

Lecture 14 – Testing

University of Illinois

ECE 422/CS

Goals

- By the end of this chapter you should:
 - Understand the various forms of testing and the dimensions that classify them
 - Provide examples of various types of testing
 - Understand the drawback and limitations of various testing methods
 - Articulate methods for reverse engineering
 - Recall the challenges associate with reverse engineering

Testing

- Testing Overview
- Automated White Box Tools
- Fuzzing
- Reverse Engineering

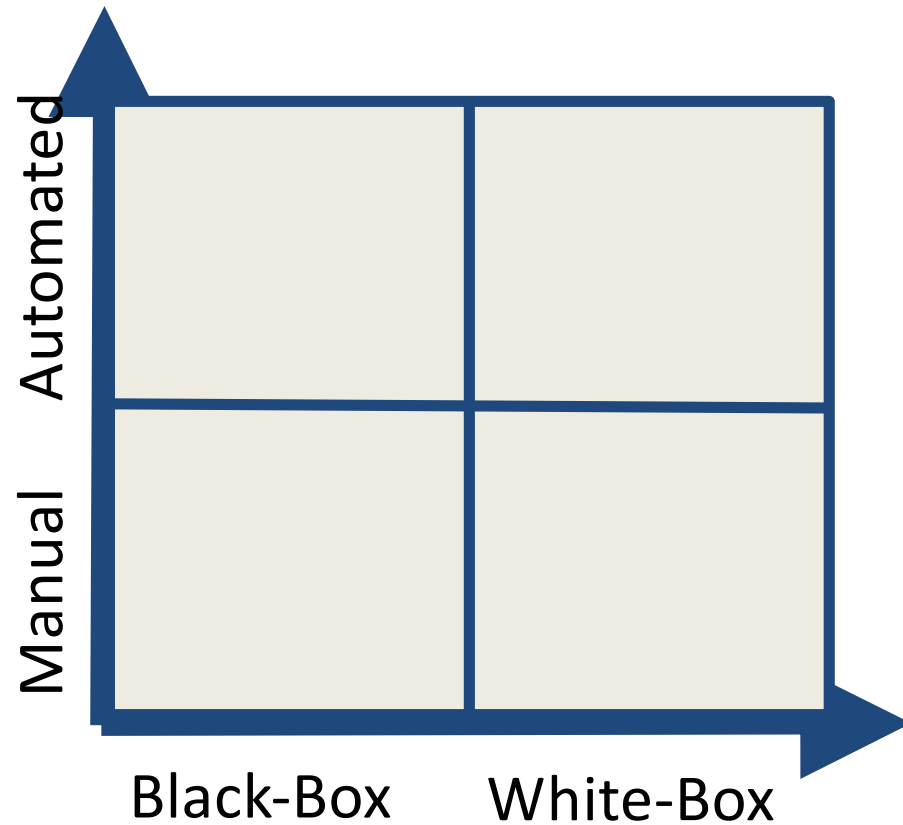
The Need for Specifications

- Testing checks whether program implementation agrees with program specification
- Without a specification, there is nothing to test!
- Testing 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

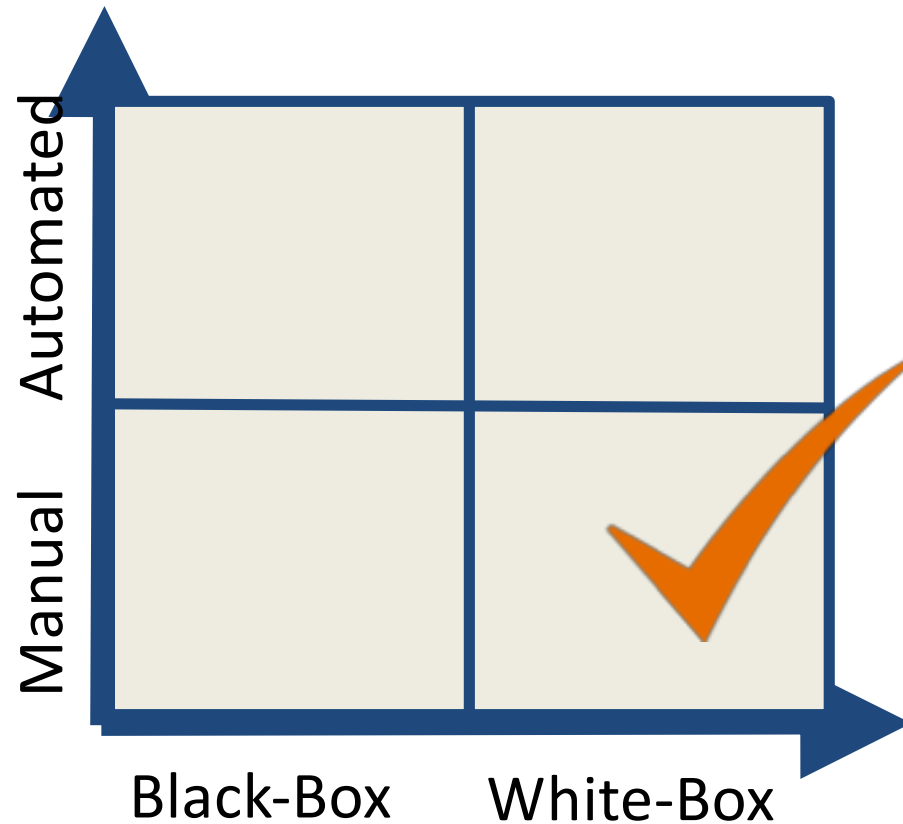
How Good Is Your Test Suite?

