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

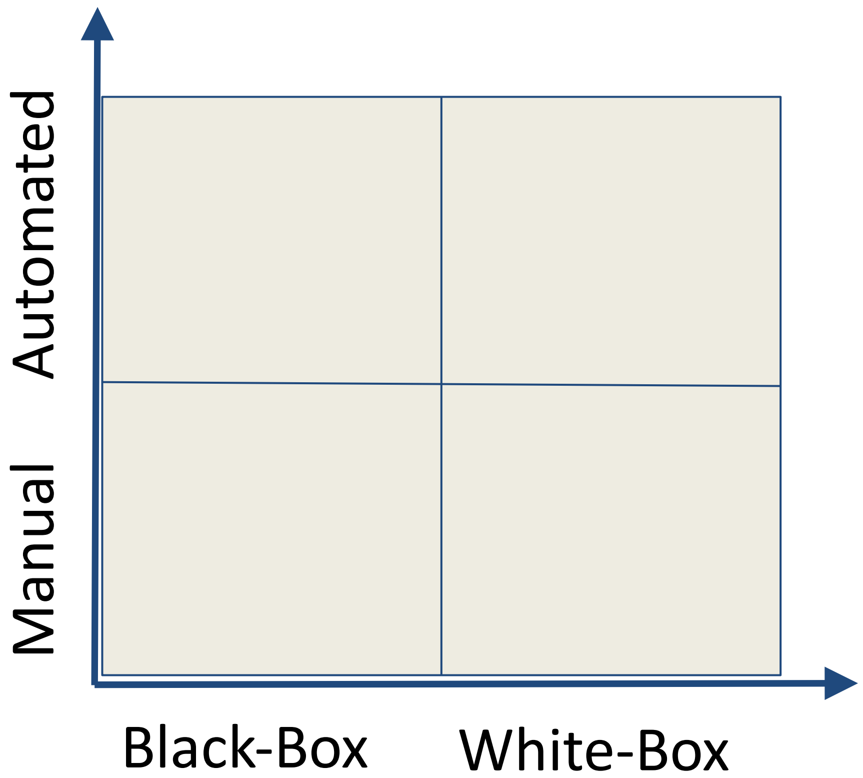
The Need for Specifications

- 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



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

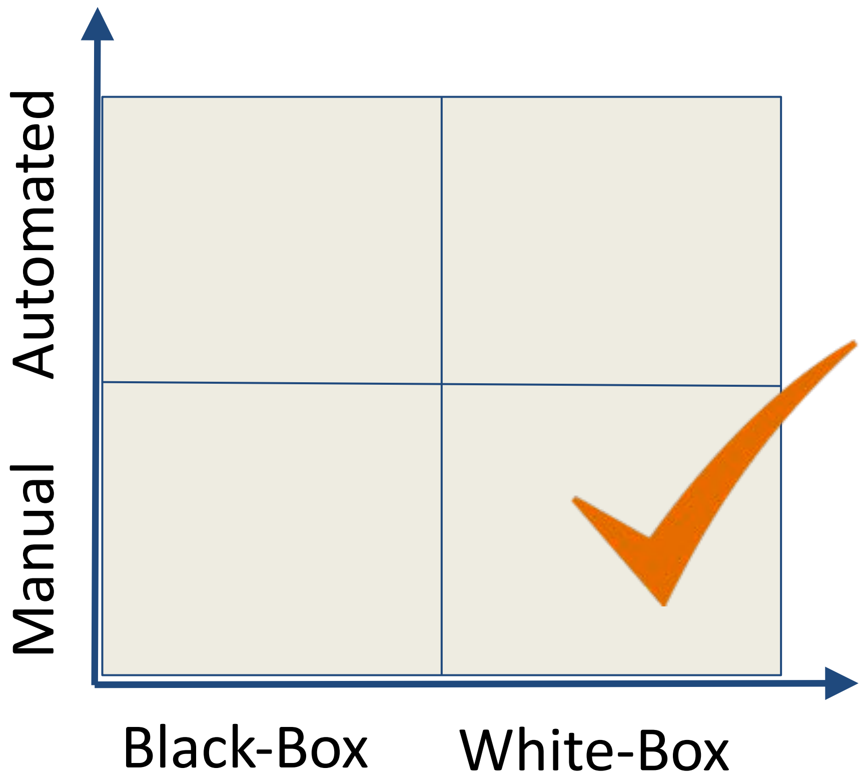
How Good Is Your Test Suite?

