Lecture 06 – Malware Defenses, Testing

Michael Bailey
University of Illinois
ECE 422/CS 461 – Spring 2018

MALWARE DEFENSES

Introduction

- Terminology
  - IDS: Intrusion detection system
  - IPS: Intrusion prevention system
  - HIDS/NIDS: Host/Network Based IDS
- Difference between IDS and IPS
  - Detection happens after the attack is conducted (i.e. the memory is already corrupted due to a buffer overflow attack)
  - Prevention stops the attack before it reaches the system (i.e. shield does packet filtering)
- Anomaly vs. Misuse, Rule-based

Signatures: A Malware Countermeasure

- Scan compare the analyzed object with a database of signatures
- A signature is a virus fingerprint
  - E.g.,a string with a sequence of instructions specific for each virus
  - Different from a digital signature
- A file is infected if there is a signature inside its code
  - Fast pattern matching techniques to search for signatures
- All the signatures together create the malware database that usually is proprietary

White/Black Listing

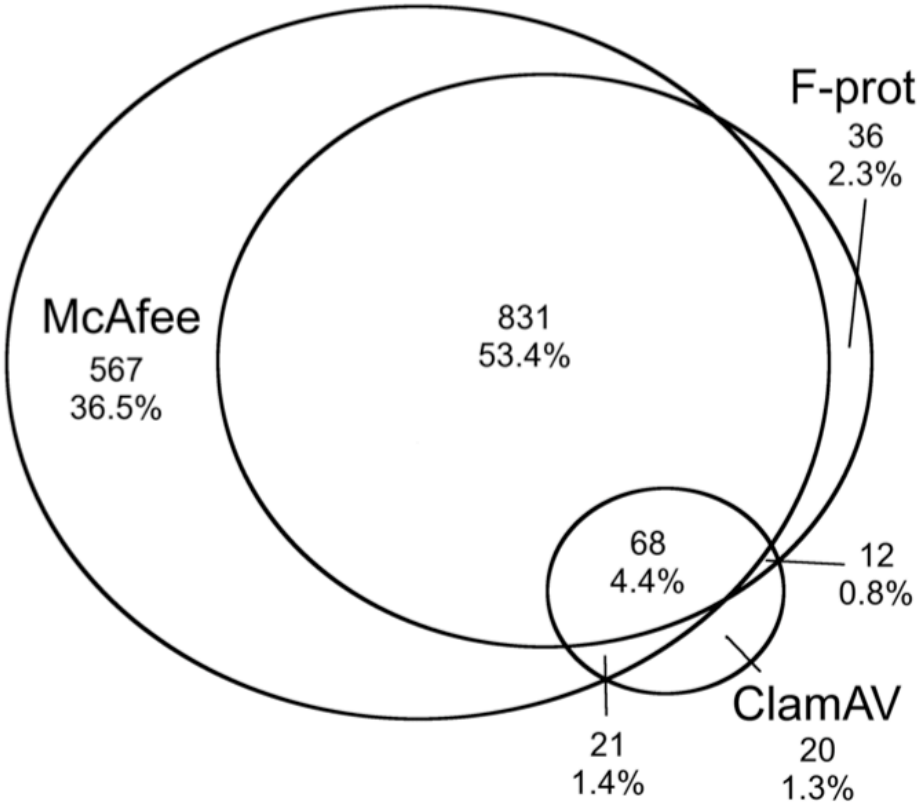
- Maintain database of cryptographic hashes for
  - Operating system files
  - Popular applications
  - Known infected files
- Compute hash of each file
- Look up into database
- Needs to protect the integrity of the database

Heuristic Analysis

- Useful to identify new and “zero day” malware
- Code analysis
  - Based on the instructions, the antivirus can determine whether or not the program is malicious, i.e., program contains instruction to delete system files,
- Execution emulation
  - Run code in isolated emulation environment
  - Monitor actions that target file takes
  - If the actions are harmful, mark as virus
- Heuristic methods can trigger false alarms

SDBot

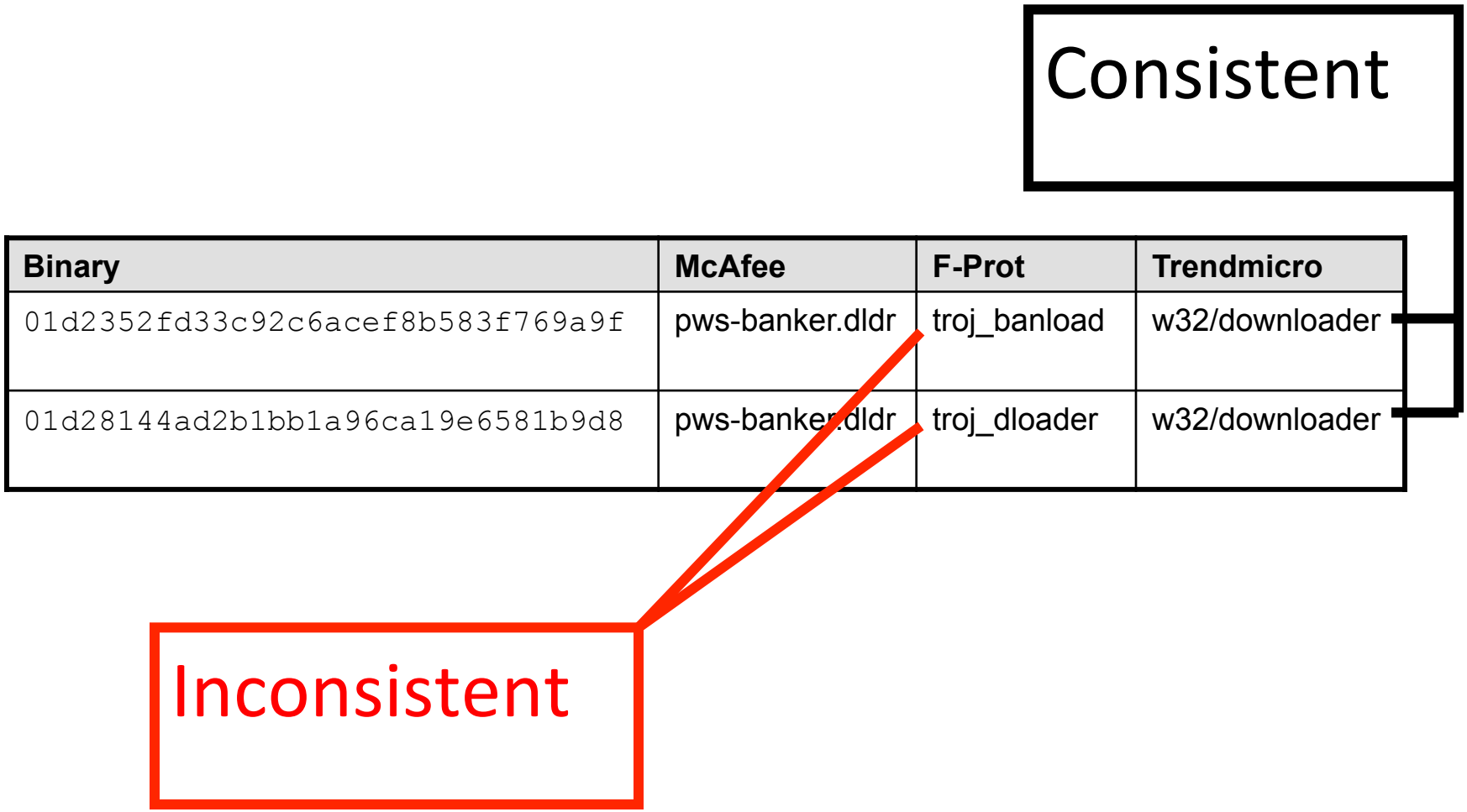
- Via manual inspection find all SDBot variants, and alias detected by McAfee, ClamAV, F-Prot



Properties of a good labeling system

- **Consistency.** Identical items must and similar items should be assigned the same label.
- **Completeness.** A label should be generated for as many items as possible.
- **…**

Consistency example



Consistency

- The percentage of time two binaries classified as the same by one AV system are classified the same by other AV systems.
- ***AV system labels are inconsistent***

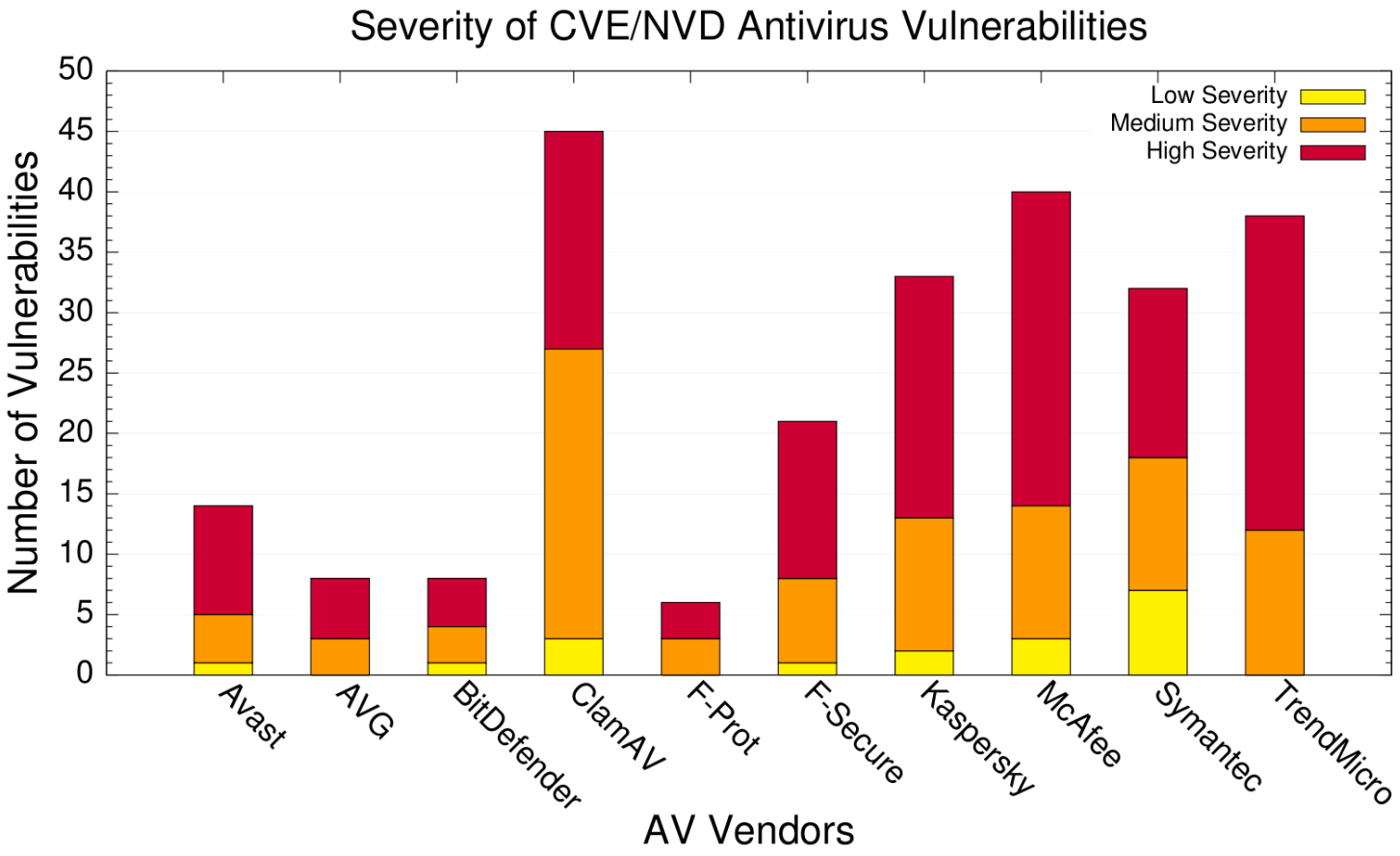
AV	McAfee	F-Prot	ClamAV	Trend	Symantec
McAfee	100	13	27	39	59
F-Prot	50	100	96	41	61
ClamAV	62	57	100	34	68
Trend	67	18	25	100	55
Symantec	27	7	13	14	100