- 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

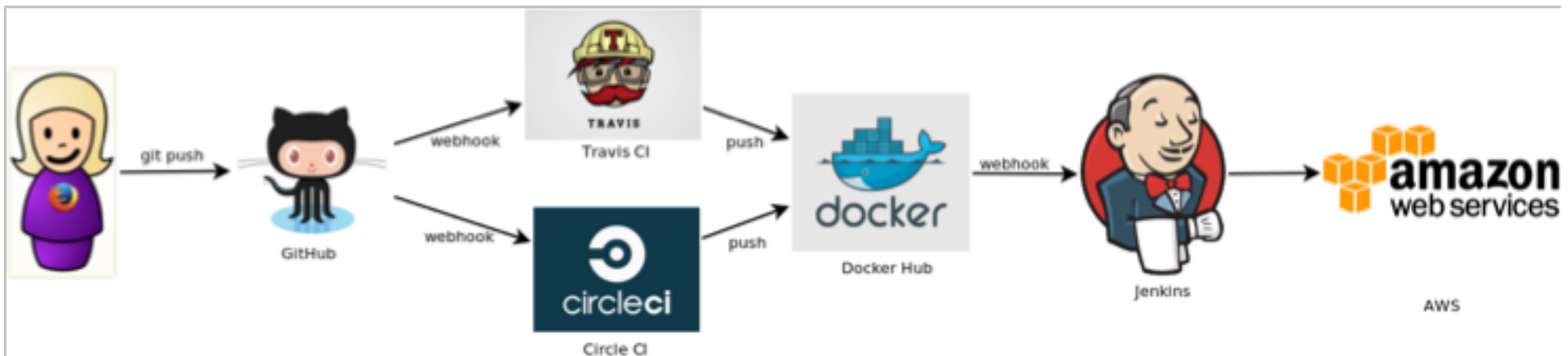
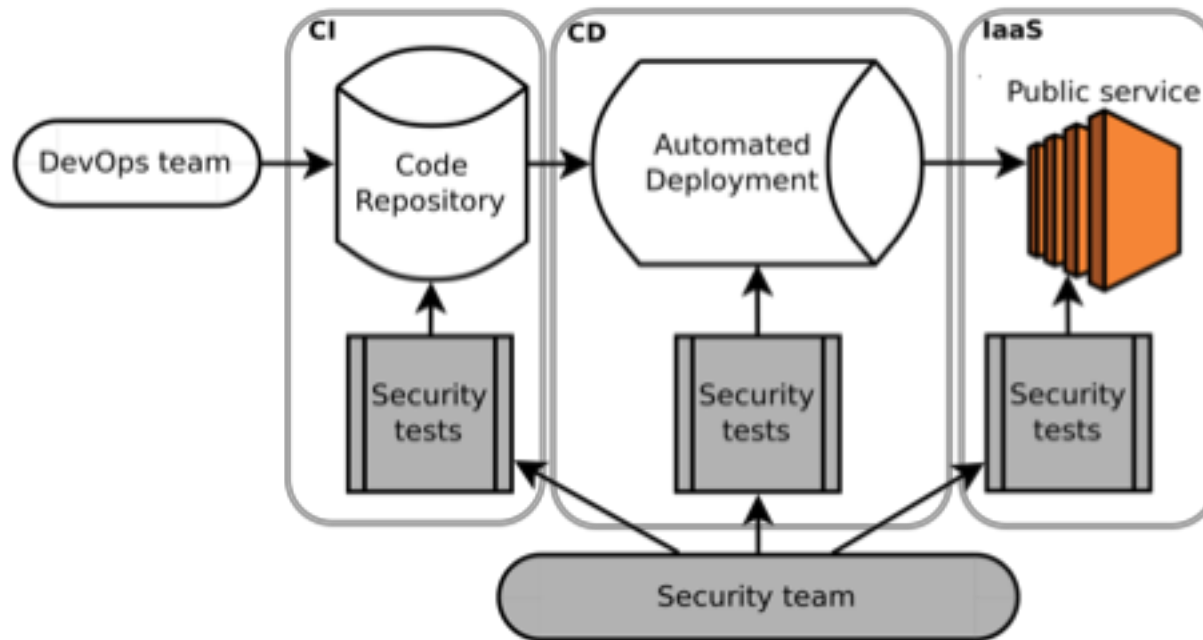
Code Coverage