- 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

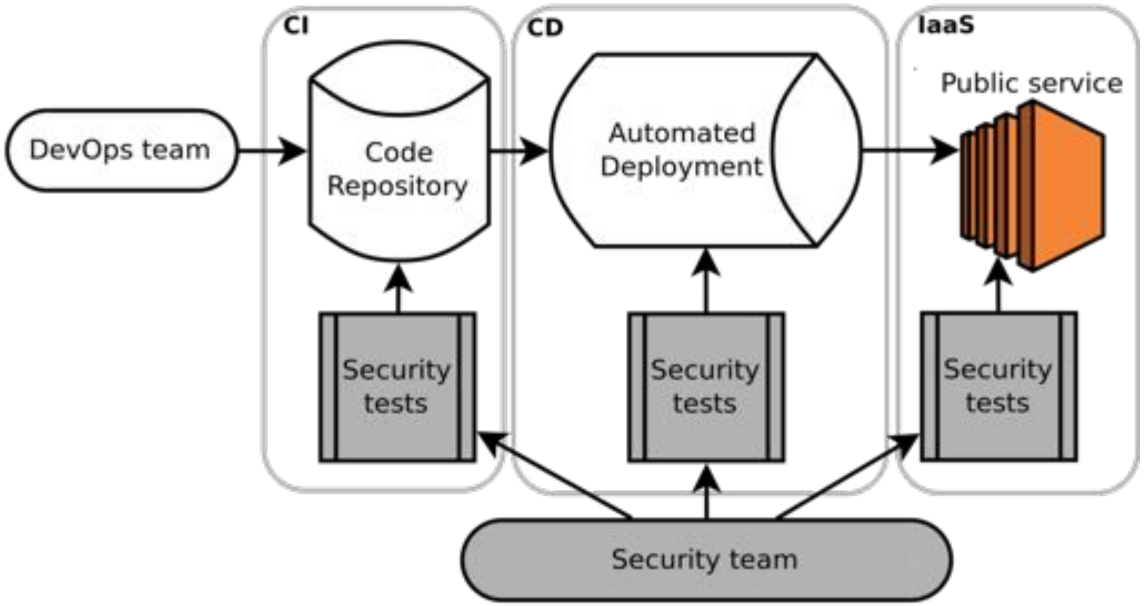
Code Coverage

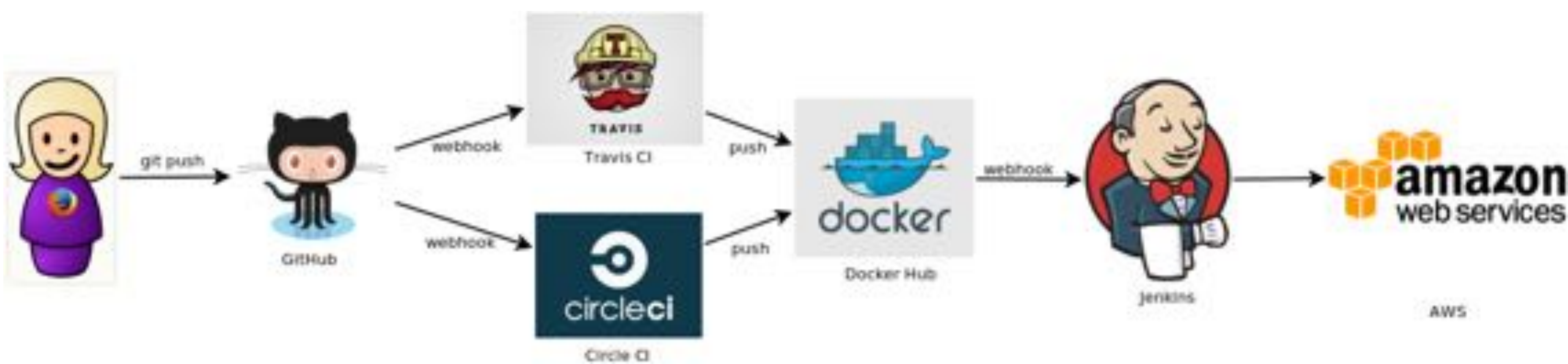
- Metric to quantify extent to which a program's code
  is tested by a given test suite
  - Function coverage: which functions were called?
  - Statement coverage: which statements were executed?
  - Branch coverage: which branches were taken?
- Given as percentage of some aspect of the program executed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