Completeness

- The percentage of malware samples detected across datasets and AV vendors
- ***AV system labels are incomplete***

Dataset	AV Updated	Percentage of Malware Samples Detected				
		McAfee	F-Prot	ClamAV	Trend	Symantec
legacy	20 Nov 2006	100	99.8	94.8	93.73	97.4
small	20 Nov 2006	48.7	61.0	38.4	54.0	76.9
small	31 Mar 2007	67.4	68.0	55.5	86.8	52.4
large	31 Mar 2007	54.6	76.4	60.1	80.0	51.5

Antivirus Vulnerabilities

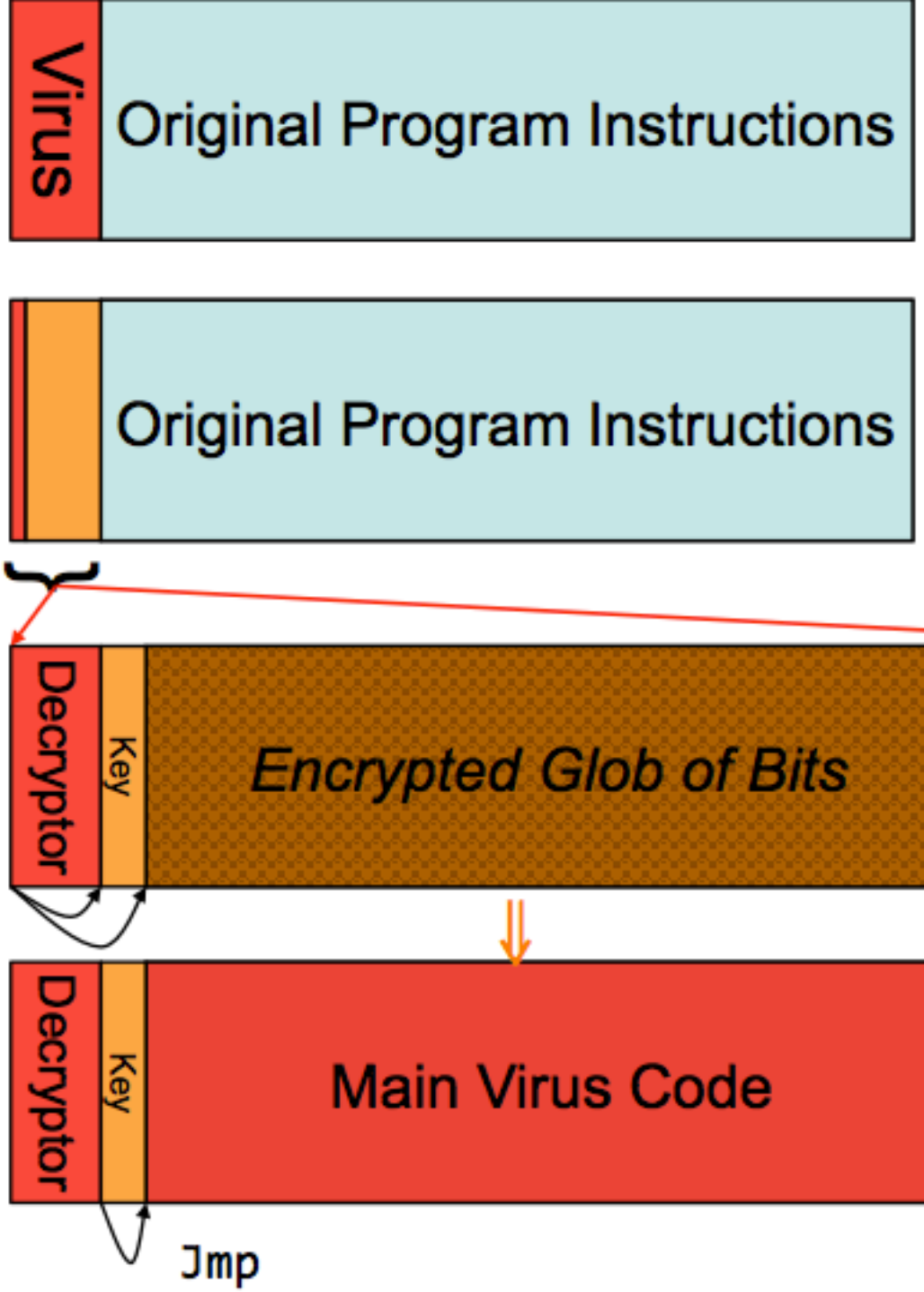


Antivirus engines vulnerable to numerous local and remote exploits

(number of vulnerabilities reported in NVD from Jan. 2005 to Nov. 2007)

Concealment

- Encrypted virus
  - Decryption engine + encrypted body
  - Randomly generate encryption key
  - Detection looks for decryption engine
- Polymorphic virus
  - Encrypted virus with random variations of the decryption engine (e.g., padding code)
  - Detection using CPU emulator
- Metamorphic virus
  - Different virus bodies
  - Approaches include code permutation and instruction replacement
  - Challenging to detect



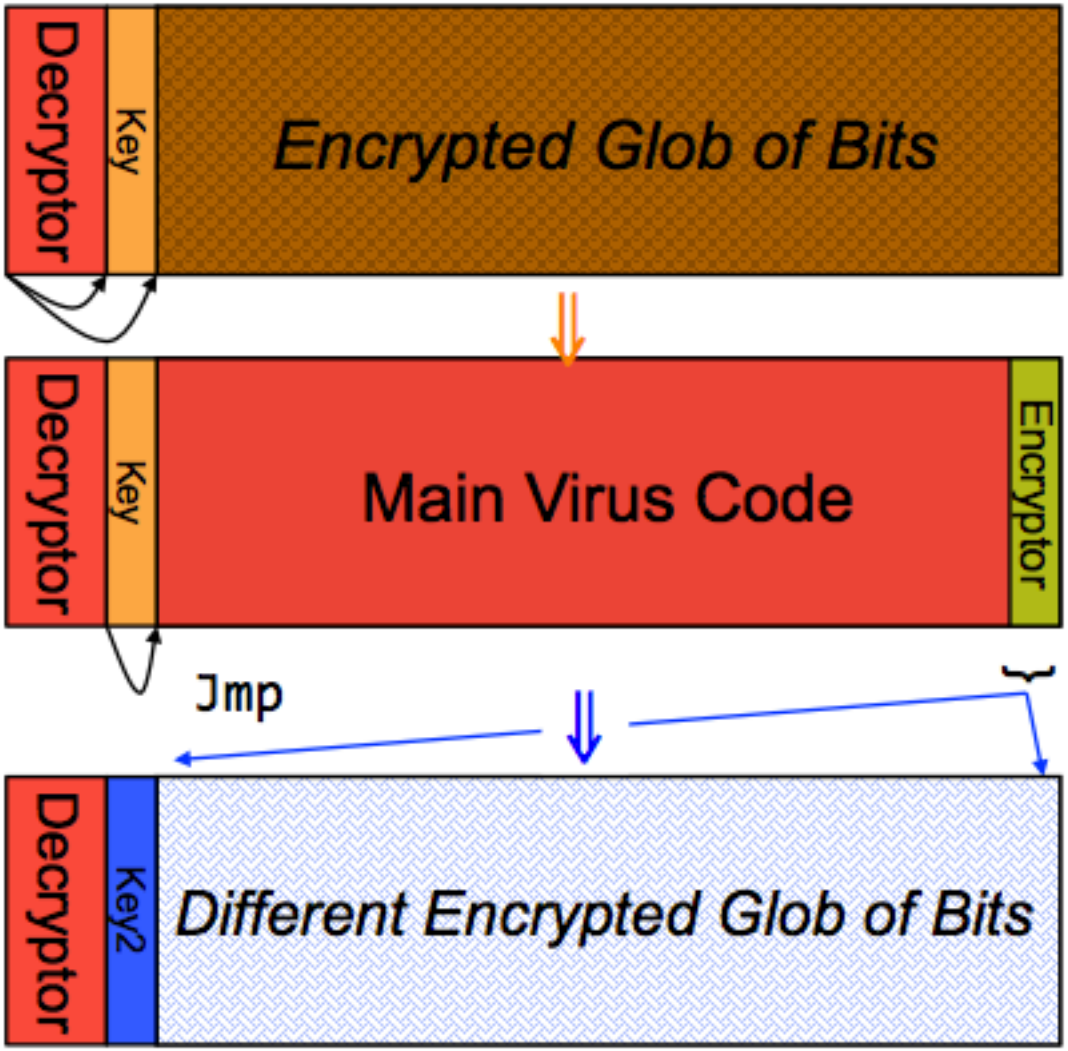
Instead of this …

Virus has *this* initial structure

When executed, decryptor applies key to decrypt the glob …

… and jumps to the decrypted code once stored in memory

Polymorphic Propagation



Once running, virus uses an *encryptor* with a new key to propagate

New virus instance bears little resemblance to original

Arms Race: Polymorphic Code

- Given polymorphism, how might we then detect viruses?
- Idea #1: use narrow sig. that targets decryptor
  - Issues?
    - Less code to match against " more false positives
    - Virus writer spreads decryptor across existing code
- Idea #2: execute (or statically analyze) suspect code to see if it decrypts!
  - Issues?
    - Legitimate "*packers*" perform similar operations (decompression)
    - How long do you let the new code execute?
      - If decryptor only acts after lengthy legit execution, difficult to spot

Metamorphic Code

- Idea: every time the virus propagates, generate *semantically* different version of it!
  - Different semantics only at immediate level of execution; higher-level semantics remain same
- How could you do this?
- Include with the virus a code rewriter:
  - Inspects its own code, generates random variant, e.g.
  - Renumber registers
  - Change order of conditional code
  - Reorder operations not dependent on one another
  - Replace one low-level algorithm with another
  - Remove some do-nothing padding and replace with different do- nothing padding (“chaff”)

Detecting Metamorphic Viruses?