- 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

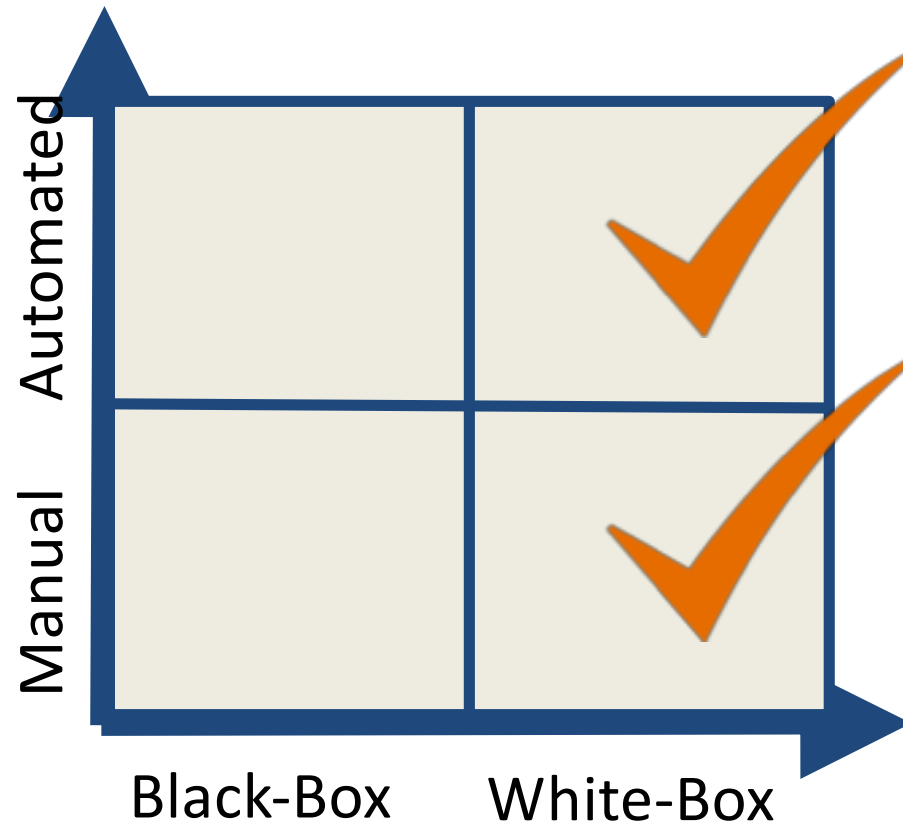
Classification of Testing Approaches



Test Driven Security



Classification of Testing Approaches



Automated White Box Testing

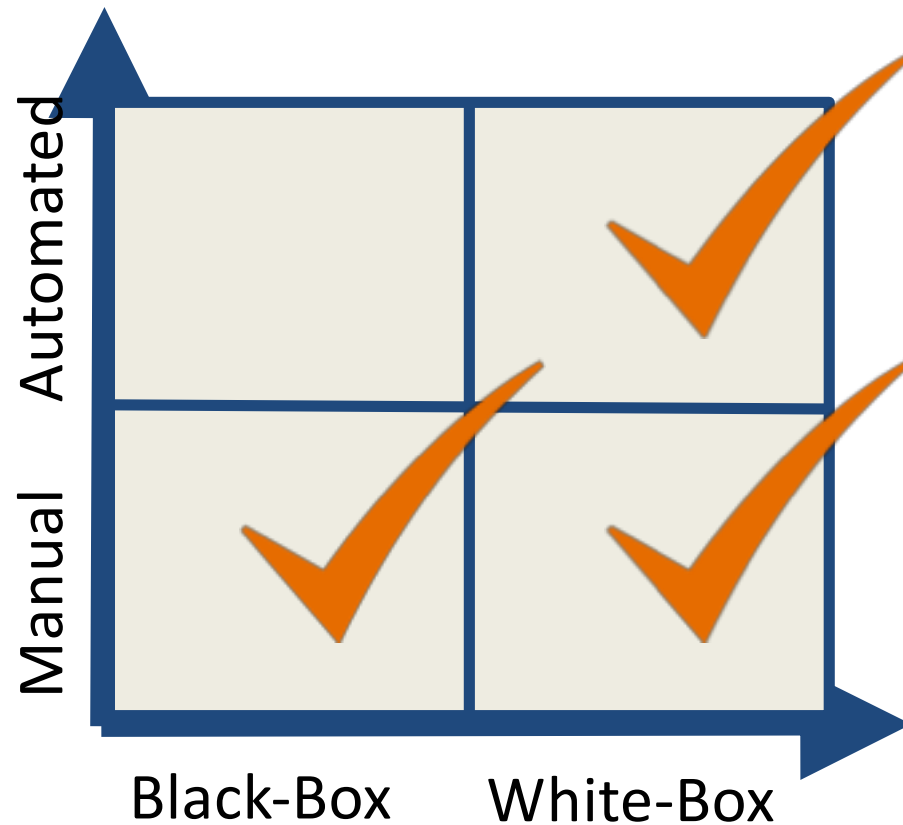
The image displays three screenshots of security tools used for automated white box testing.

Top Screenshot: Immunity Canvas
The interface shows a list of exploits on the left, including 'Current', 'cacheoverflow', 'cmsd_xdrarray', 'dispcid', 'it_ipid', 'kcms_server', 'portscan', 'rpcdump', 'sadmind', 'samba_nttrans', 'samba_nttrans2', 'snmpcidmd', 'sunlogin', 'sunlogin_pamh', 'tftp_xdrarray', and 'Exploits'. The 'dispcid' exploit is selected. The right pane shows references for CVE-2001-0803, dated Nov 06, 2001, with a CERT Advisory link.

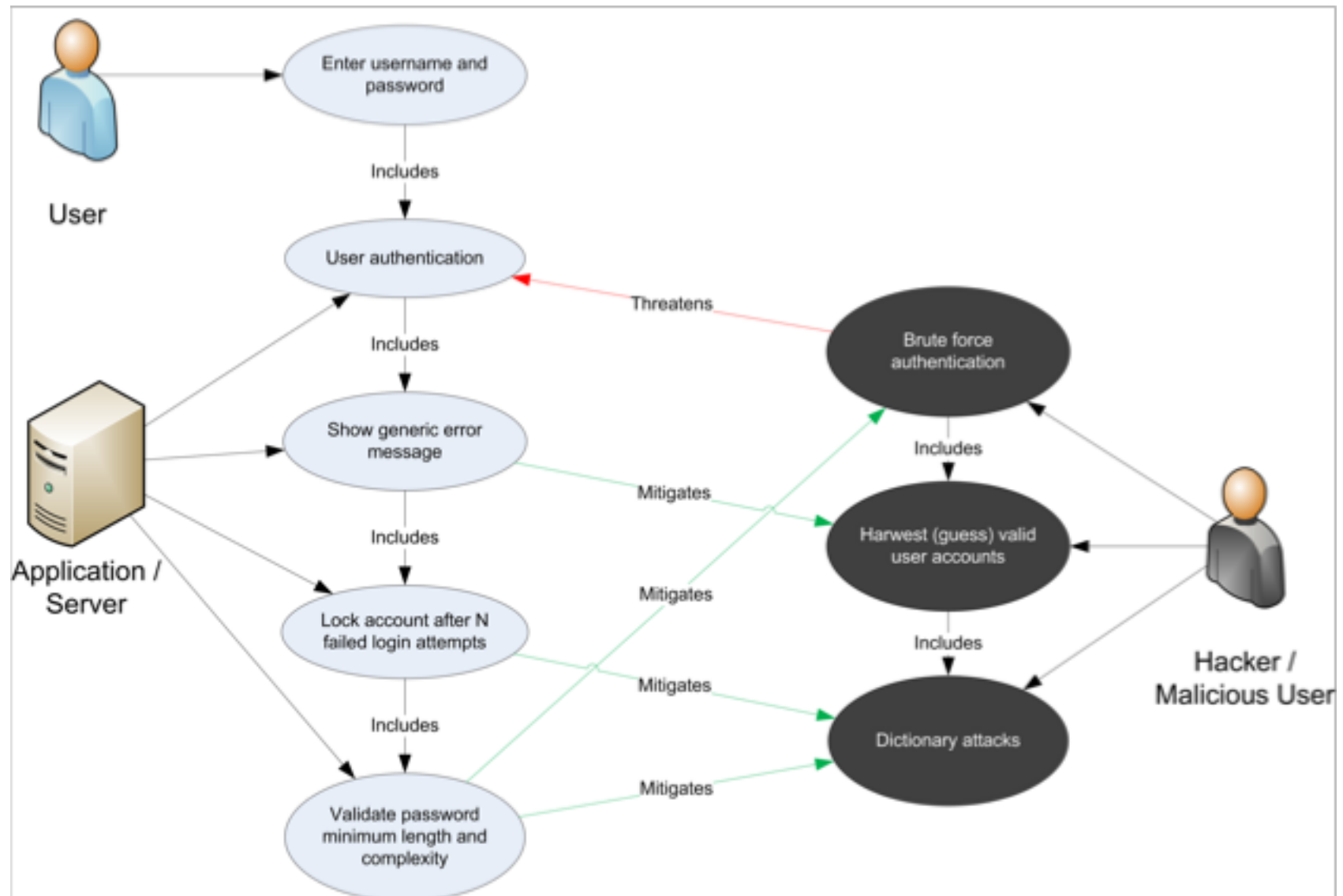
Middle Screenshot: Sample Penetration Test - CORE IMPACT
The interface shows a 'Rapid Penetration Test' workflow with steps: 1: Information Gathering, 2: Attack and Penetration, 3: Local Information Gathering, 4: Privilege Escalation, 5: Clean Up, and 6: Report Generation. The 'Entity View' shows a list of hosts, including 'localhost' and '192.168.36.55'. The 'Executed Modules' list shows 'Information Gathering' modules. The 'Executed Module Info' section displays 'Information Gathering' details for '192.168.36.55', including the operating system 'windows'.