Classification of Testing Approaches

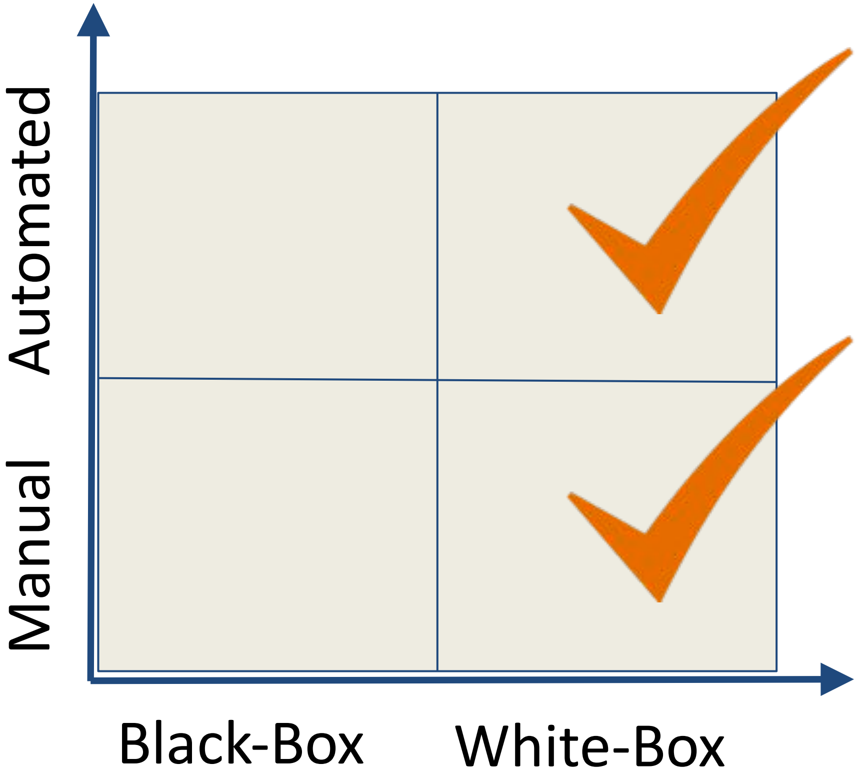


Test Driven Security

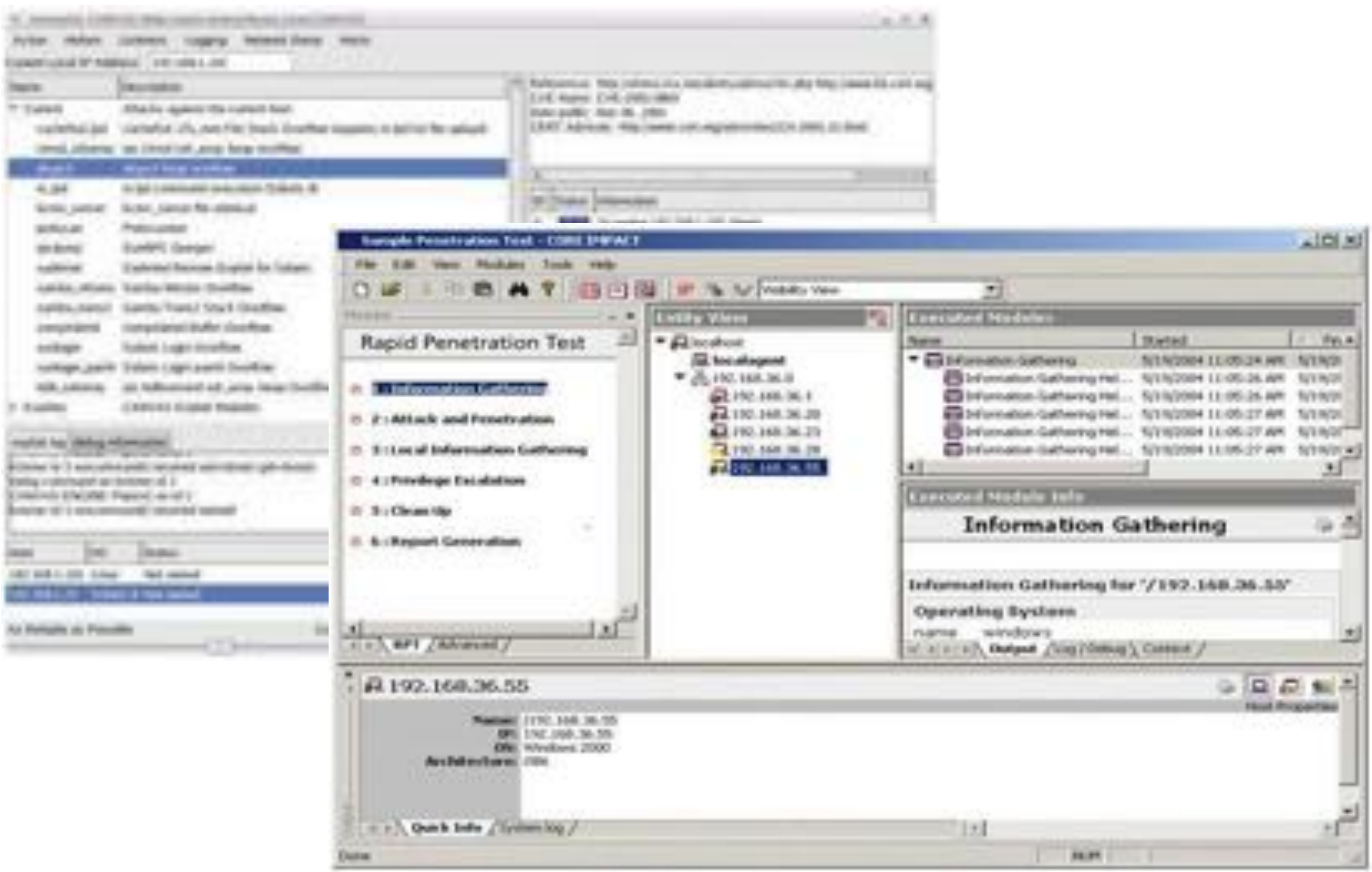




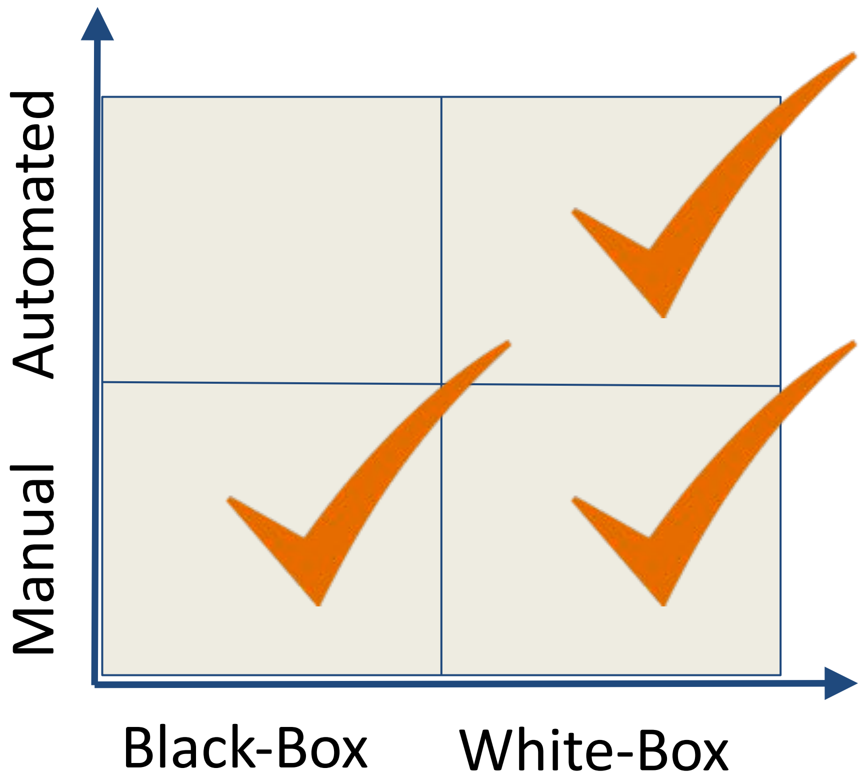
Classification of Testing Approaches



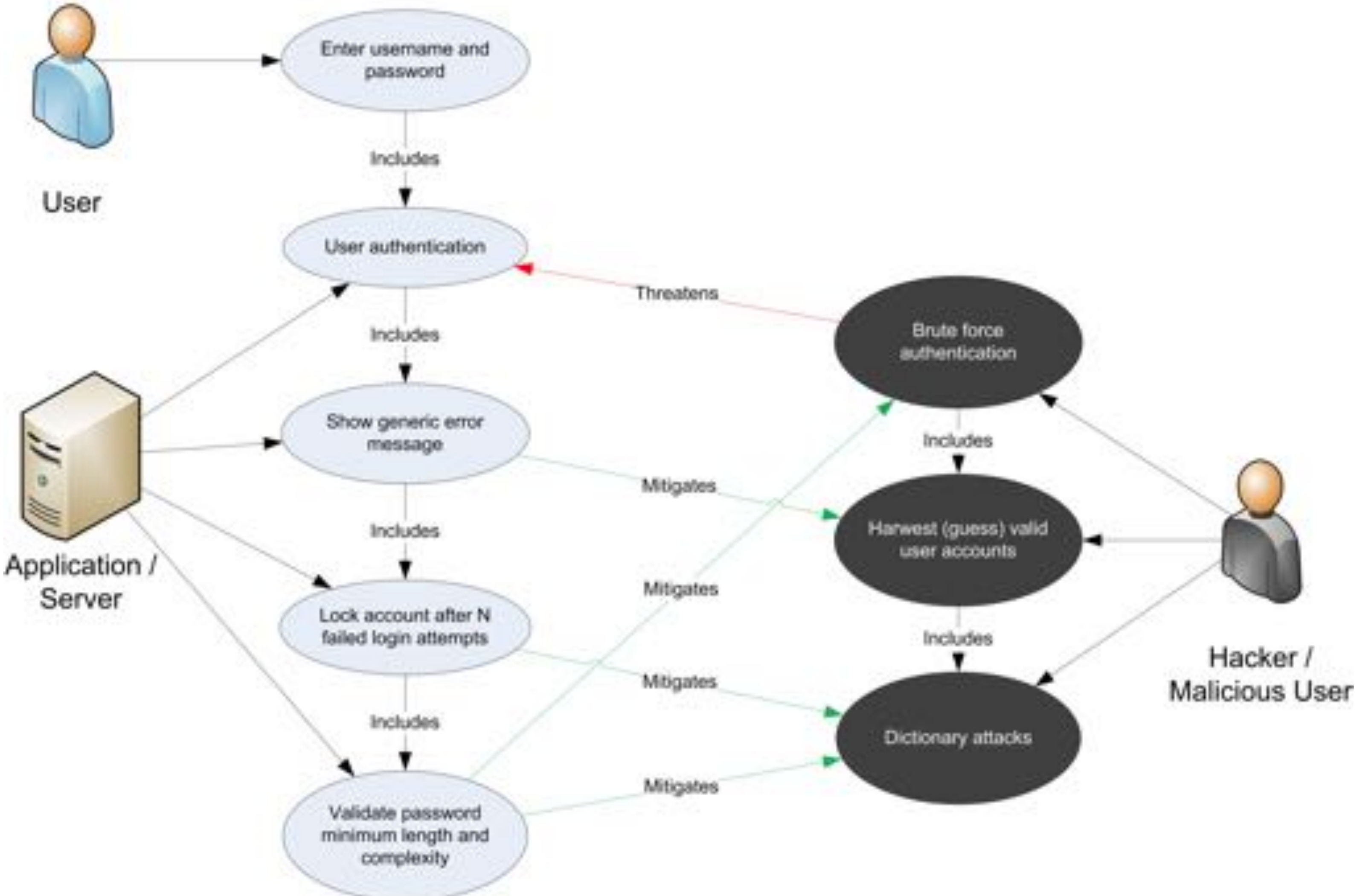
Automated White Box Testing



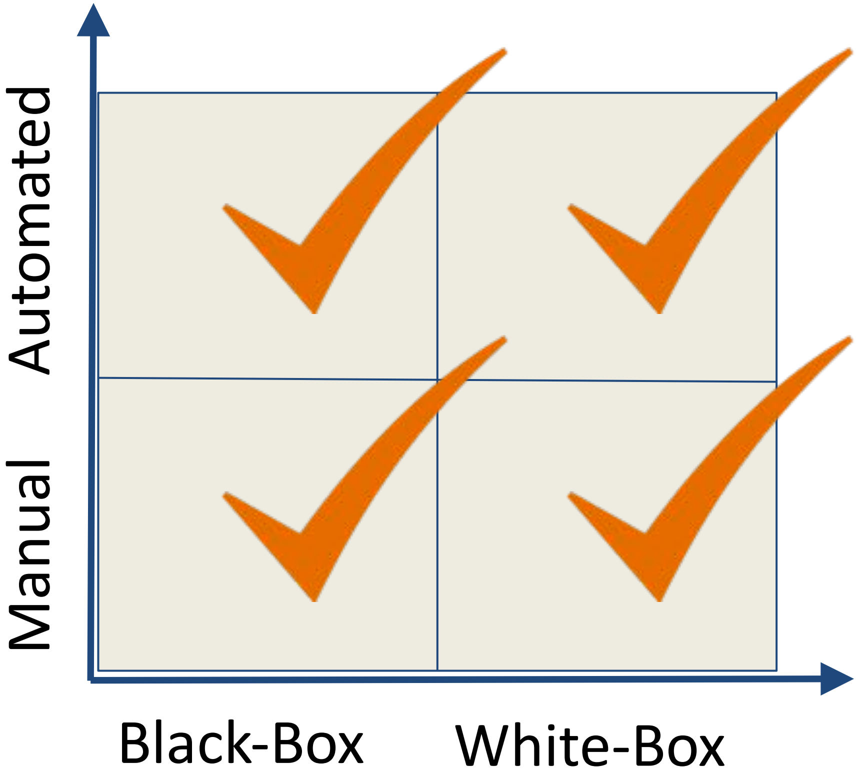
Classification of Testing Approaches



Web Pen Testing Simple Example



Classification of Testing Approaches



Fuzzing Components

