

Lecture 36 – Finding Vulnerabilities 2

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

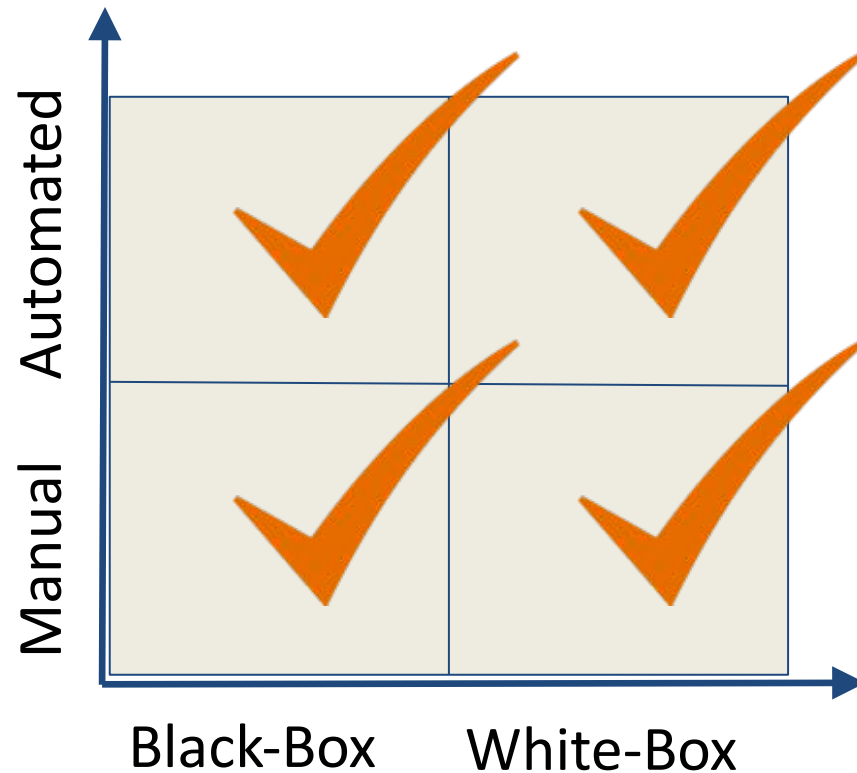
CS461 – Security 2

- Social networks
- Machine learning
- Web privacy
- Health IT
- Crypto constructs
- De-intenfication
- DoS
- Censorship
- Bitcoin
- Trusted computing
- Insider threats
- Steganography
- Smartphones