Bottom Screenshot: Host Properties
The 'Host Properties' window shows details for '192.168.36.55', including Name, IP, OS (Windows 2000), and Architecture (x86).

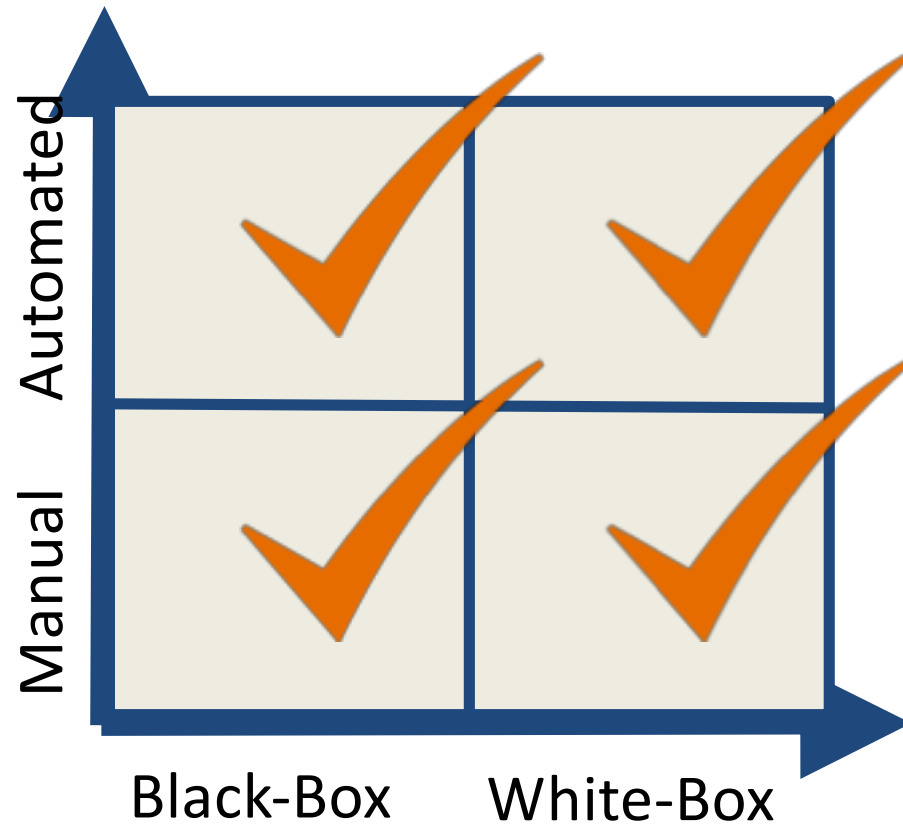
Classification of Testing Approaches



Web Pen Testing Simple Example



Classification of Testing Approaches



Fuzzing Components

- Test case generation
- Application execution
- Exception detection and logging

Test Case Generation

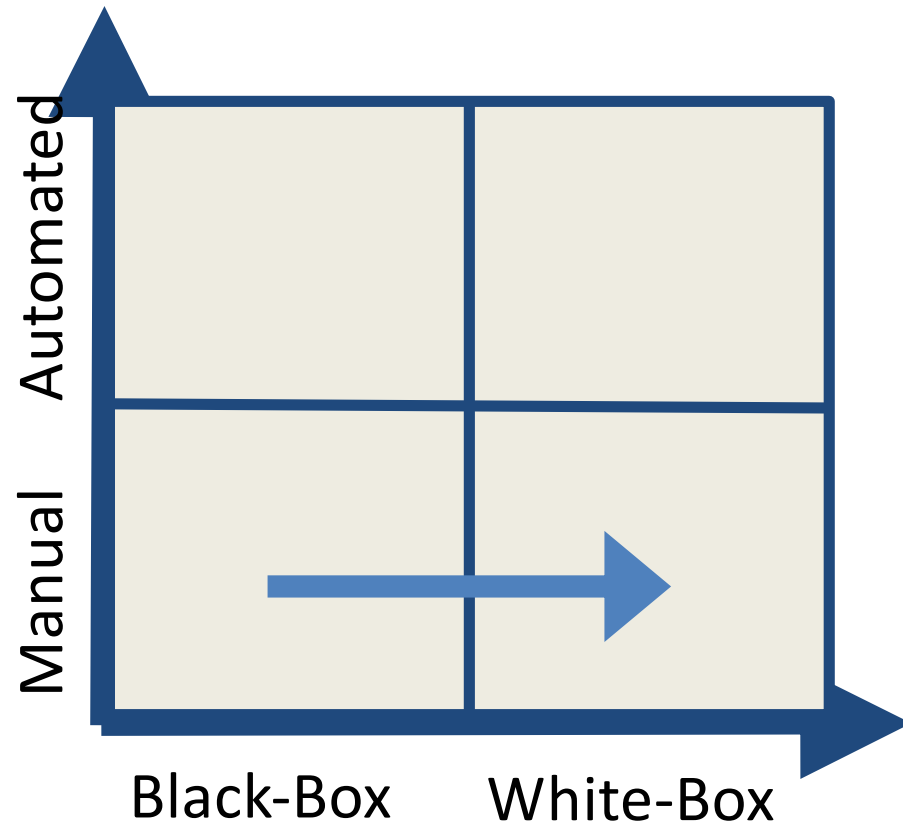
- Random Fuzzing
- “Dumb” (mutation-based) Fuzzing
 - Mutate an existing input
- “Smart” (generation-based) Fuzzing
 - Generate an input based on a model (grammar)

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 (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