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

Test Case Generation

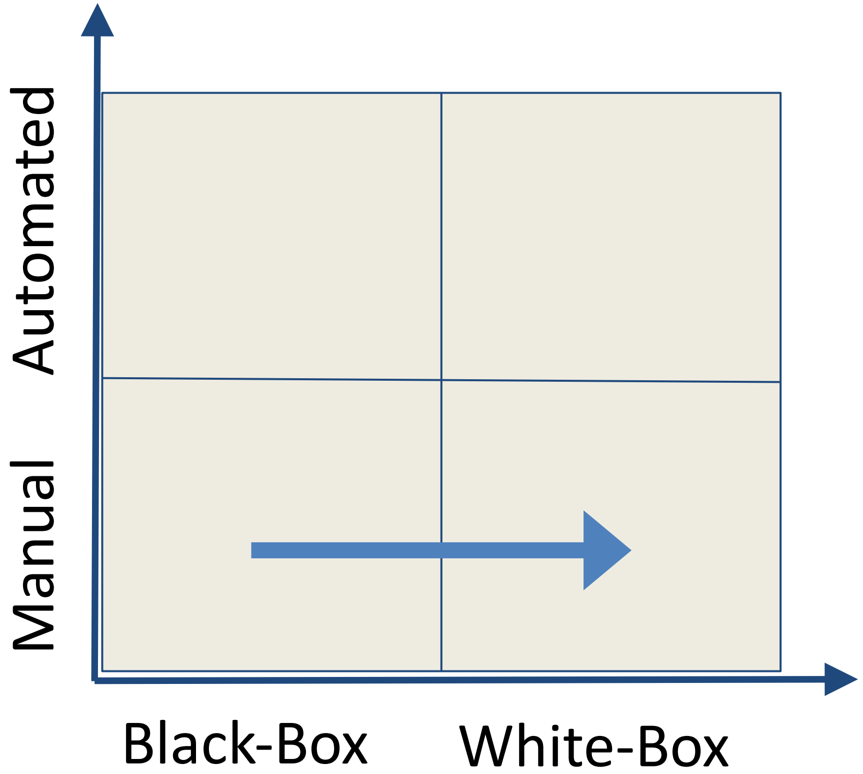
- Random Fuzzing
- “Dumb” (mutation-based) Fuzzing
  - Mutate an existing input
- “Smart” (generation-based) Fuzzing
  - Generate an input based on a model (grammar)

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
- 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

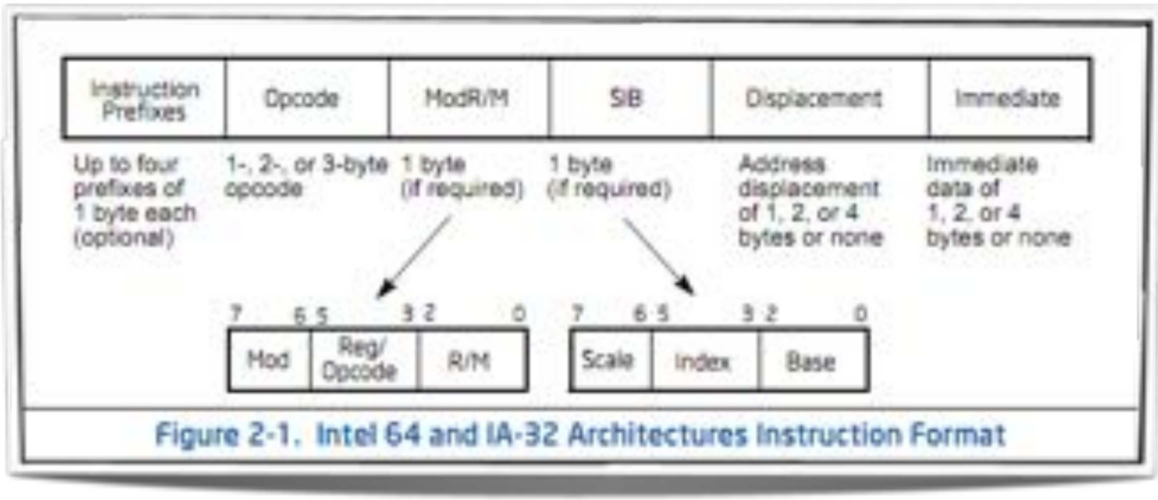
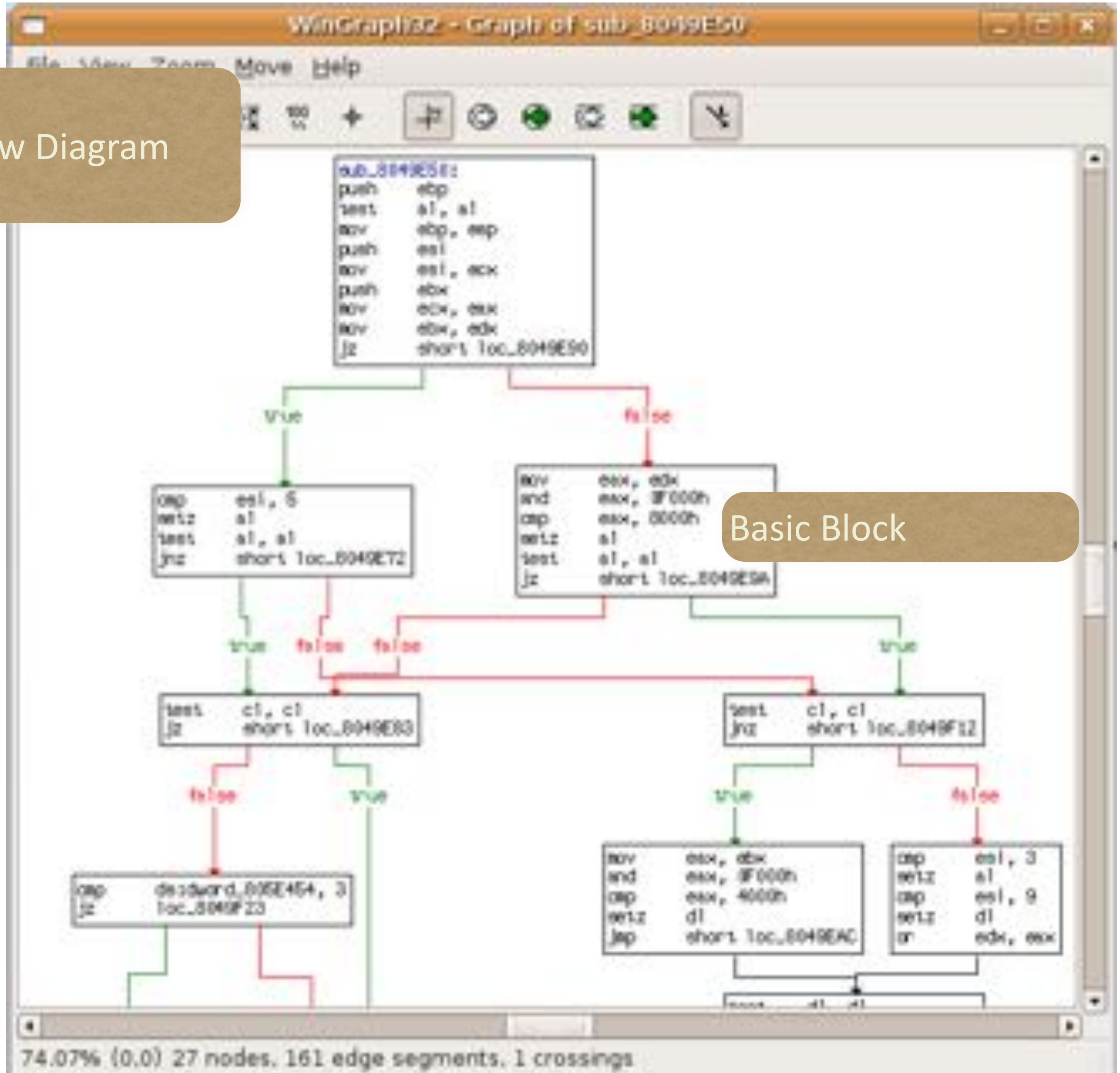