- Need to analyze execution behavior
  - Shift from syntax (*appearance* of instructions) to semantics (*effect* of instructions)
- Two stages: (1) AV company analyzes new virus to find behavioral signature; (2) AV software on end systems analyze suspect code to test for match to signature
- What countermeasures will the virus writer take?
  - Delay analysis by taking a long time to manifest behavior
    - Long time = await particular condition, or even simply clock time
  - Detect that execution occurs in an analyzed environment and if so behave differently
    - E.g., test whether running inside a debugger, or in a Virtual Machine
- Counter-countermeasure?
  - AV analysis looks for these tactics and skips over them
- Note: attacker has edge as *AV products supply an **oracle!***

Anomaly-Based HIDS

- Idea behind HIDS
  - Define normal behavior for a process
    - Create a model that captures the behavior of a program during normal execution.
  - Monitor the process
    - Raise a flag if the program behaves abnormally

Why System Calls? (Motivation)

- The program is a layer between user inputs and the operating system
- A compromised program cannot cause significant damage to the underlying system without using system calls
- i.e Creating a new process, accessing a file etc.

Model Creation Techniques

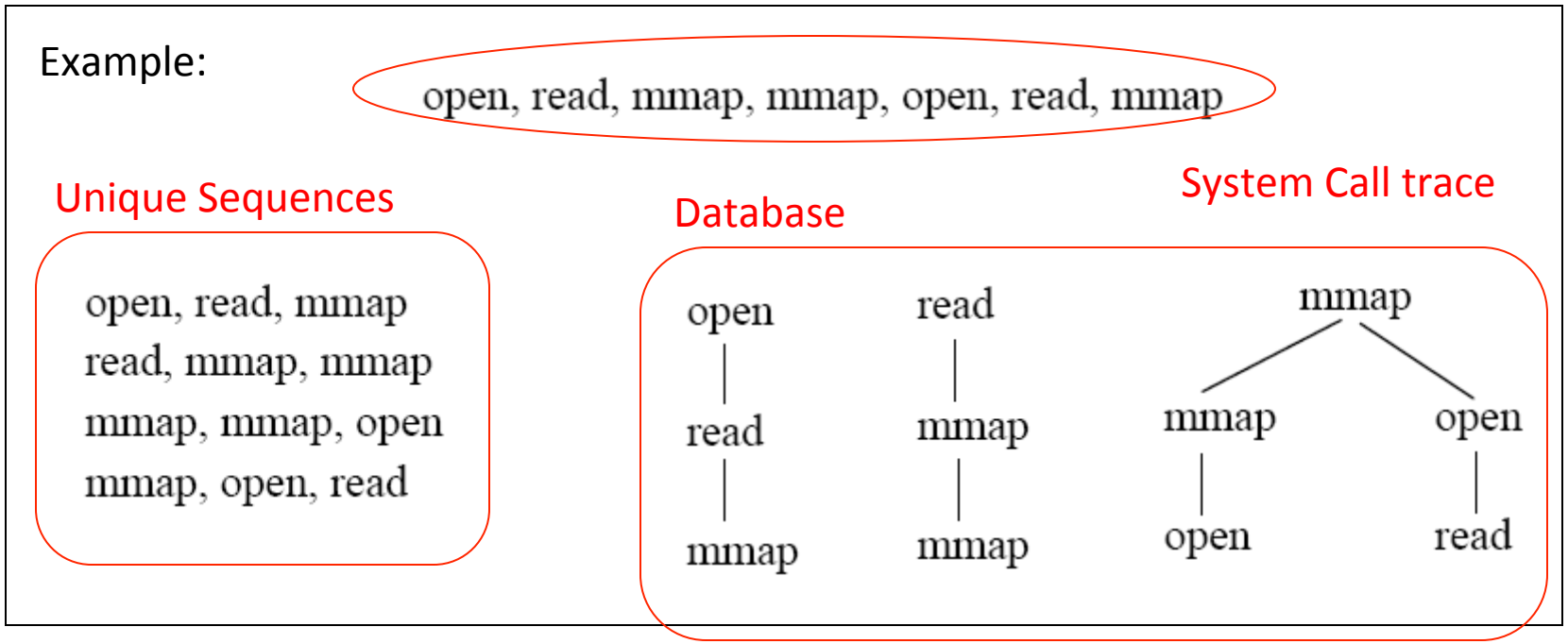
- Models are created using two different methods:
  - Training: The programs behavior is captured during a training period, in which, there is assumed to be no attacks. Another way is to craft synthetic inputs to simulate normal operation.
  - Static analysis: The information required by the model is extracted either from source code or binary code by means of static analysis.
- Training is easy, however, the model may miss some of the behavior and therefore produce false positives.

N-Gram

- Forrest et. al. A Sense of Self for Unix Processes, 1996.
- Tries to define a normal behavior for a process by using sequences of system calls.
- As the name of their paper implies, they show that fixed length short sequences of system calls are distinguishing among applications.
- For every application a model is constructed and at runtime the process is monitored for compliance with the model.
- *Definition:* The list of system calls issued by a program for the duration of it's execution is called a *system call trace*.

N-Gram: Building the Model by Training

- Slide a window of length N over a given system call trace and extract unique sequences of system calls.



N-Gram: Monitoring

- Monitoring
  - A window is slid across the system call trace as the program issues them, and the sequence is searched in the database.
  - If the sequence is in the database then the issued system call is valid.
  - If not, then the system call sequence is either an intrusion or a normal operation that was not observed during training (false positive) !!

Experimental Results for N-Gram

- Databases for different processes with different window sizes are constructed
- A normal sendmail system call trace obtained from a user session is tested against all processes databases.
- The table shows that sendmail's sequences are unique to sendmail and are considered as anomalous by other models.

	5		6		11	
Process	%	#	%	#	%	#
sendmail	0.0	0	0.0	0	0.0	0
ls	6.9	23	8.9	34	13.9	93
ls -l	30.0	239	32.1	304	38.0	640
ls -a	6.7	23	8.3	34	13.4	93
ps	1.2	35	8.3	282	13.0	804
ps -ux	0.8	45	8.1	564	12.9	1641
finger	4.6	21	4.9	27	5.7	54
ping	13.5	56	14.2	70	15.5	131
ftp	28.8	450	31.5	587	35.1	1182
pine	25.4	1522	27.6	1984	30.0	3931
httpd	4.3	310	4.8	436	4.7	824

The table shows the number of mismatched sequences and their percentage wrt the total number of subsequences in the user session