Security News

- Apple rushing patch for login vuln
- DOJ indicted Iranian for HBO hack

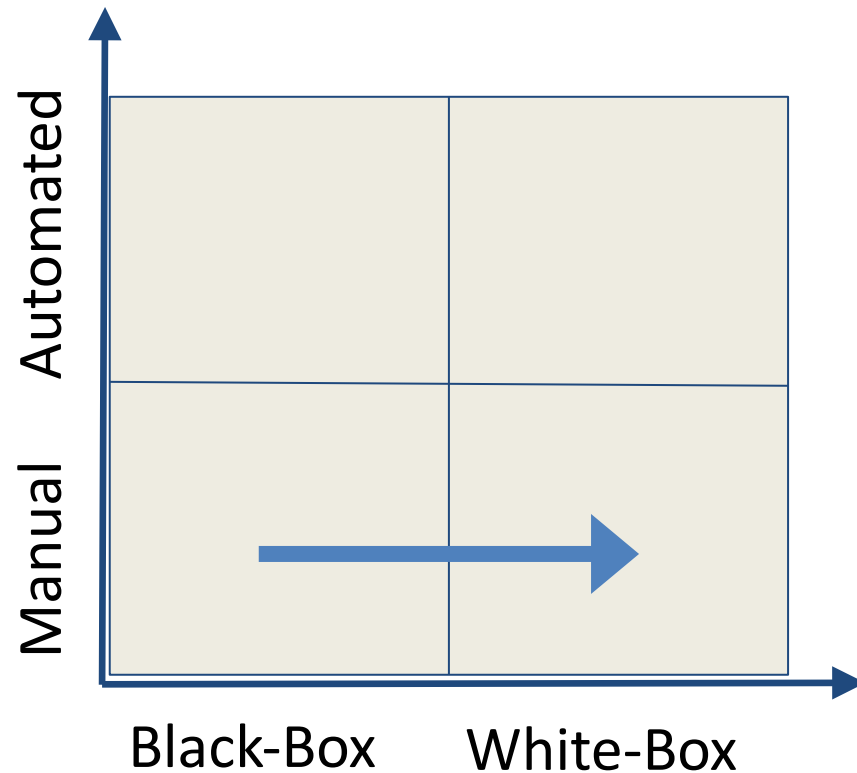
Classification of Testing Approaches



Fuzzing Components

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

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!

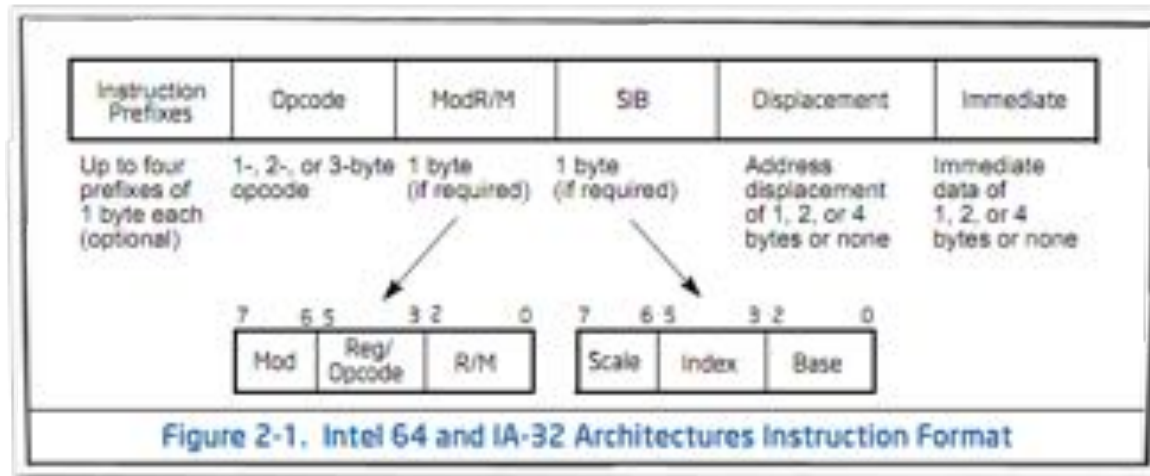
Legality

- 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



Control Flow Diagram



IDA - C:\Users\Delphi Ote\Desktop\Reversing\demo_stackframe.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Data Regular function Unexplored Instruction External symbol

Functions window

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_relocati
- _calloc

Graph overview

Output window

Executing Function "Unload"...

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)]

IDA Python v1.7.0 Final (serial 0) (c) The IDA Python 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