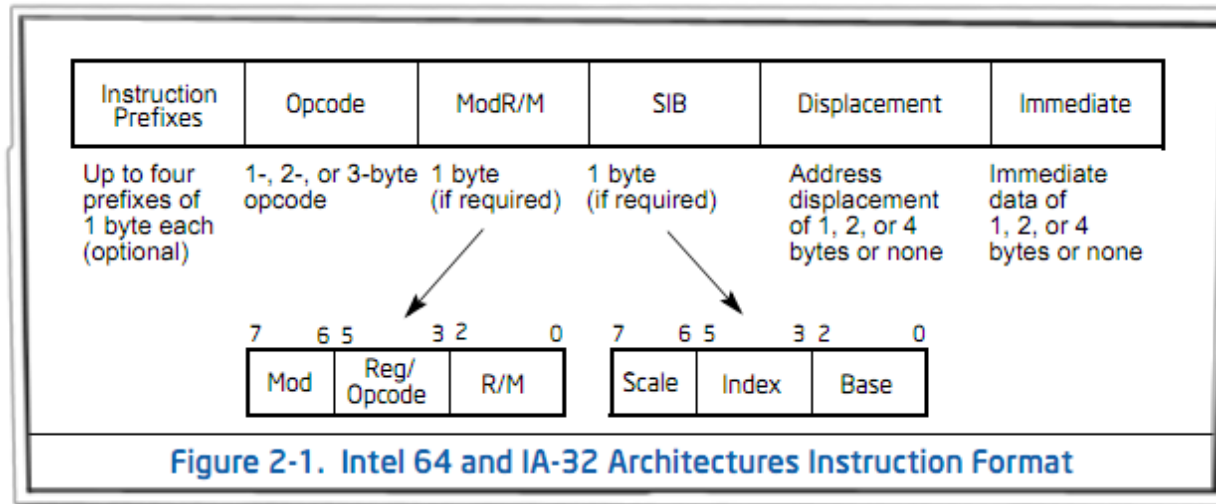
Legality

- 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”

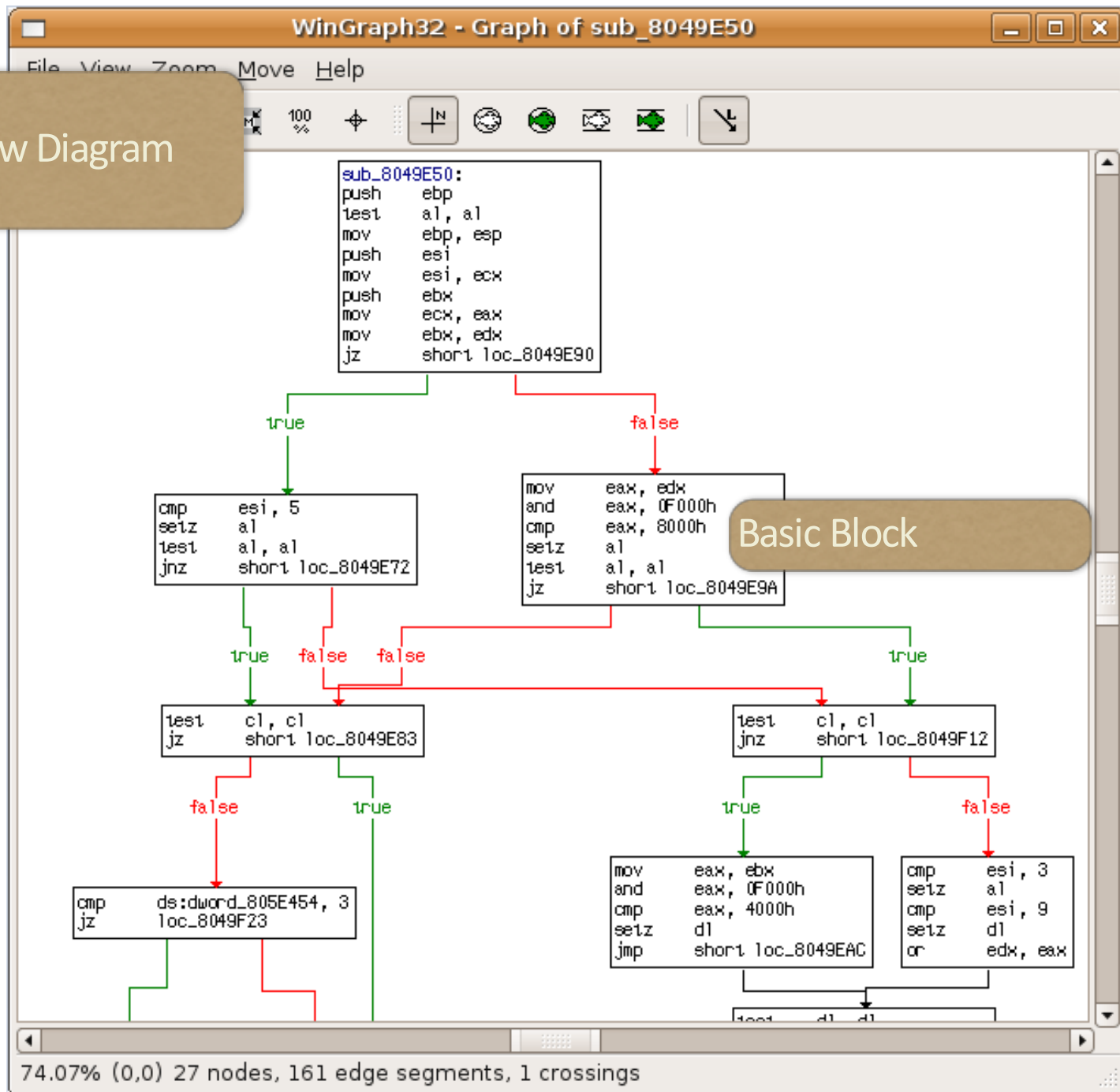
Two Techniques

- Static Code Analysis (structure)
 - Disassemblers
- Dynamic Code Analysis (operation)
 - Tracing / Hooking
 - Debuggers