FINDING VULNERABILITIES

Testing

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

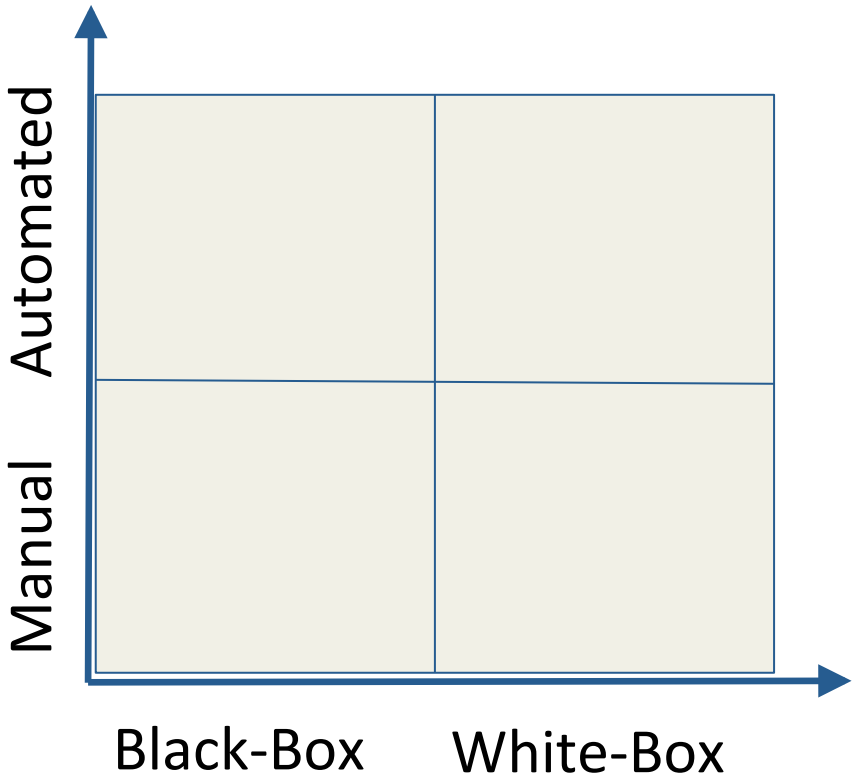
The Need for Specifications

- 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

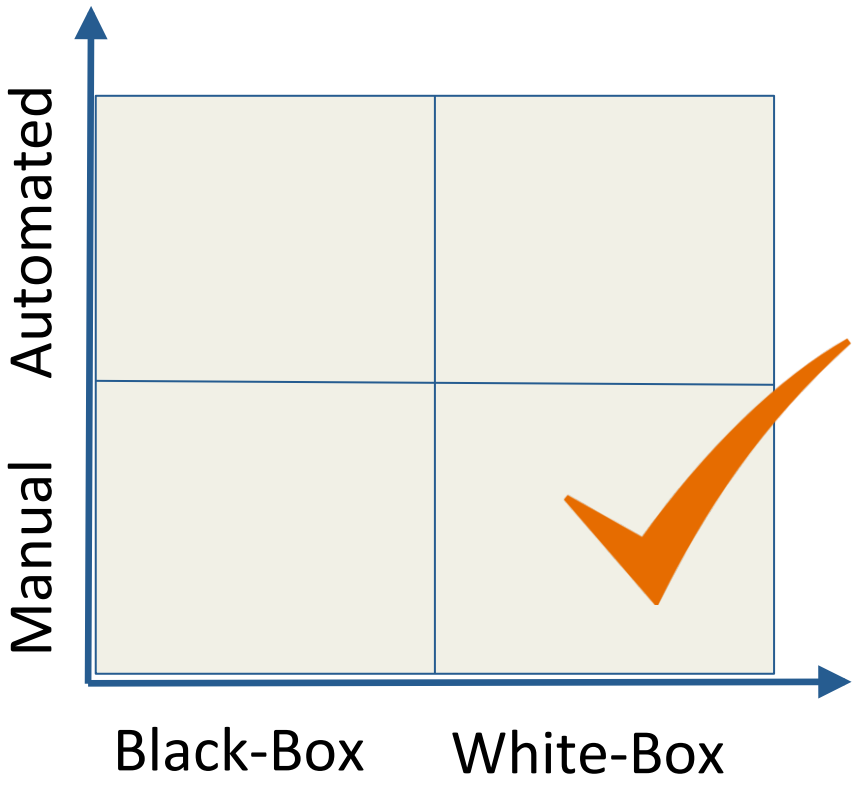
How Good Is Your Test Suite?

- 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

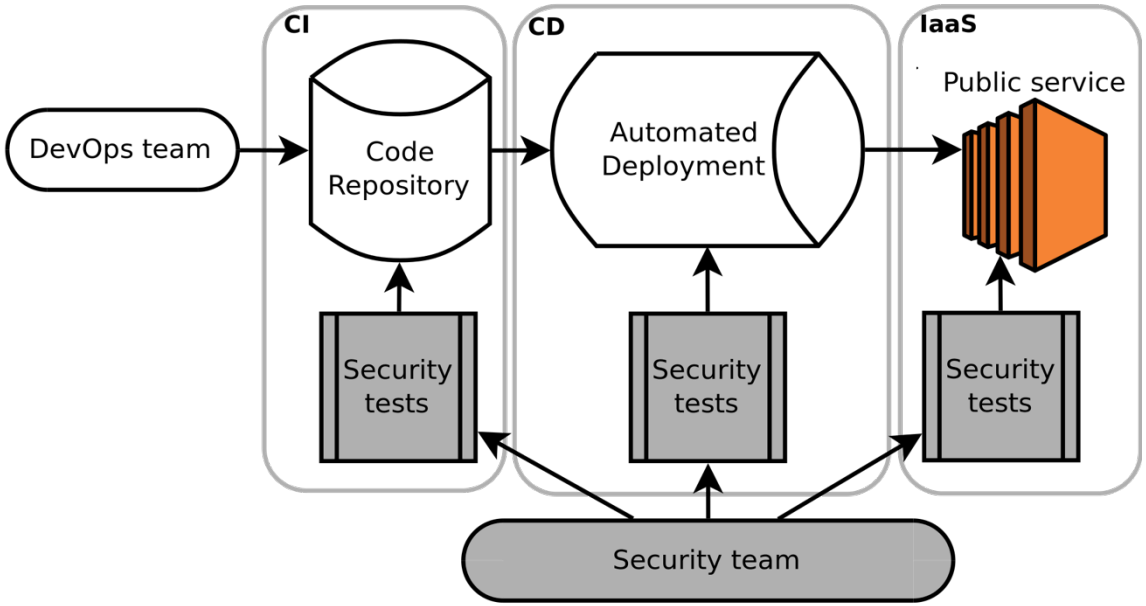
Code Coverage

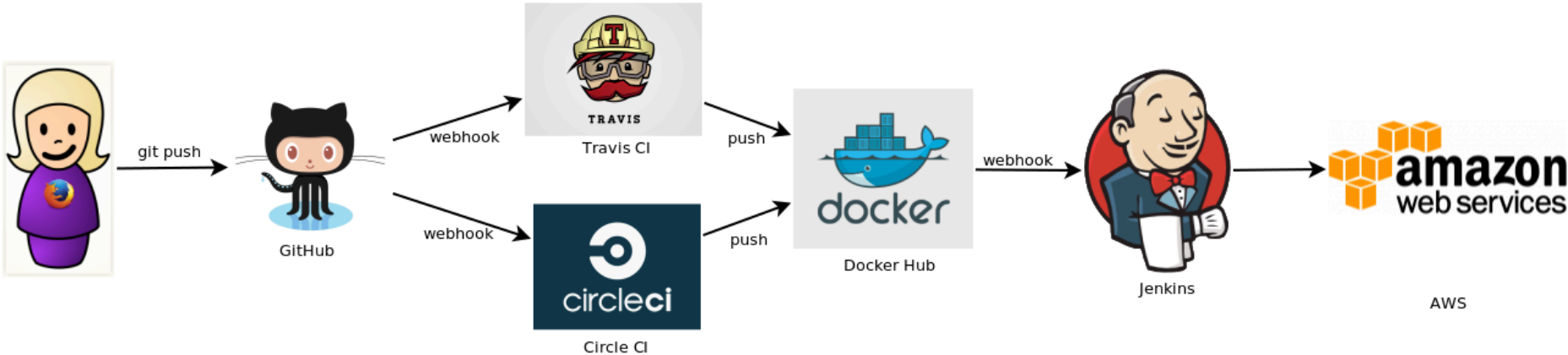
- Metric to quantify extent to which a program's code is tested by a given test suite
  - Function coverage: which functions were called?
  - Statement coverage: which statements were executed?
  - Branch coverage: which branches were taken?
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., inaccessible code
  - Often required for safety-critical applications