AD: idle Down Disk: 220GB

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

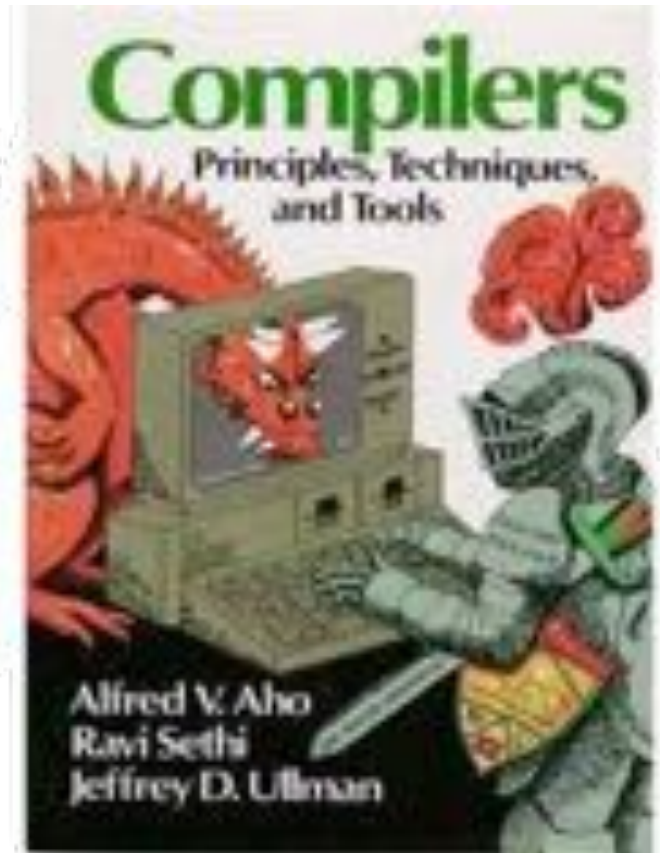
push    ebp
mov     ebp, esp
sub     esp, 18h
and     esp, 0FFFFFF0h
mov     eax, 0

```

100.014 (-58,-53) (187,539) 000004C1 004010C1: _main (Synchronized with Hex View-1)

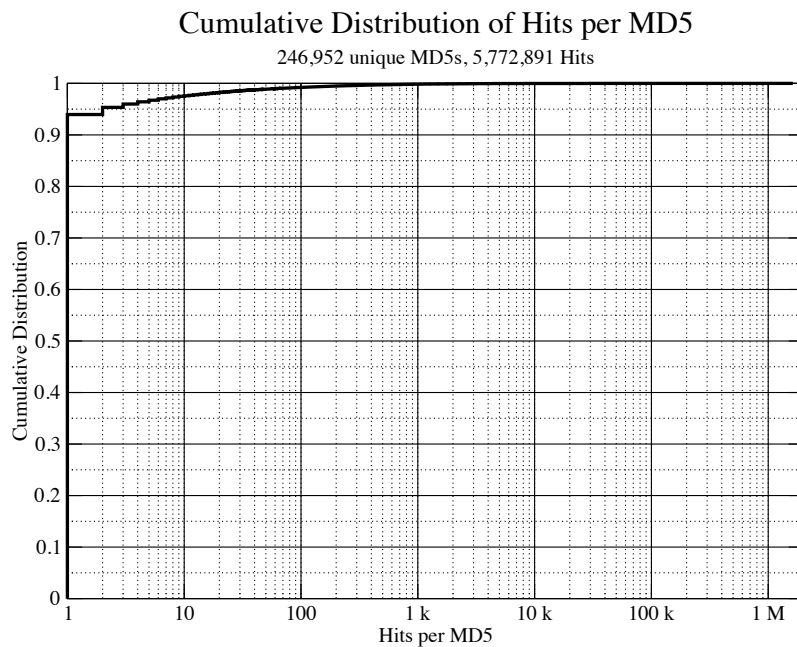
Difficulties

- 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%

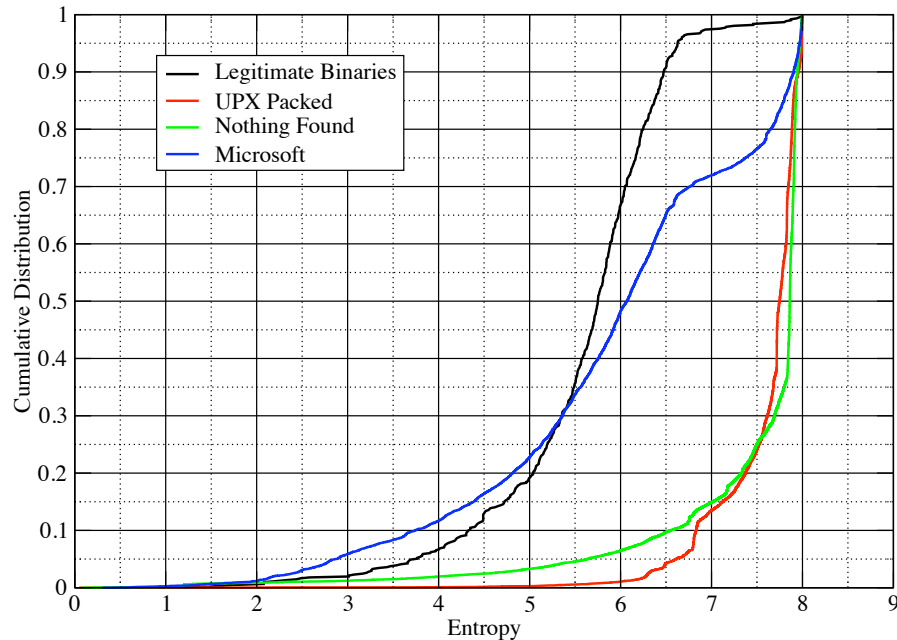
SigBuster	Count
Allaple	22050
UPX	11324
PECompact	5278
FSG	5080
Upack	3639
Themida	1679
NsPack	1645
ASpack	1505
