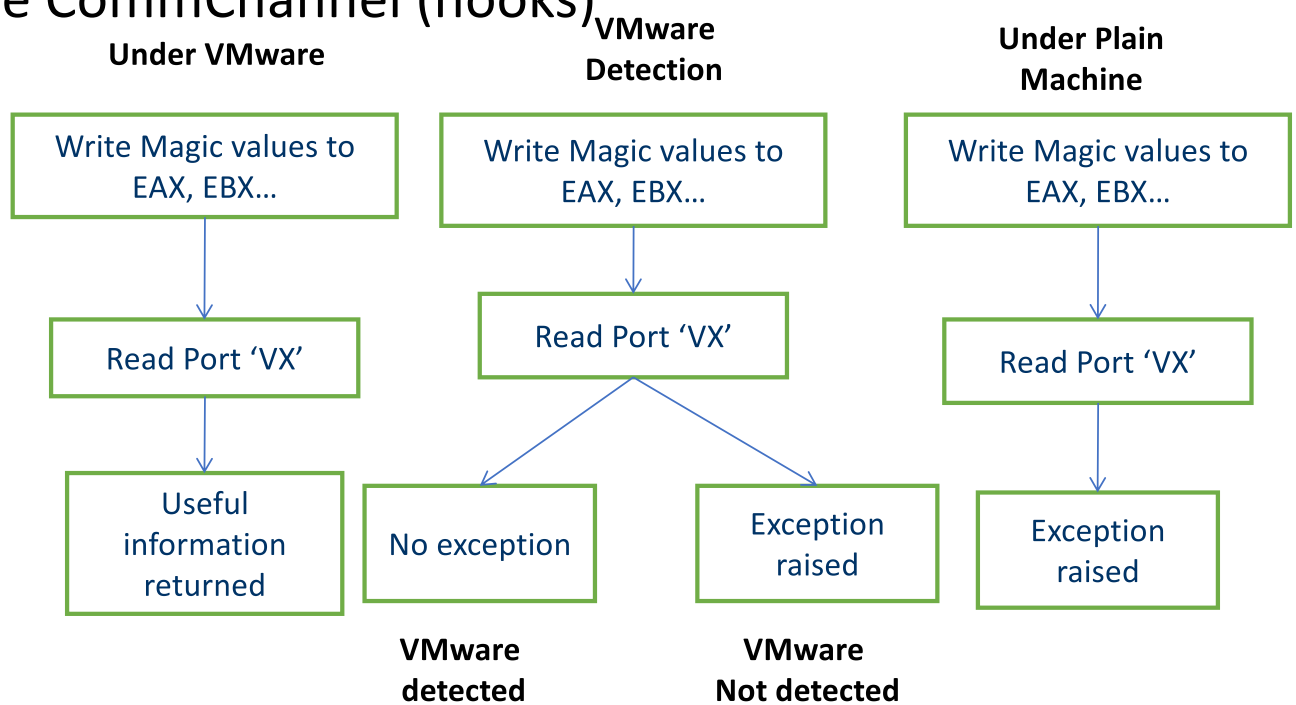
Example 1

- Device driver strings
  - Network cards



Example 2

- VMWare CommChannel (hooks)



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