Classification of Testing Approaches

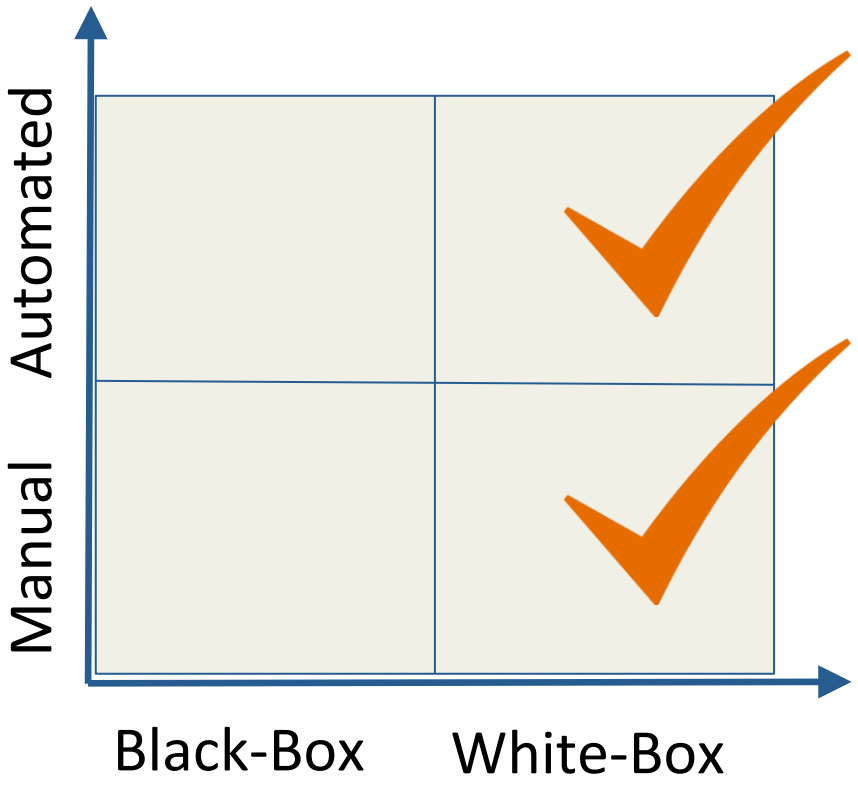


Test Driven Security

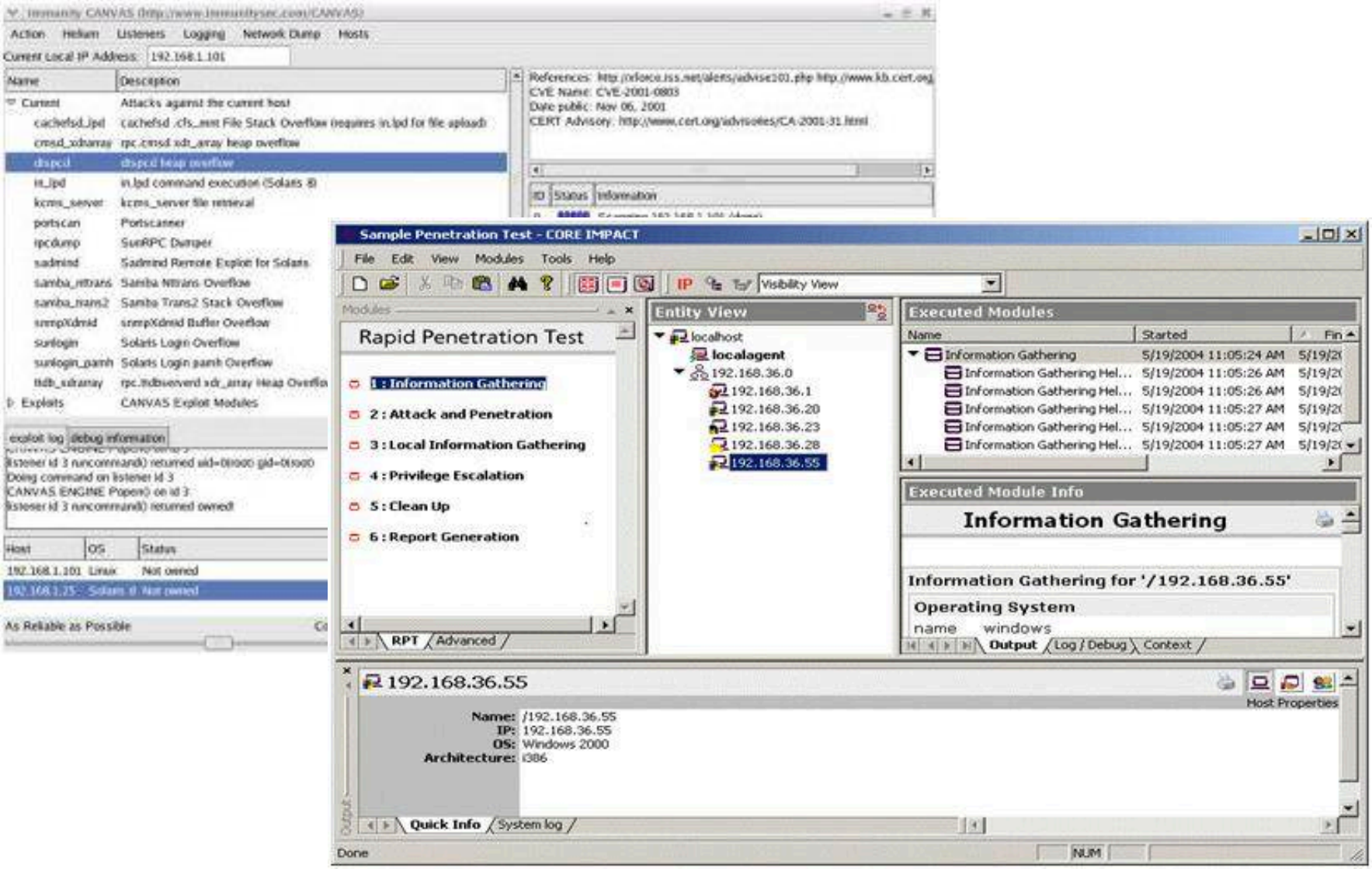




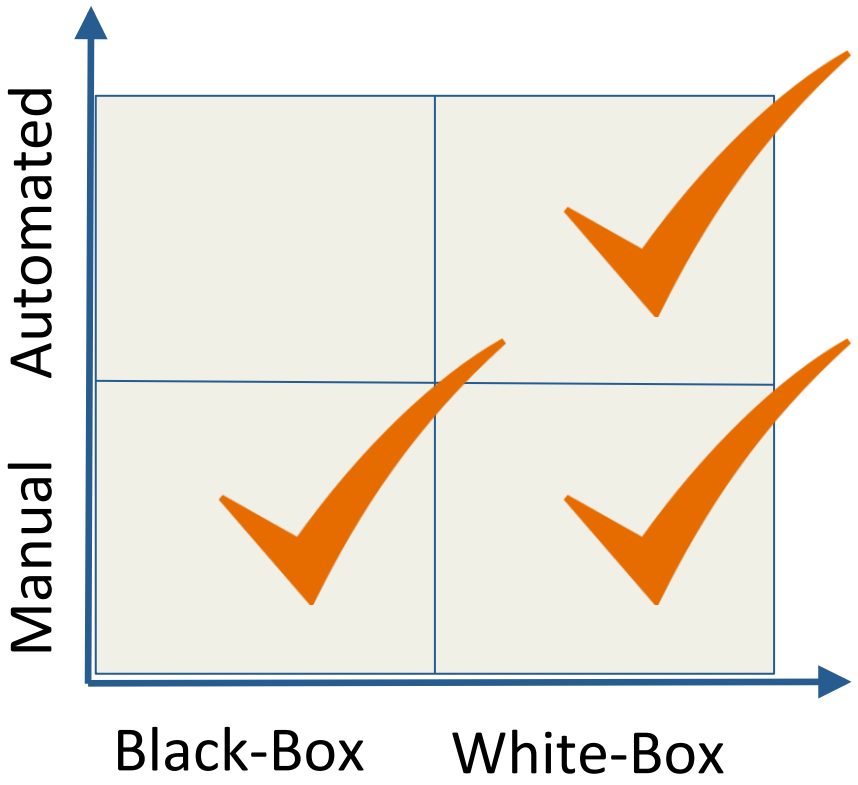
Classification of Testing Approaches



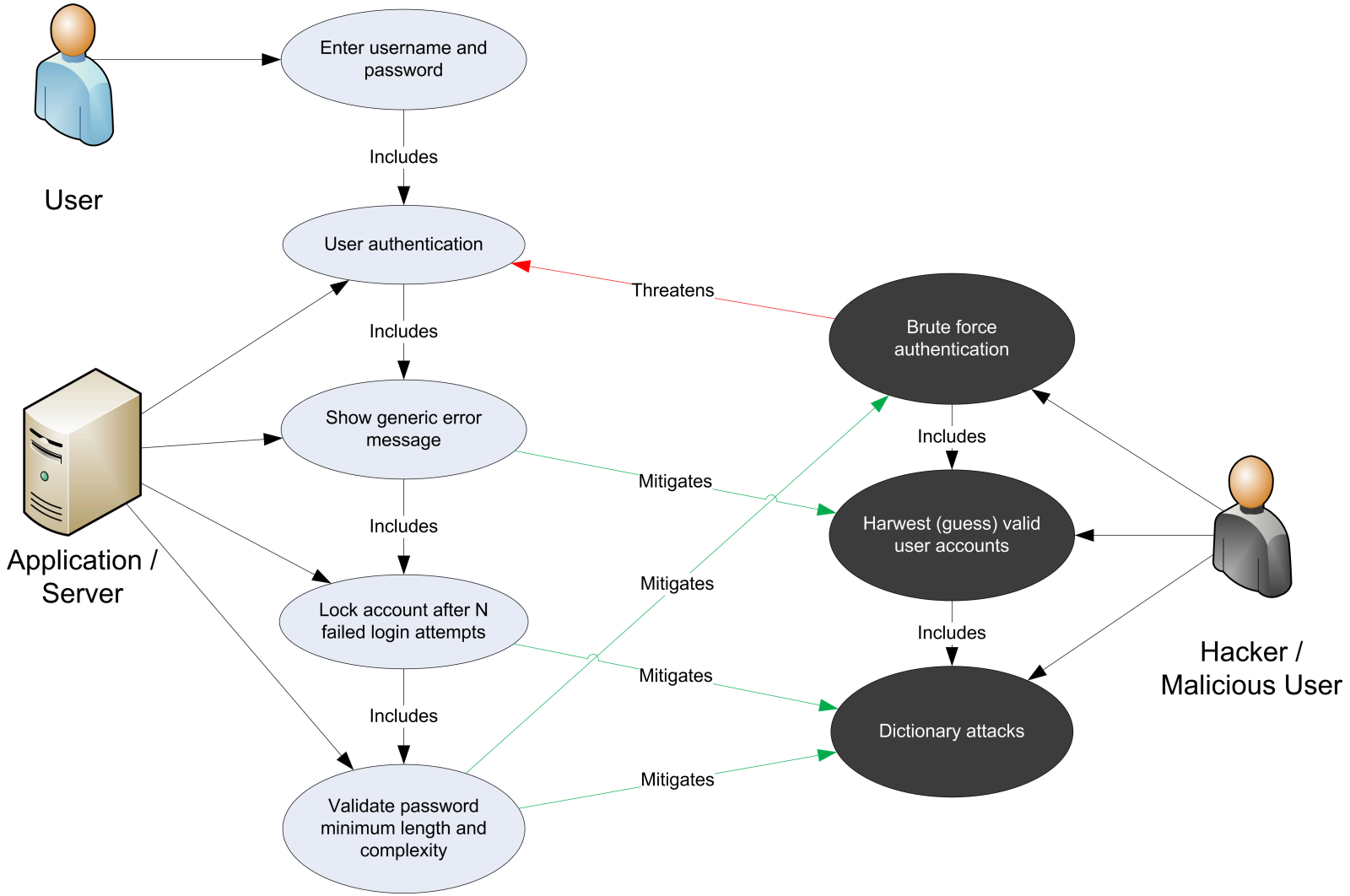
Automated White Box Testing



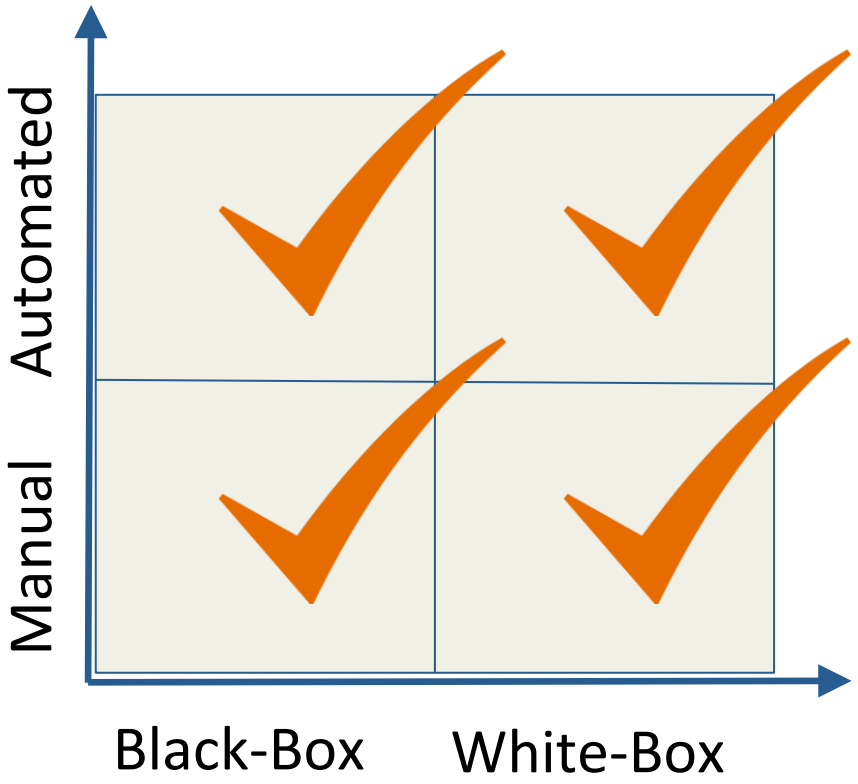
Classification of Testing Approaches



Web Pen Testing Simple Example



Classification of Testing Approaches



Fuzzing Components

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

Test Case Generation