tElock	1332
Nullsoft	1058

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

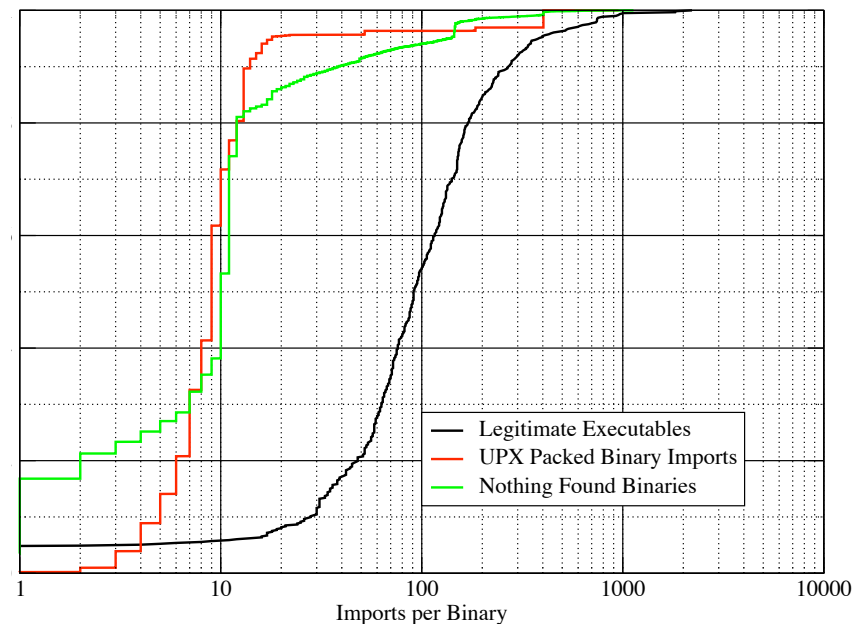
How about the unidentified?

Cumulative Distribution of Entropy per MD5

Using Ent tool, 613 legit, 11,326 UPX, 39,731 Nothing Found, 7,213 Microsoft



Cumulative Distribution of Imports per Binary

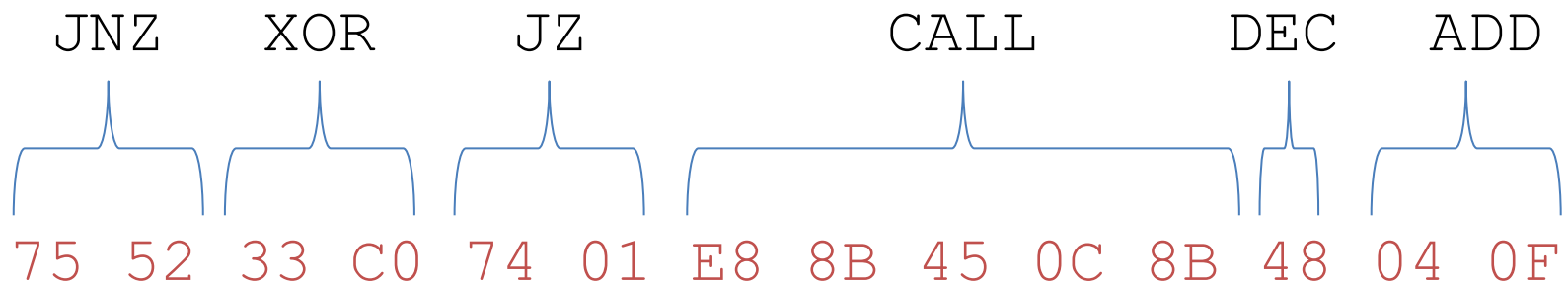


- Unidentified have: high entropy, small IATs
- Overall: > 90% packed

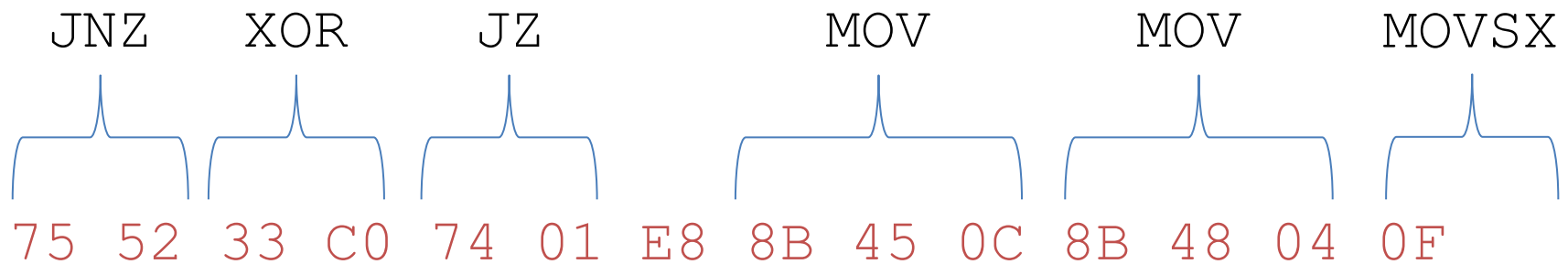
Anti-disassembly

- 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