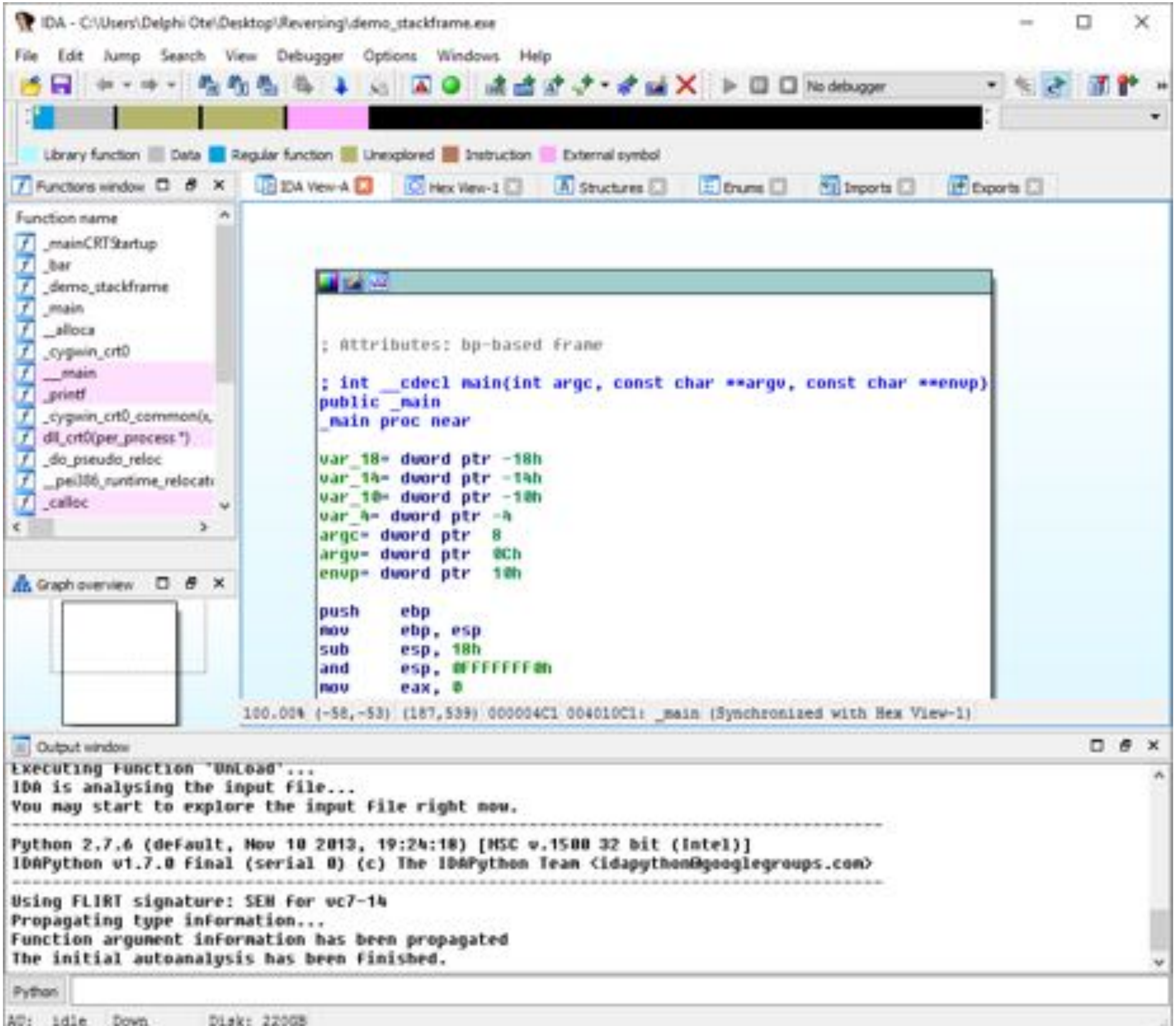
Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