Disassembly

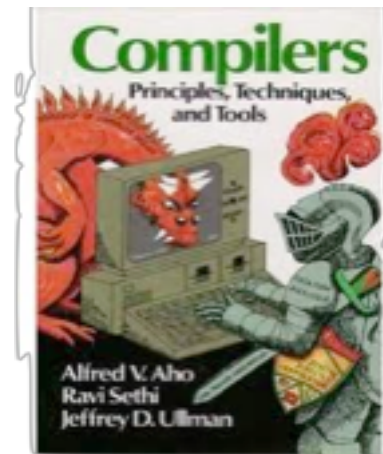


Control Flow Diagram



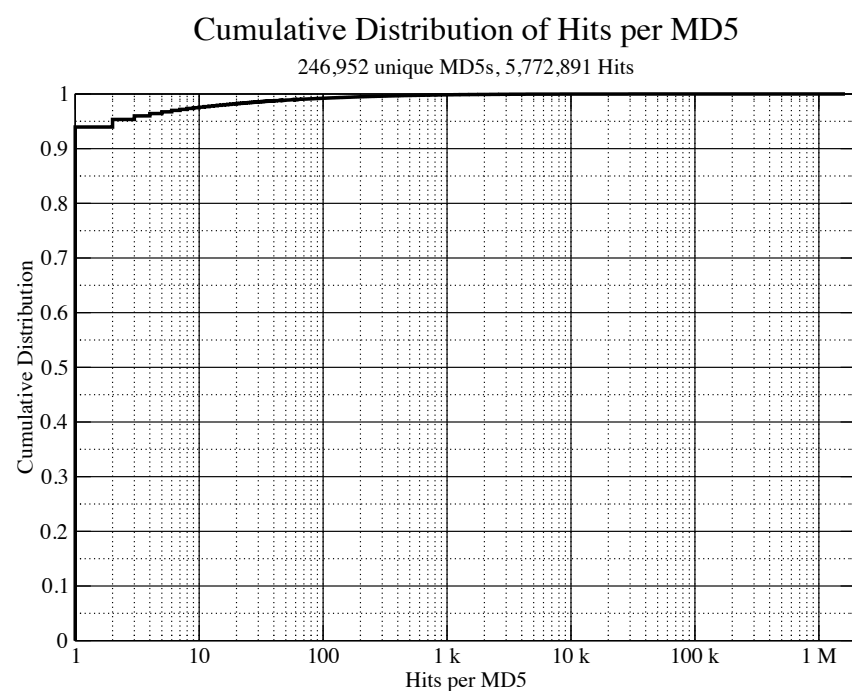
Difficulties

- 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%

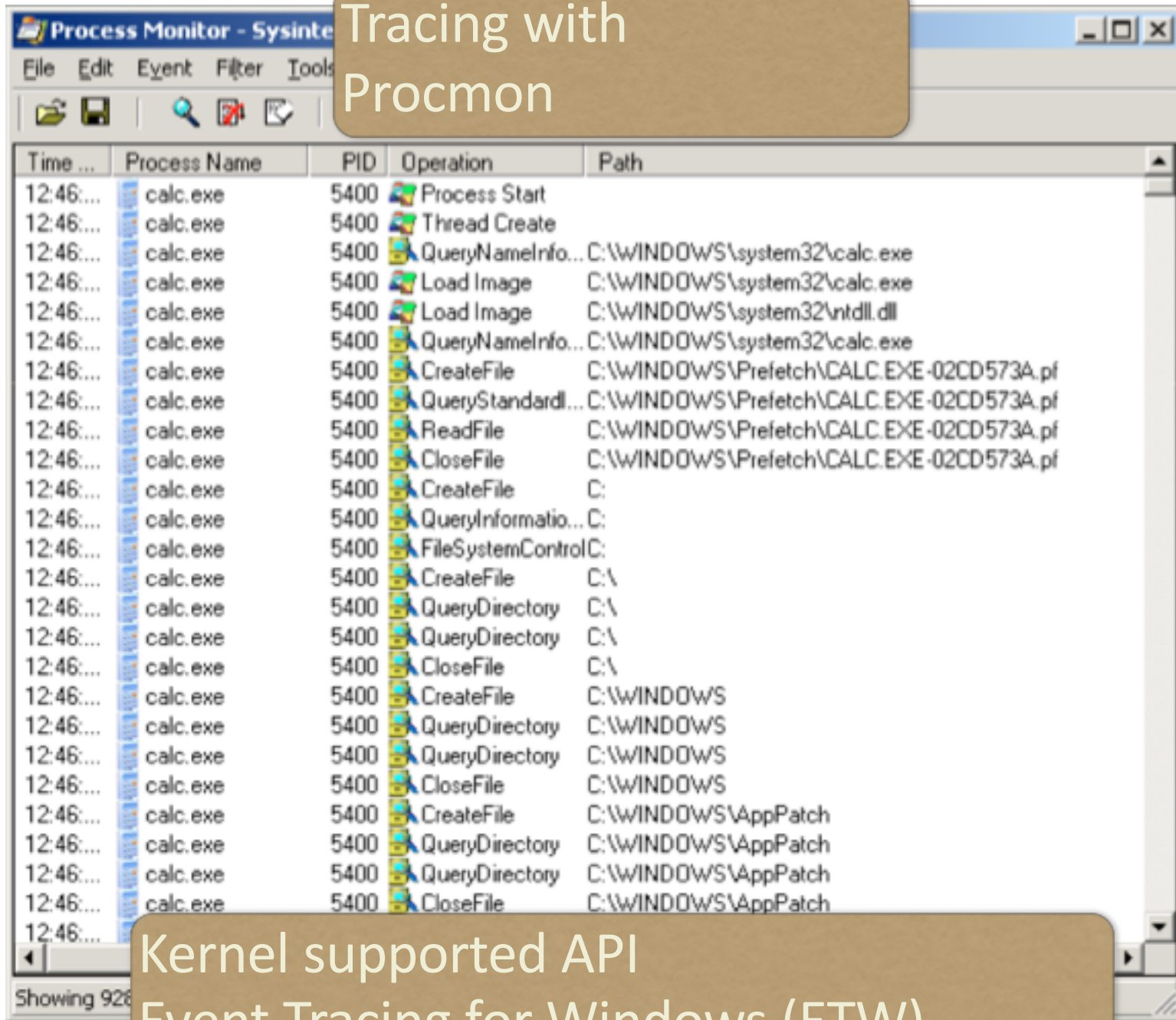
SigBuster	Count
Allaple	22050
UPX	11324
PECompact	5278
FSG	5080
Upack	3639
Themida	1679
NsPack	1645
ASpack	1505
tElock	1332
Nullsoft	1058