```
.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
```

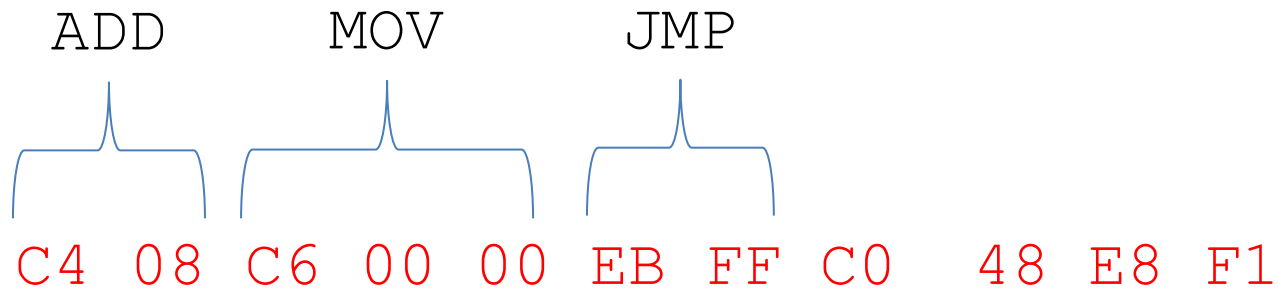


```
.text:0040100A jnz     short loc_40105E
.text:0040100C xor      eax,  eax
.text:0040100E jz      short loc_401011
.text:00401010 db      0E8h
.text:00401011 mov      eax,  [ebp+0Ch]
.text:00401014 mov      ecx,  [eax+4]
.text:00401017 movsx   edx,  byte ptr [ecx]
```

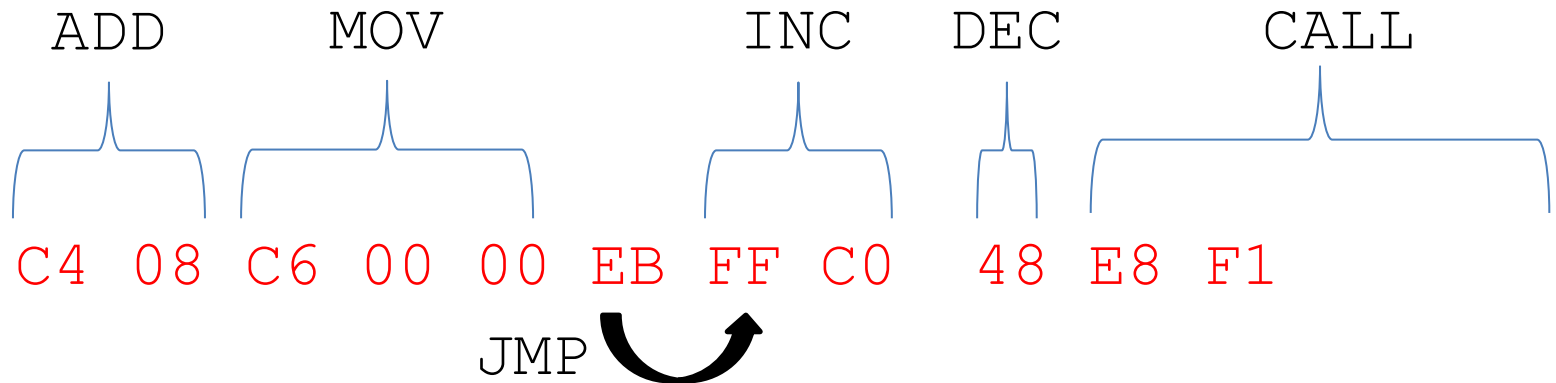
Rogue False Branch

- 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