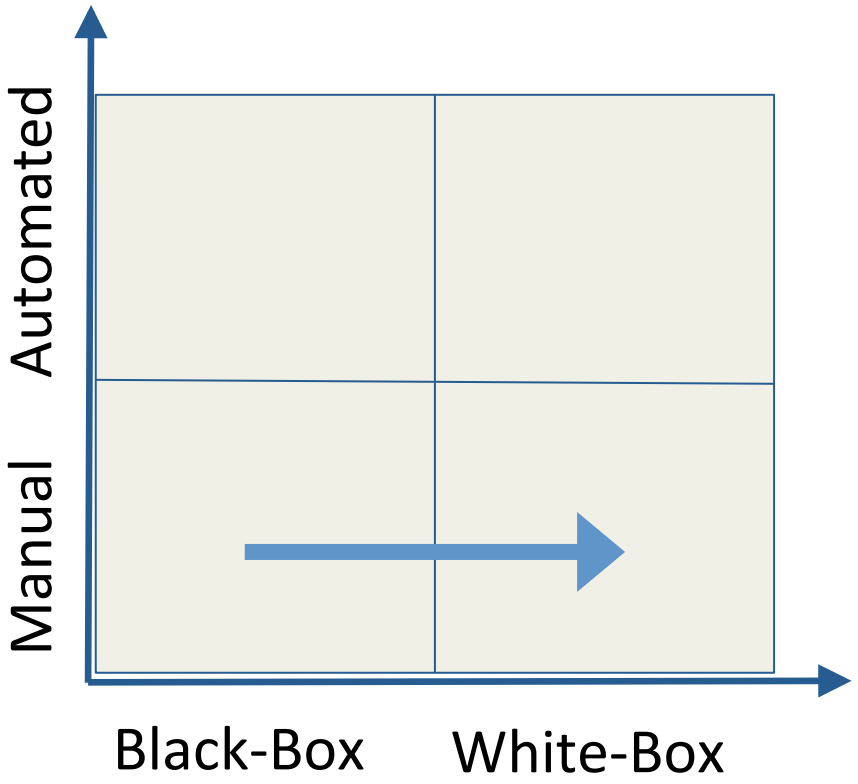
- Random Fuzzing
- “Dumb” (mutation-based) Fuzzing
  - Mutate an existing input
- “Smart” (generation-based) Fuzzing
  - Generate an input based on a model (grammar)

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

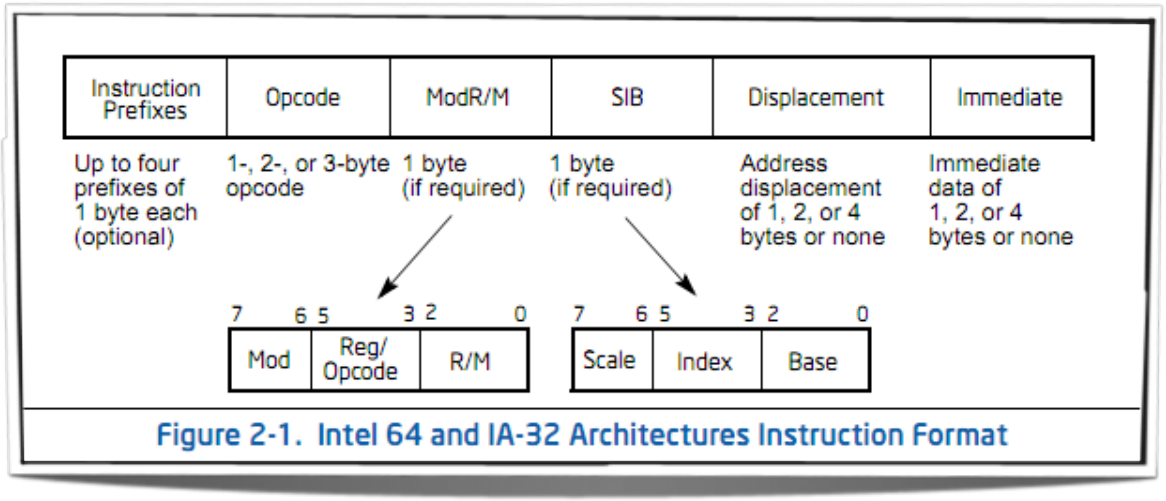


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

11101011 00000110

01010000

Bits

0xEB 0x06

0x50

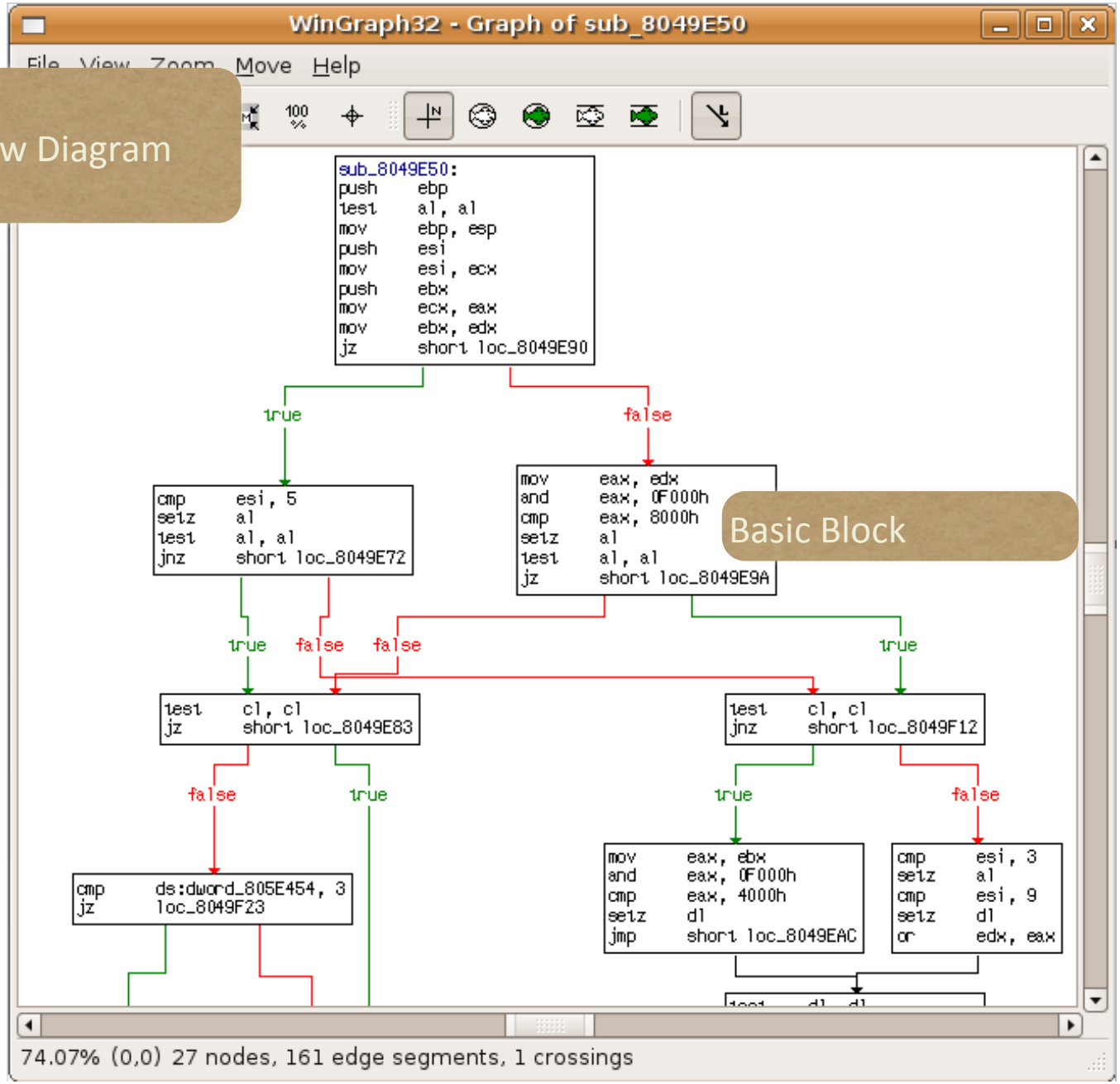
Hex Bytes

JMP +6

PUSH EAX

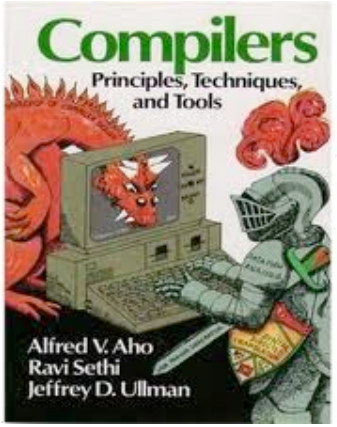
Instructions (human-readable)