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

Dynamic Analysis

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

Tracing with Procmon



Time ...	Process Name	PID	Operation	Path
12:46:...	calc.exe	5400	Process Start	
12:46:...	calc.exe	5400	Thread Create	
12:46:...	calc.exe	5400	QueryNameInfo...	C:\WINDOWS\system32\calc.exe
12:46:...	calc.exe	5400	Load Image	C:\WINDOWS\system32\calc.exe
12:46:...	calc.exe	5400	Load Image	C:\WINDOWS\system32\ntdll.dll
12:46:...	calc.exe	5400	QueryNameInfo...	C:\WINDOWS\system32\calc.exe
12:46:...	calc.exe	5400	CreateFile	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pf
12:46:...	calc.exe	5400	QueryStandardl...	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pf
12:46:...	calc.exe	5400	ReadFile	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pf
12:46:...	calc.exe	5400	CloseFile	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pf
12:46:...	calc.exe	5400	CreateFile	C:
12:46:...	calc.exe	5400	QueryInformatio...	C:
12:46:...	calc.exe	5400	FileSystemControlC:	C:
12:46:...	calc.exe	5400	CreateFile	C:\
12:46:...	calc.exe	5400	QueryDirectory	C:\
12:46:...	calc.exe	5400	QueryDirectory	C:\
12:46:...	calc.exe	5400	CloseFile	C:\
12:46:...	calc.exe	5400	CreateFile	C:\WINDOWS
12:46:...	calc.exe	5400	QueryDirectory	C:\WINDOWS
12:46:...	calc.exe	5400	QueryDirectory	C:\WINDOWS
12:46:...	calc.exe	5400	CloseFile	C:\WINDOWS
12:46:...	calc.exe	5400	CreateFile	C:\WINDOWS\AppPatch
12:46:...	calc.exe	5400	QueryDirectory	C:\WINDOWS\AppPatch
12:46:...	calc.exe	5400	QueryDirectory	C:\WINDOWS\AppPatch
12:46:...	calc.exe	5400	CloseFile	C:\WINDOWS\AppPatch
12:46:...	calc.exe	5400		

Showing 926

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

CPU - main thread, module ollydbg

Address	Hex dump	Command	Comment	Registers (FPU)
00401020	• 6A 00	PUSH 0	Module KERNEL	EAX 00000000
00401022	• E8 85C60E00	CALL <JMP.&KERNEL32.Get		ECX 0012FFB4
00401027	• 8BD0	MOV EDX,EAX		EDX 7C90EB94 ntdll.
00401029	• E8 C6E20D00	CALL 004DF2F4		EBX 7FFDA000
0040102E	• 5A	POP EDX		ESP 0012FFC0
0040102F	• E8 24E20D00	CALL 004DF258		EBP 0012FFFF
00401034	• E8 FBE20D00	CALL 004DF334	Collydb	ESI 00000000
00401039	• 6A 00	PUSH 0	Arg1 =	EDI 00000000
0040103B	• E8 14F80D00	CALL 004E0854	Collydb	EIP 0040103B ollydbg
00401040	• 59	POP ECX		C 0 ES 0023
00401041	• 60 00F14F00	PUSH 00F14F00		P 1 CS 001B
Dest=ollydbg.004E0854				A 0 SS 0023
				Z 1 DS 0023
				S 0 FS 003B 32bit
				T 0 GS 0000 NULL

Address	Hex dump	Address	Value
004EE080	DC 88 4E 00 00 03 C8 8A 4E 00 00 06 00 90 4E 00	0012FFC0	00000000
004EE090	00 01 D0 91 4E 00 00 01 B0 93 4E 00 00 00 34 95	0012FFC4	7C816D4F
004EE0A0	4E 00 00 00 4C 95 4E 00 00 20 D1 BA 4E 00 00 1F	0012FFC8	00000000
004EE0B0	84 D4 4E 00 00 1E 4C EB 4D 00 00 1E B8 12 40 00	0012FFCC	00000000
004EE0C0	00 1E D8 70 4D 00 00 1E C0 73 4D 00 00 1E 10 71	0012FFD0	7FFDA000
004EE0D0	4D 00 00 1E EC A8 4D 00 00 1E A0 70 4D 00 00 20	0012FFD4	8054A6ED
004EE0E0	F4 EE 4D 00 00 00 3C F2 4D 00 00 1F 78 D4 4E 00	0012FFD8	0012FFC8
004EE0F0	00 20 D4 F3 4D 00 00 20 00 F5 4D 00 00 01 07 FE	0012FFDC	892FFAA8
004EE100	4D 00 00 00 28 0A 4E 00 00 00 74 32 4E 00 00 03	0012FFE0	FFFFFFFF

Log data

Address	Message
773D0000	Module C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b641-1046-11d1-8475-005056959611_x-ww-6595b641-1046-11d1-8475-005056959611
00401000	Entry point of main module
0040103B	INT3: EAX = 0 EBX = 7FFDA000 (2147328000) EDX = 00000000 ECX = 00000000 ESI = 00000000 EDI = 00000000 EIP = 0040103B
0040103B	Breakpoint

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 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*