```
.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 ; ±
```

```
.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;`

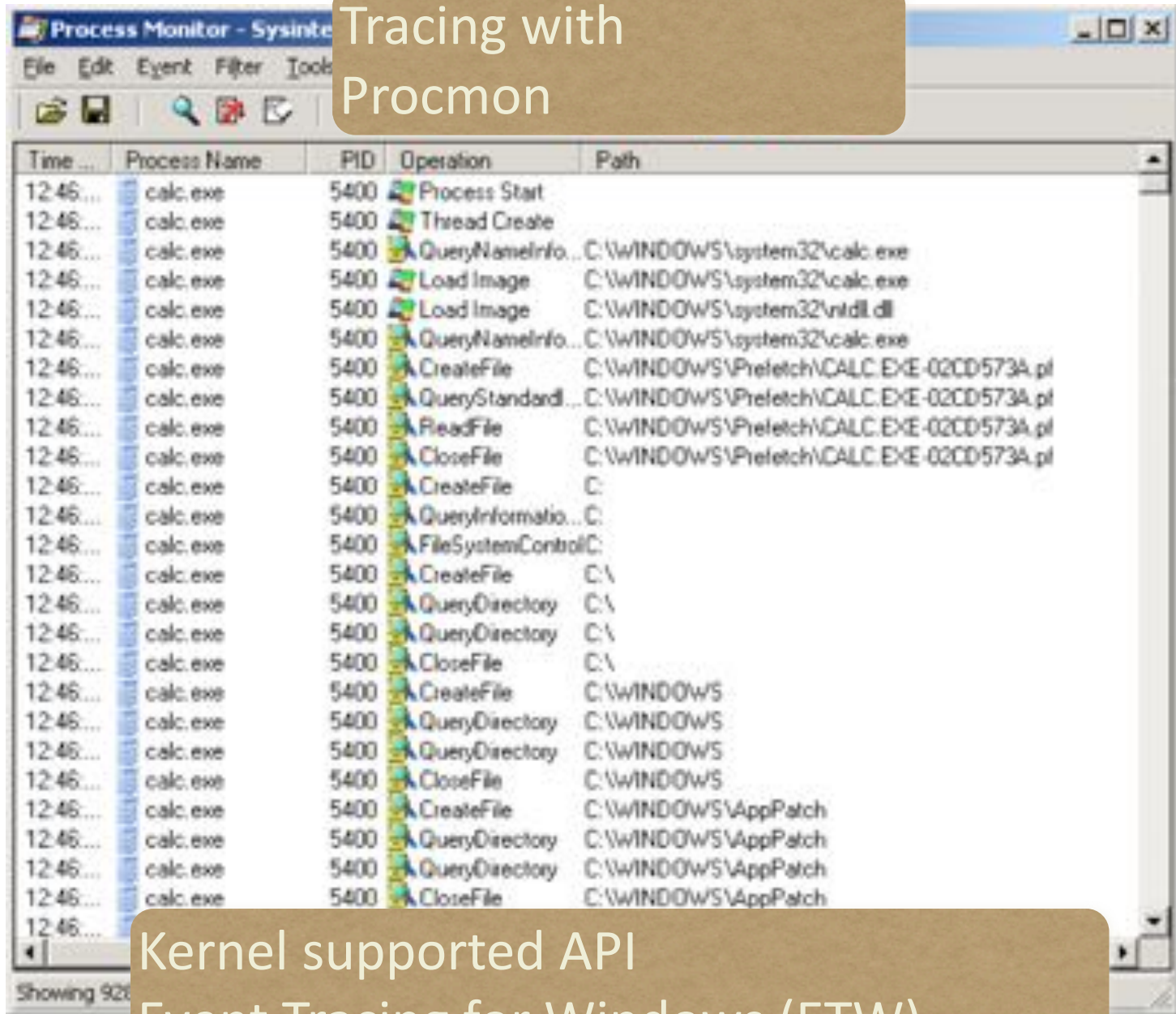
Thwarting Stack Analysis

- 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



The screenshot displays the Process Monitor (Procmon) application window. The title bar reads 'Process Monitor - Sysinte'. The menu bar includes 'File', 'Edit', 'Event', 'Filter', and 'Tools'. Below the menu is a toolbar with icons for file operations and search. The main area is a table with columns: 'Time ...', 'Process Name', 'PID', 'Operation', and 'Path'. The table shows a series of events for the 'calc.exe' process (PID 5400). The events include 'Process Start', 'Thread Create', 'QueryNameInfo...', 'Load Image' (for 'C:\WINDOWS\system32\calc.exe' and 'C:\WINDOWS\system32\ntdll.dll'), 'CreateFile' (for 'C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl'), 'QueryStandard...', 'ReadFile', 'CloseFile', 'CreateFile' (for 'C:\'), 'QueryInformatio...', 'FileSystemControlC:', 'CreateFile' (for 'C:\'), 'QueryDirectory' (for 'C:\'), 'CloseFile' (for 'C:\'), 'CreateFile' (for 'C:\WINDOWS'), 'QueryDirectory' (for 'C:\WINDOWS'), 'CloseFile' (for 'C:\WINDOWS'), 'CreateFile' (for 'C:\WINDOWS\AppPatch'), 'QueryDirectory' (for 'C:\WINDOWS\AppPatch'), and 'CloseFile' (for 'C:\WINDOWS\AppPatch'). The status bar at the bottom left indicates 'Showing 928'.

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.pl
12:46...	calc.exe	5400	QueryStandard...	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl
12:46...	calc.exe	5400	ReadFile	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl
12:46...	calc.exe	5400	CloseFile	C:\WINDOWS\Prefetch\CALC.EXE-02CD573A.pl