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

Control Flow Diagram

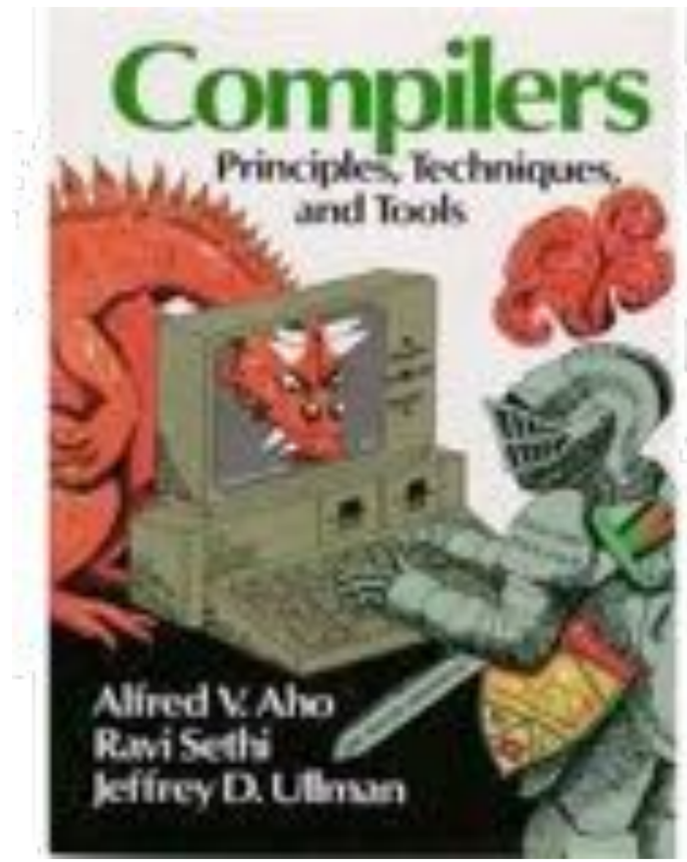
Basic Block




IDA - C:\Users\Delphi Ote\Desktop\Reversing\demo_stackframe.exe
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
Function name
_mainCRTStartup
_bar
_demo_stackframe
_main
__alloca
_cygwin_crt0
__main
_printf
_cygwin_crt0_common(x,
dll_crt0(per_process *)
_do_pseudo_reloc
__pei386_runtime_relocat
_calloc
; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
push ebp
mov ebp, esp
sub esp, 18h
and esp, 0FFFFFFF0h
mov eax, 0
100.00% (-58,-53) (187,539) 000004C1 004010C1: _main (Synchronized with Hex View-1)
Graph overview
Output window
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
Using FLIRT signature: SEH for vc7-14
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
Python
AU: idle Down Disk: 220GB

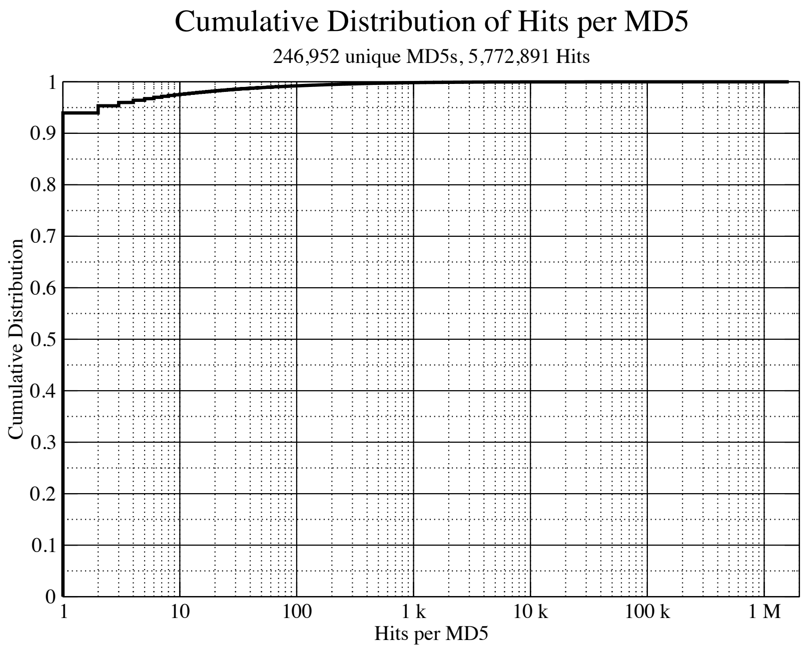
Difficulties

- 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

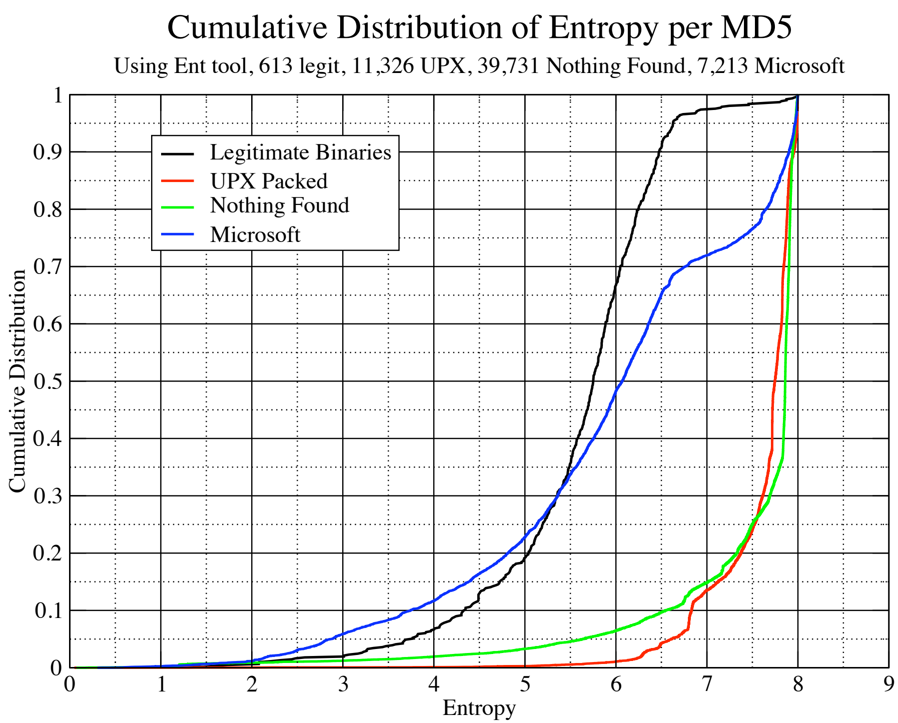
PEiD	Count
UPX	11244
Upack	6079
PECompact	4672
Nullsoft	2295
Themida	1688
FSG	1633
tElock	1398
NsPack	1375
ASpack	1283
WinUpack	1234