Example 1

- Device driver strings
 - Network cards

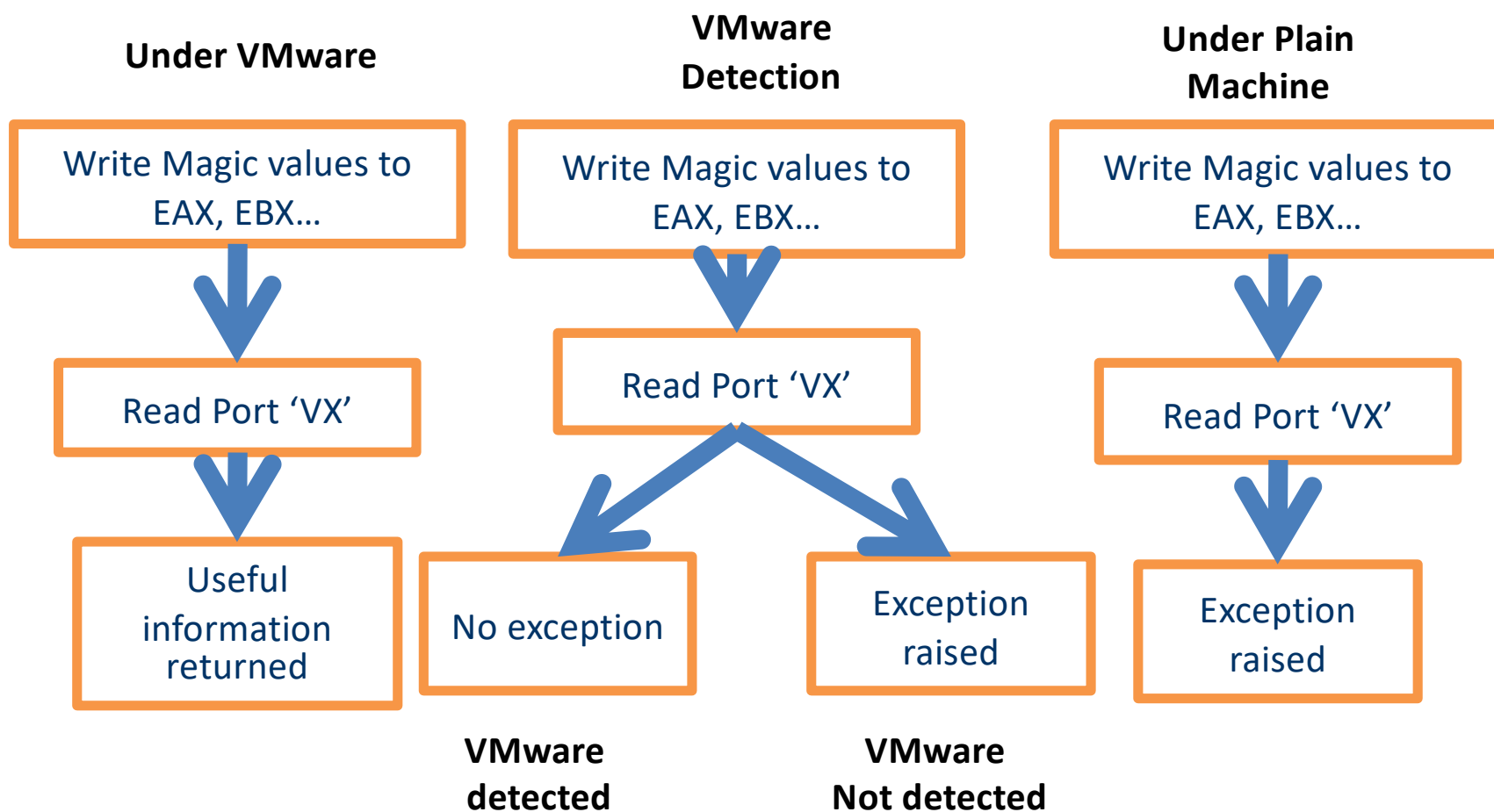
```
Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : 
    Description . . . . . : VMware Accelerated AMD PCNet Adapter
    Physical Address. . . . . : 00-0C-29-0B-68-1A
    DHCP Enabled. . . . . : No
    IP Address. . . . . : 10.10.1.17
    Subnet Mask . . . . . : 255.255.0.0
    Default Gateway . . . . . : 10.10.2.225
    DNS Servers . . . . . : 10.10.2.2

C:\>
```

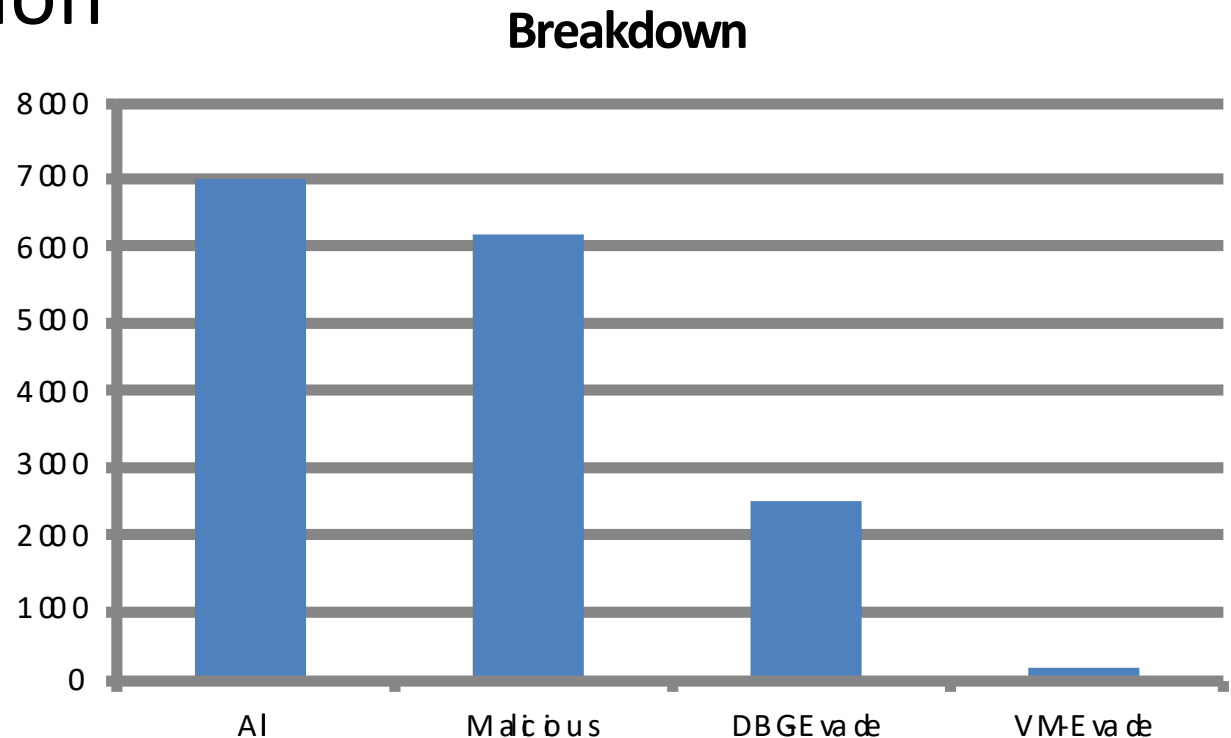
Example 2

- VMWare CommChannel (hooks)



Prevalence of evasion

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



To Learn More ...

- Books
 - Stallings and Brown, Chapter 6
 - Pfleeger and Pfleeger, Chapter 3
 - Goodrich and Tamassia, Chapter 4
 - Anderson, Chapter 21
 - Easttom, Chapter 5
- Papers
 - Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software - Newsome*
 - Efficient Software-Based Fault Isolation
 - Scheduling Black-box Mutational Fuzzing
 - Skyfire: Data-Driven Seed Generation for Fuzzing - Wang