Control Flow Diagram



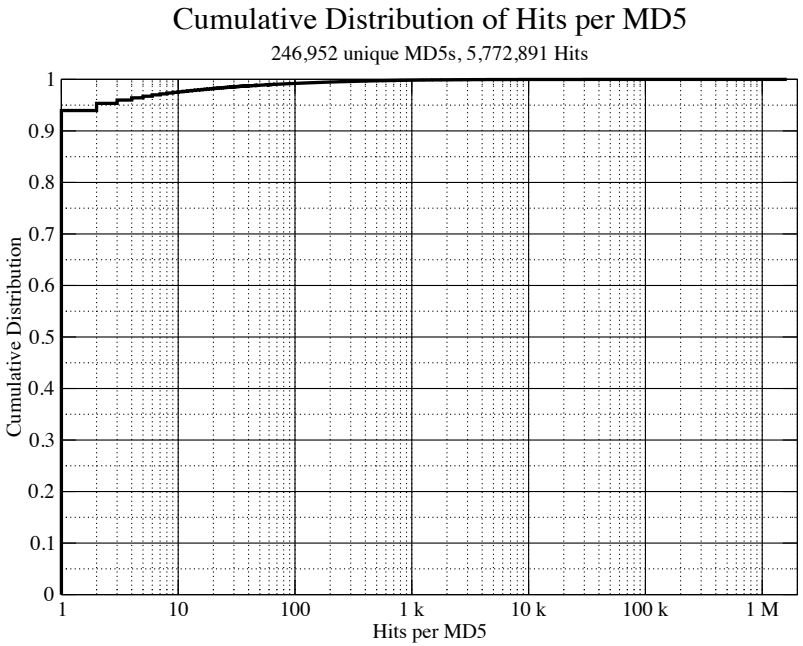
Difficulties

- 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

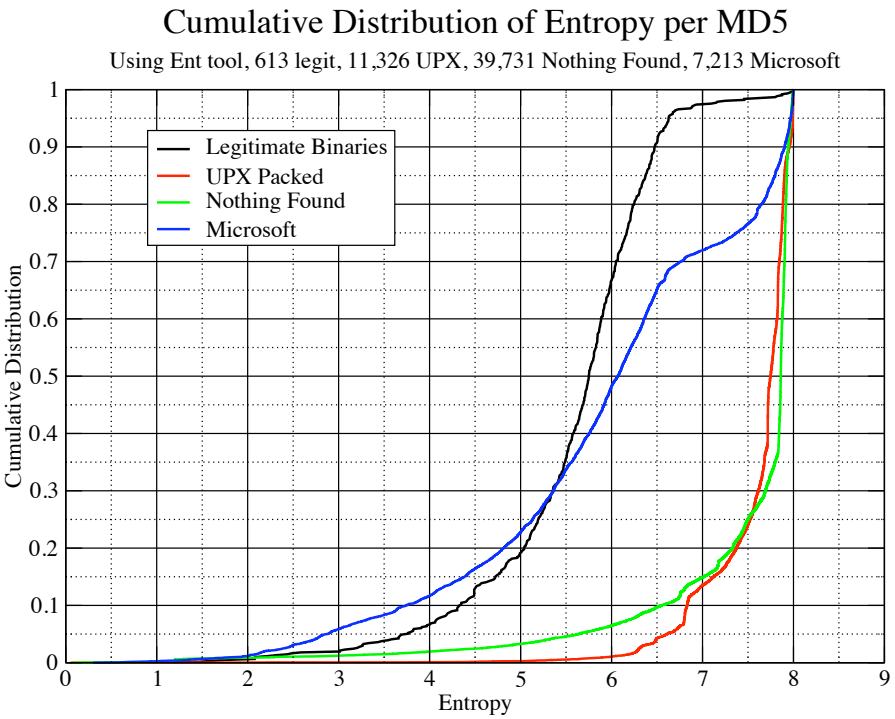
PEiD	Count
UPX	11244
Upack	6079
PECompact	4672
Nullsoft	2295
Themida	1688
FSG	1633
tElock	1398
NsPack	1375
ASpack	1283
WinUpack	1234

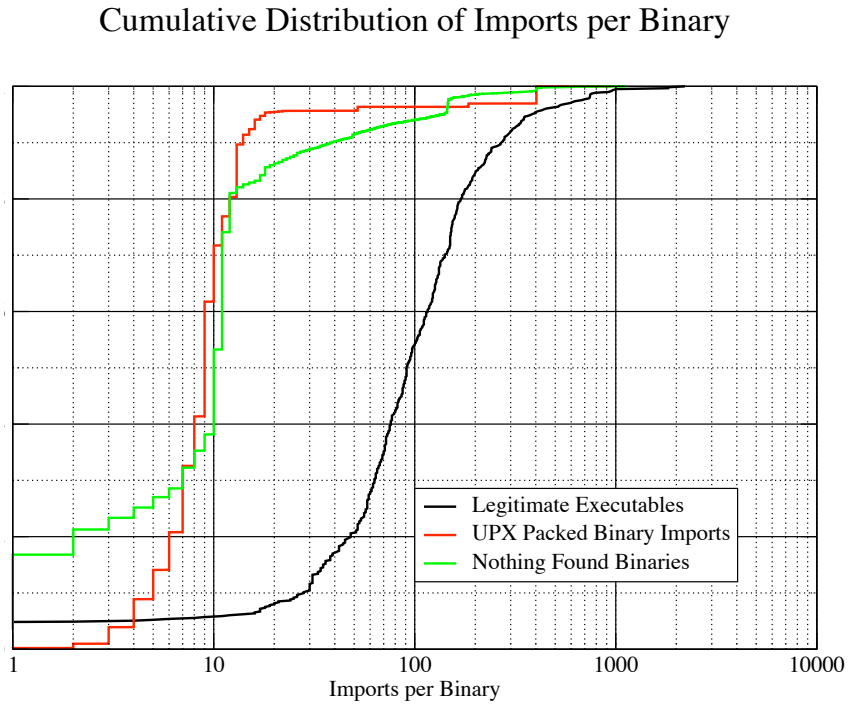
Identified: 59,070 (60%)
Top 10: 33.3%

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%

How about the unidentified?