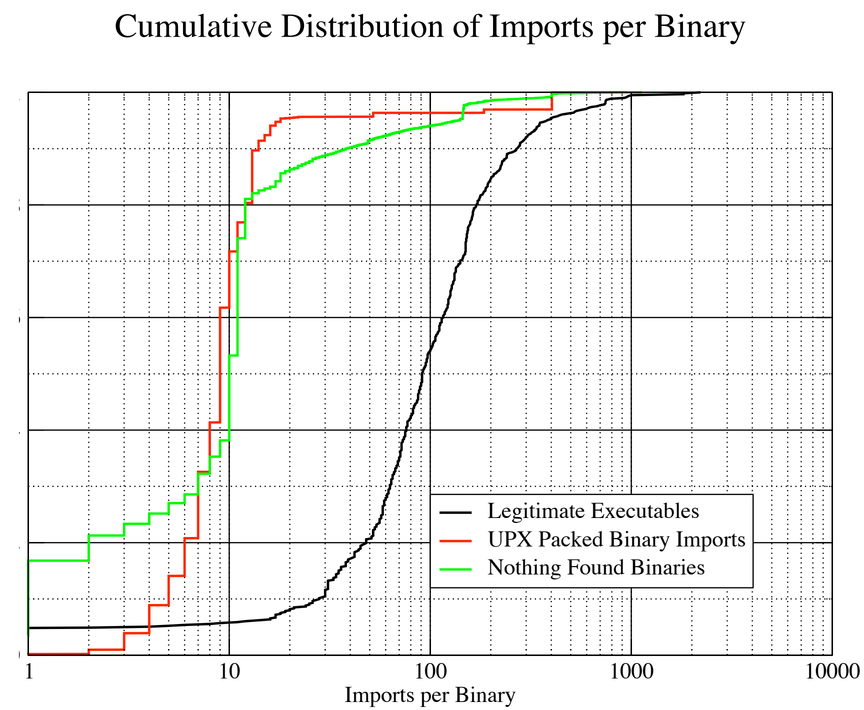
Identified: 59,070 (60%)
Top 10: 33.3%

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

Anti-disassembly

- Attackers use clever tricks to confuse the disassembler

```
55 8B EC 53 56 57 83 7D  08 02 75 52 33 C0 74 01
E8 8B 45 0C 8B 48 04 0F  BE 11 83 FA 70 75 3F 33
C0 74 01 E8 8B 45 0C 8B  48 04 0F BE 51 02 83 FA
71 75 2B 33 C0 74 01 E8  8B 45 0C 8B 48 04 0F BE
```

$$\overbrace{\texttt{75 52}}^{\text{JNZ}}\ \overbrace{\texttt{33 C0}}^{\text{XOR}}\ \overbrace{\texttt{74 01}}^{\text{JZ}}\ \overbrace{\texttt{E8 8B 45 0C 8B}}^{\text{CALL}}\ \overbrace{\texttt{48}}^{\text{DEC}}\ \overbrace{\texttt{04 0F}}^{\text{ADD}}$$

```
.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
```

$$\overbrace{\texttt{75 52}}^{\texttt{JNZ}}\ \overbrace{\texttt{33 C0}}^{\texttt{XOR}}\ \overbrace{\texttt{74 01}}^{\texttt{JZ}}\ \texttt{E8}\ \overbrace{\texttt{8B 45 0C}}^{\texttt{MOV}}\ \overbrace{\texttt{8B 48 04}}^{\texttt{MOV}}\ \overbrace{\texttt{0F}}^{\texttt{MOVSX}}$$

```
.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).

```
C4 08 C6 00 00 EB FF C0  48 E8 F1 00 00 00 89 85
58 FD FE FF 68 00 00 A0  00 FF 15 28 20 40 00 83
C4 04 89 85 54 FD FE FF  8B 8D 68 FD FF FF 83 C1
08 89 8D 68 FD FF FF 6A  00 6A 00 6A 00 6A 00 8B
95 68 FD FF FF 52 8B 85  5C FD FE FF 50 FF 15 64
20 40 00 89 85 64 FD FF  FF 74 03 75 01 E8 8D 8D
FC FE FF FF 51 68 00 00  01 00 8B 95 54 FD FE FF
```

$$\overbrace{\texttt{C4 08}}^{\text{ADD}}\ \overbrace{\texttt{C6 00 00}}^{\text{MOV}}\ \overbrace{\texttt{EB FF}}^{\text{JMP}}\ \texttt{C0}\quad \texttt{48 E8 F1}$$

```
.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

JMP

```
.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
```

<u>NOT</u> 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

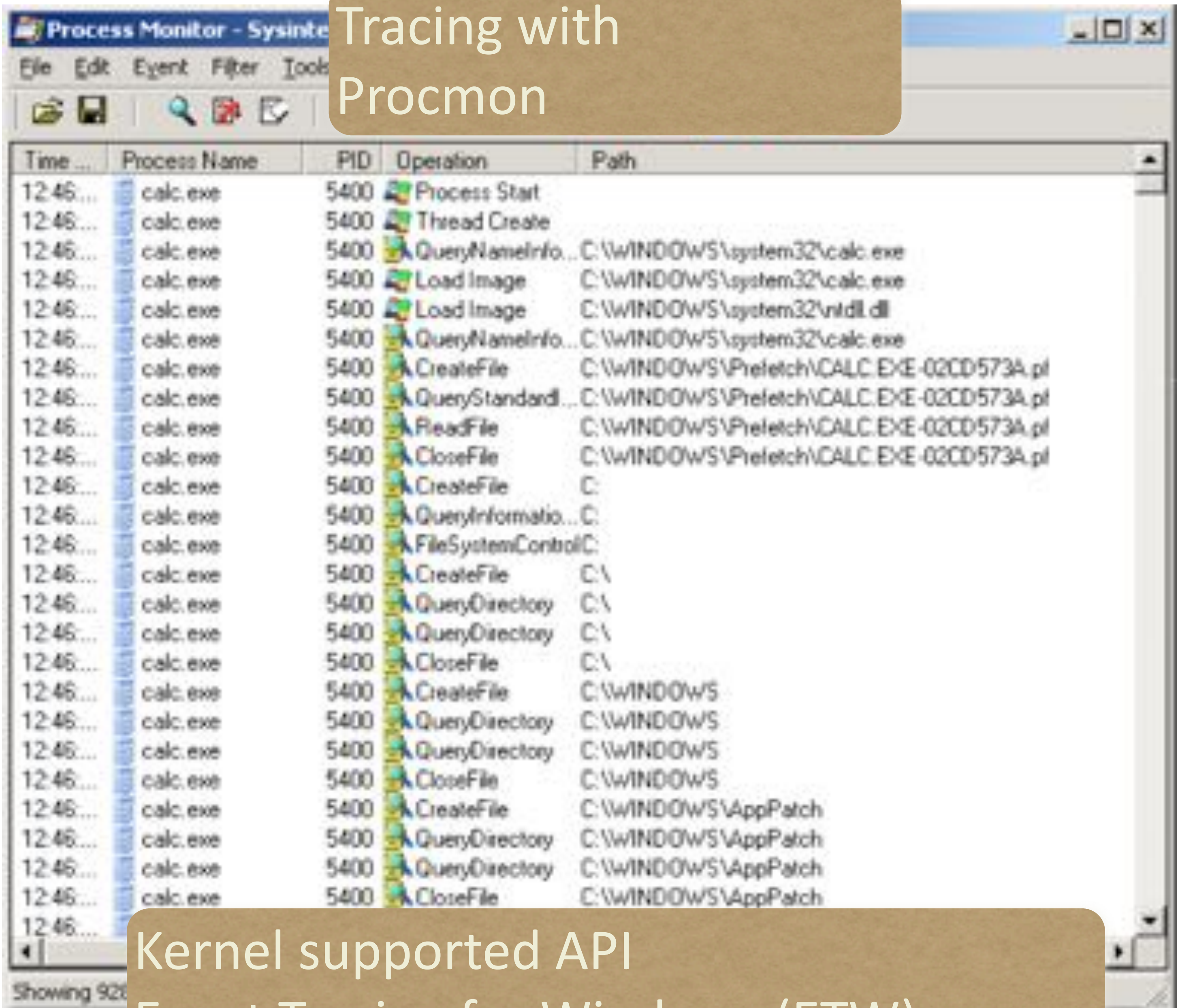
- IDA identifies stack variables by checking ESP math.
- Easy to throw off that analysis by abusing, or not following, calling conventions (e.g., compute variable offsets using ESP and strange math).
- That will absolutely wreck decompilers.