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		

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 ntdll.dll

Address	Hex dump	Comment	Comment	Registers (FPU)
00401000	44 00	PUSH 0		EAX 00000000
00401002	EB 05C60000	CALL 7JMP, 5A KERNEL32, 0x...	Kernel32	ECX 0012FF04
00401007	8B00	MOV EAX, EAX		EDX 7C903B94
00401009	EB 05C62000	CALL 0040F2F4		EDI 7FFD4000
0040100B	58	POP EAX		EIP 0012FFC0
0040100D	EB 05C62000	CALL 0040F2F4		EBP 0012FF00
00401010	EB 05C62000	CALL 0040F2F4		ESI 00000000
00401014	44 00	PUSH 0	Collydb	EDI 00000000
00401016	EB 05C62000	CALL 0040F2F4	Collydb	EIP 00401000
00401018	58	POP EAX	Collydb	
0040101A	58	POP EAX		

GetCollydb, 00401014

Address	Hex dump	Address	Value
00401000	44 00 4C 00 00 00 00 00 4C 00 00 00 00 20 4C 00	0012FF04	7C903B94
00401002	00 01 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	00000000
00401004	4C 00 00 00 00 00 00 00 4C 00 00 00 4C 00 00 1F	0012FFC0	00000000
00401006	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	00000000
00401008	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	7FFD4000
0040100A	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	005404F0
0040100C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	0012FFC0
0040100E	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	00000000
00401010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FFC0	FFFFFFFF

Log data

Address	Message
00401000	Module C:\WINDOWS\system32\ntdll.dll loaded
00401002	Entry point of main module
00401004	INT04 EAX = 0
00401006	EDI = 7FFD4000 (2147300000)
00401008	Collydb
00401010	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 intent if necessary
 - IF VM_PRESENT() or DEBUGGER_PRESENT()
 - Terminate() *// hide real intents*
 - ELSE
 - Malicious_Behavior() *//real intents*