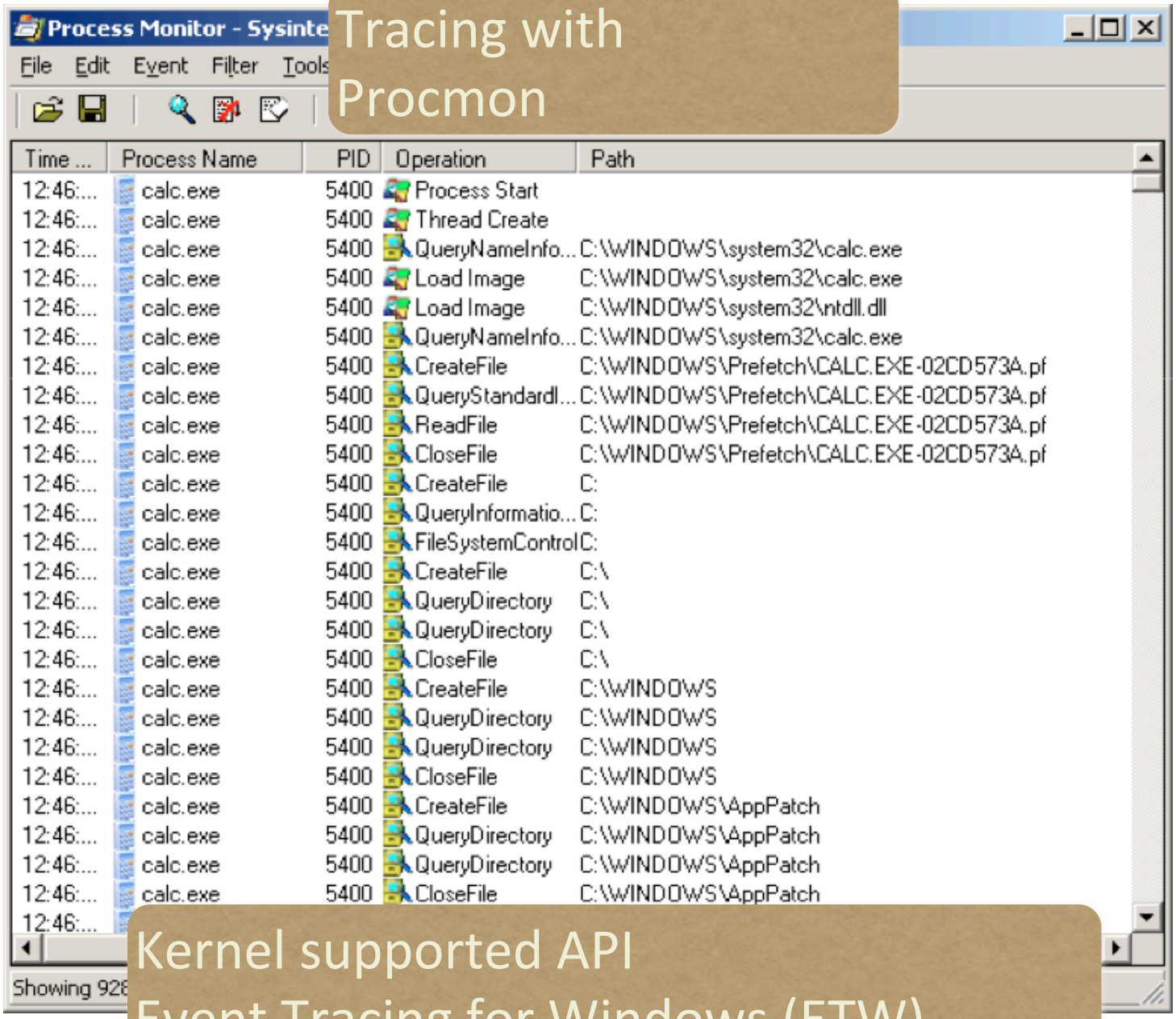


- Unidentified have: high entropy, small IATs
- Overall: > 90% packed

Dynamic Analysis

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

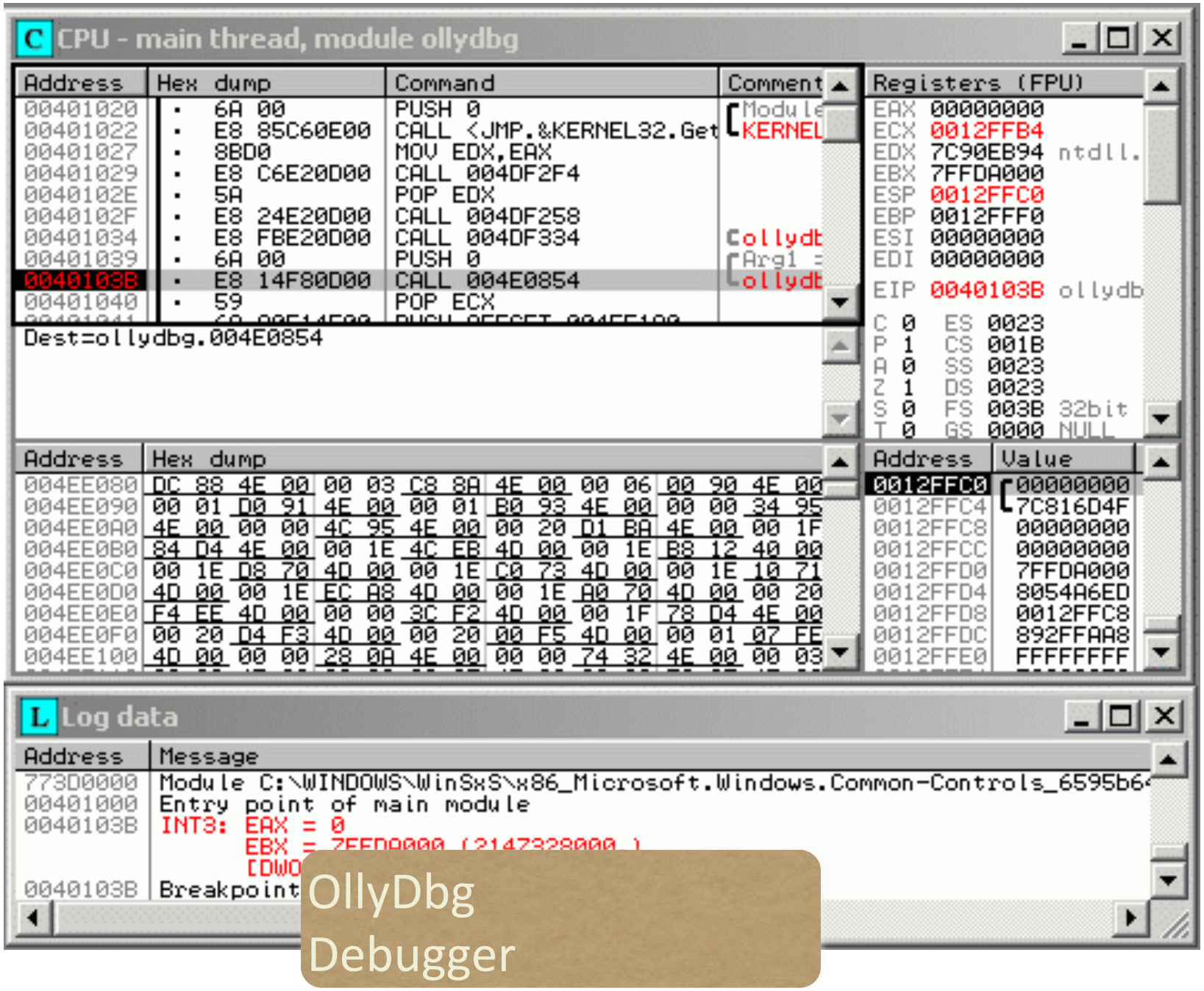
Tracing with Procmon



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


CPU - main thread, module ollydbg
Address
Hex dump
Command
Comment
00401020 · 6A 00 PUSH 0
00401022 · E8 85C60E00 CALL <JMP.&KERNEL32.Get
00401027 · 8BD0 MOV EDX,EAX
00401029 · E8 C6E20D00 CALL 004DF2F4
0040102E · 5A POP EDX
0040102F · E8 24E20D00 CALL 004DF258
00401034 · E8 FBE20D00 CALL 004DF334
00401039 · 6A 00 PUSH 0
0040103B · E8 14F80D00 CALL 004E0854
00401040 · 59 POP ECX
Dest=ollydbg.004E0854
Registers (FPU)
EAX 00000000
ECX 0012FFB4
EDX 7C90EB94 ntdll.
EBX 7FFDA000
ESP 0012FFC0
EBP 0012FFF0
ESI 00000000
EDI 00000000
EIP 0040103B ollydb
C 0 ES 0023
P 1 CS 001B
A 0 SS 0023
Z 1 DS 0023
S 0 FS 003B 32bit
T 0 GS 0000 NULL
Address
Hex dump
004EE080 DC 88 4E 00 00 03 C8 8A 4E 00 00 06 00 90 4E 00
004EE090 00 01 D0 91 4E 00 00 01 B0 93 4E 00 00 00 34 95
004EE0A0 4E 00 00 00 4C 95 4E 00 00 20 D1 BA 4E 00 00 1F
004EE0B0 84 D4 4E 00 00 1E 4C EB 4D 00 00 1E B8 12 40 00
004EE0C0 00 1E D8 70 4D 00 00 1E C0 73 4D 00 00 1E 10 71
004EE0D0 4D 00 00 1E EC A8 4D 00 00 1E A0 70 4D 00 00 20
004EE0E0 F4 EE 4D 00 00 00 3C F2 4D 00 00 1F 78 D4 4E 00
004EE0F0 00 20 D4 F3 4D 00 00 20 00 F5 4D 00 00 01 07 FE
004EE100 4D 00 00 00 28 0A 4E 00 00 00 74 32 4E 00 00 03
Address
Value
0012FFC0 00000000
0012FFC4 7C816D4F
0012FFC8 00000000
0012FFCC 00000000
0012FFD0 7FFDA000
0012FFD4 8054A6ED
0012FFD8 0012FFC8
0012FFDC 892FFAA8
0012FFE0 FFFFFFFF
Log data
Address
Message
773D0000 Module C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b6
00401000 Entry point of main module
0040103B INT3: EAX = 0
EBX = 7FFDA000 (2147328000.)
[DWO
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 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

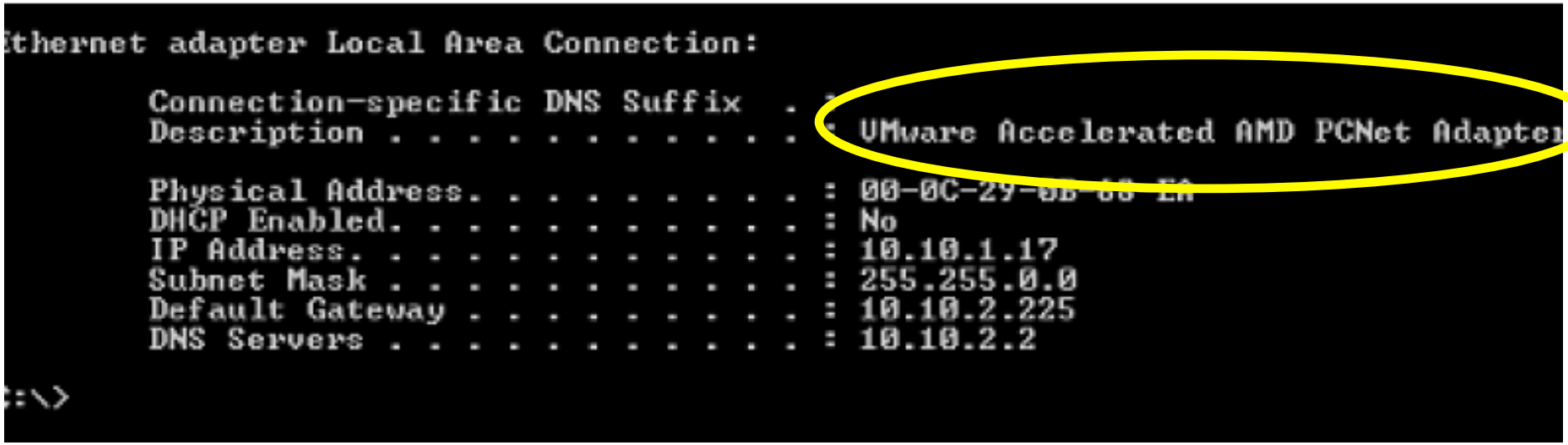
Easier

Harder

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

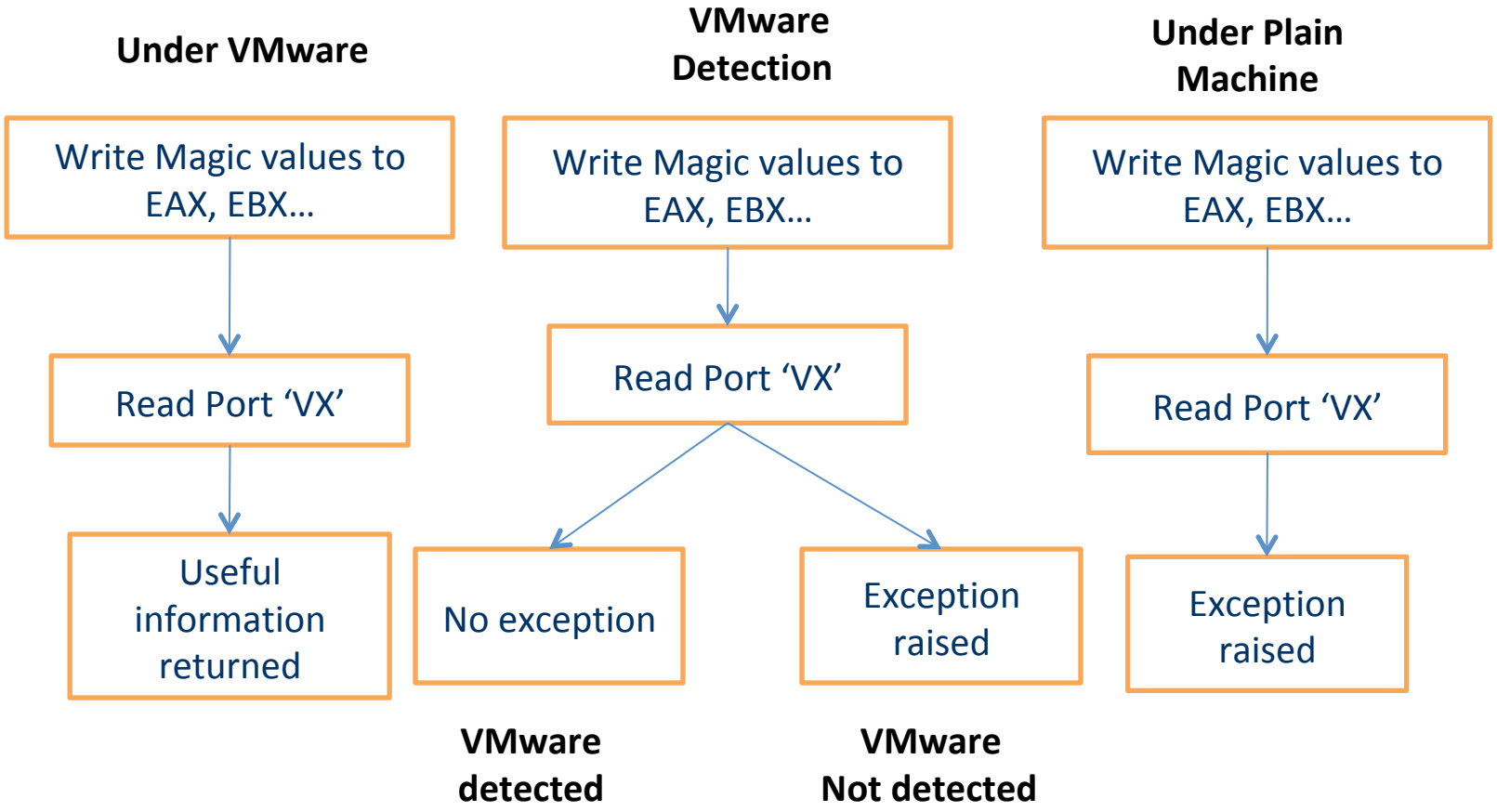
Example 1

- Device driver strings
  - Network cards



Example 2

- VMWare CommChannel (hooks)



Prevalence of evasion

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