Dynamic Analysis

- A couple techniques available:

1. Tracing / Hooking
2. Debugging

Tracing with Procmon



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 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

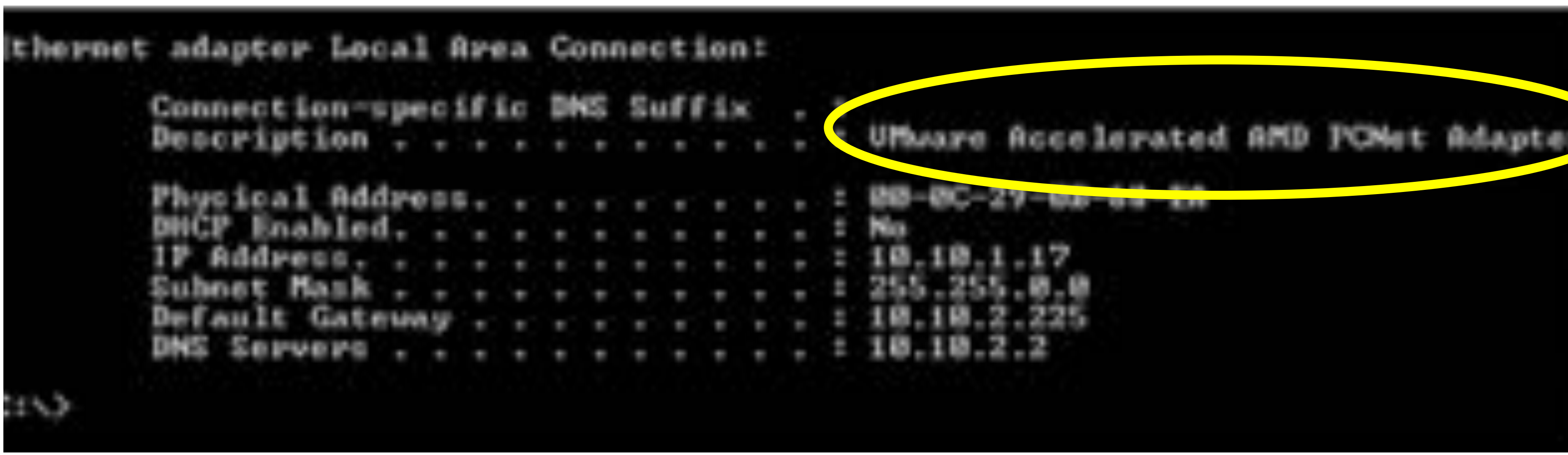
Easier

Harder

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

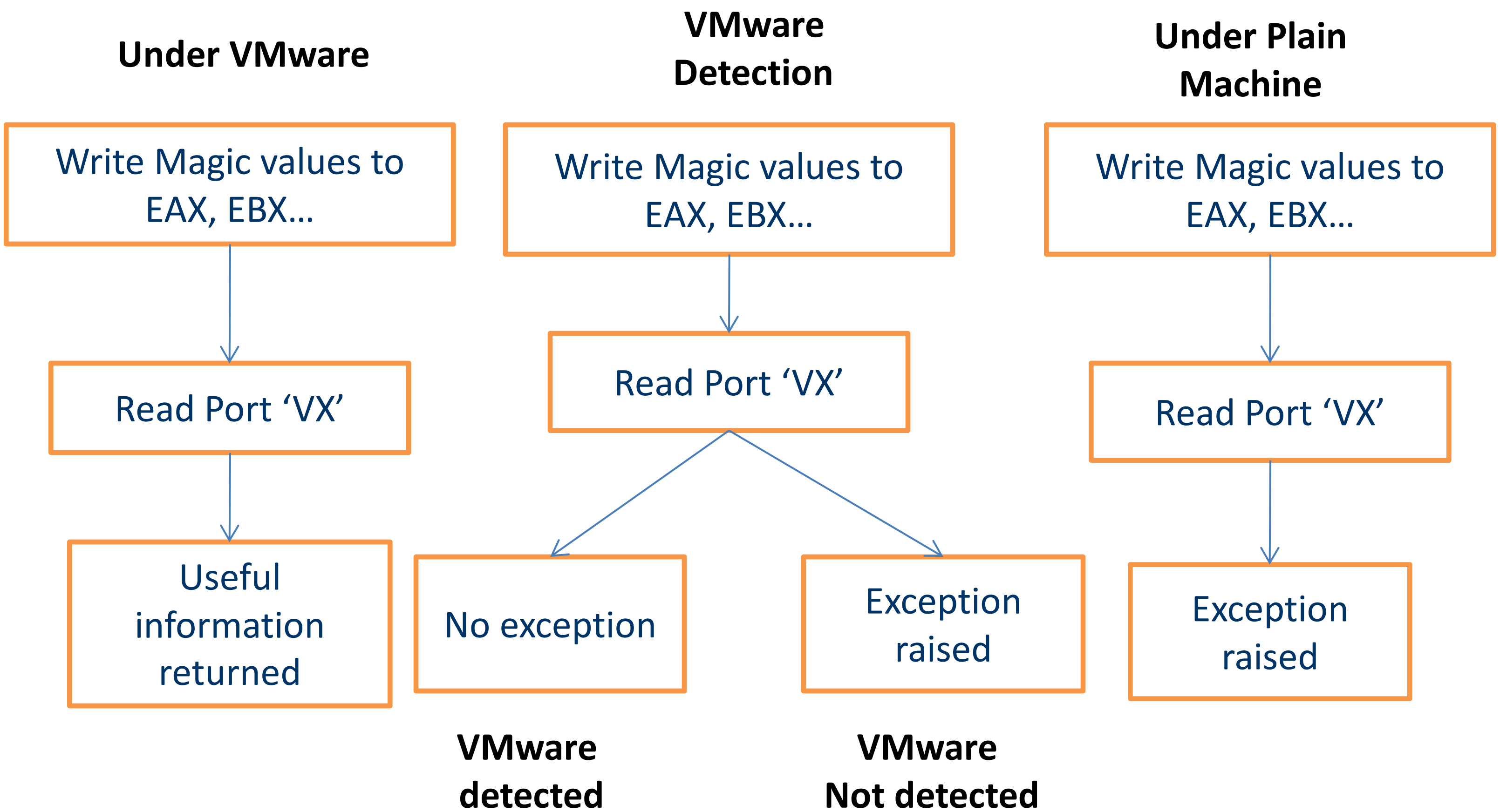
Example 1

- 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