Taxonomy of malware evasion



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

Example 1

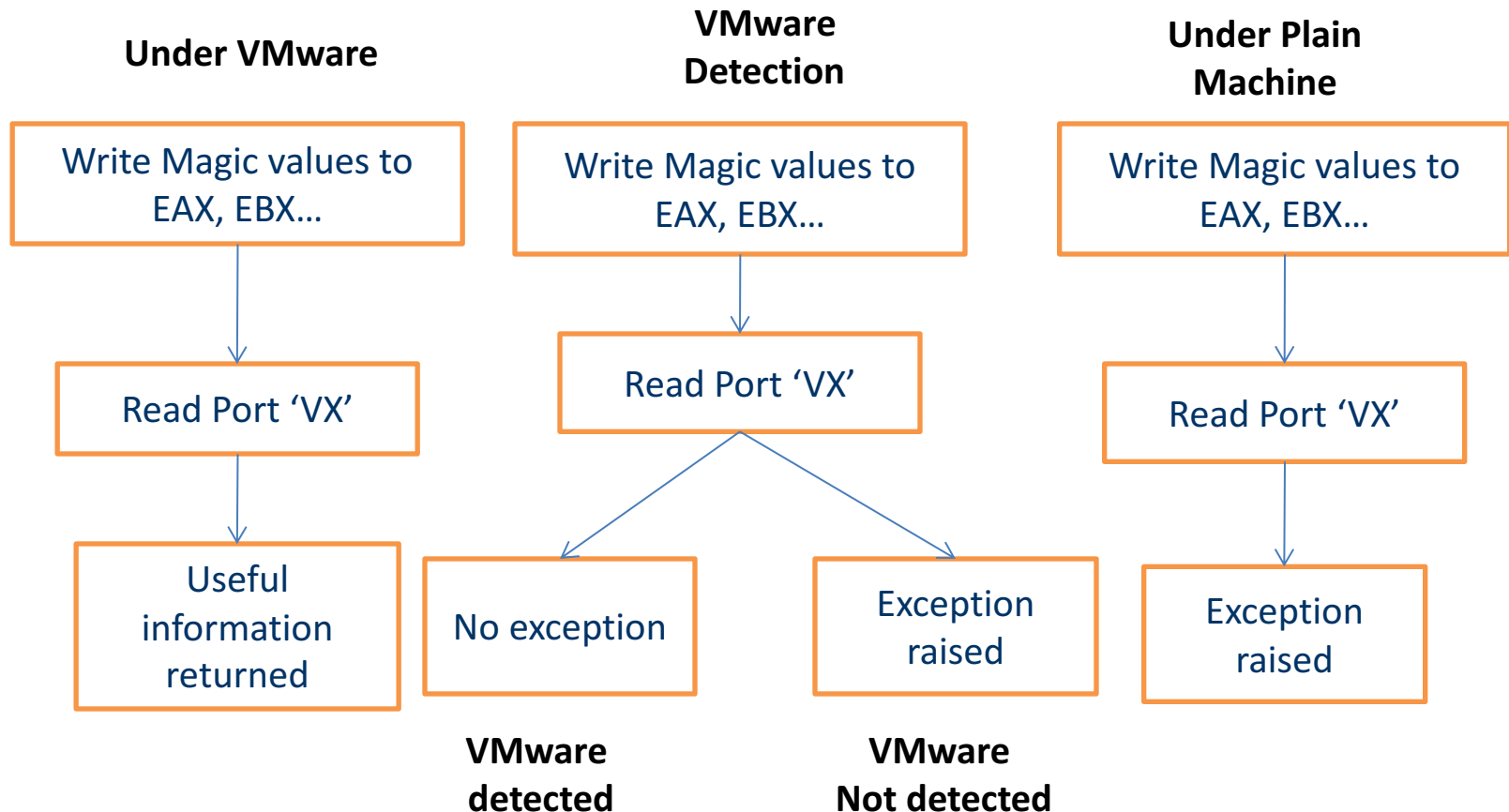
- Device driver strings
 - Network cards

```
Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : 
    Description . . . . . : VMware Accelerated AMD PCNet Adapter
    Physical Address. . . . . : 08-00-27-00-00-00
    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
```

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

