

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

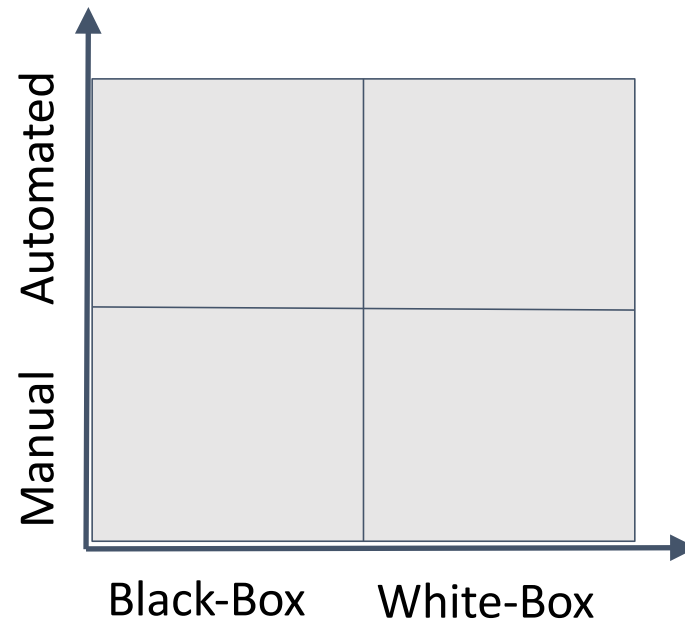
The Need for Specifications

- 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?

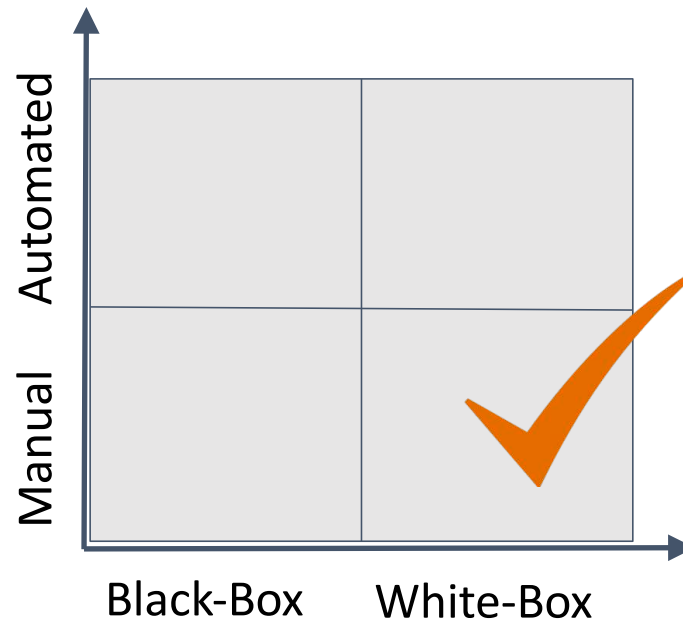
- How do we know that our test suite is good?
 - Too few tests: may miss bugs
 - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Example: SQLite

“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

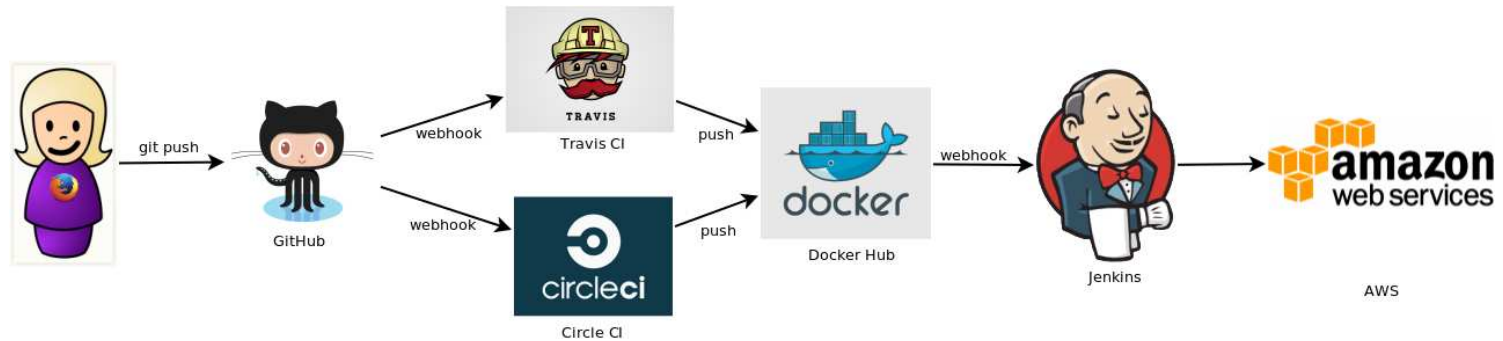
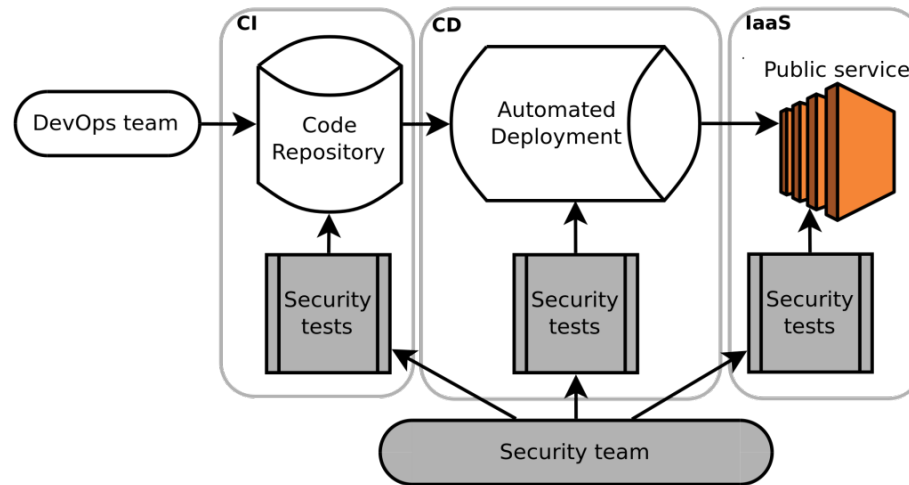
Classification of Testing Approaches



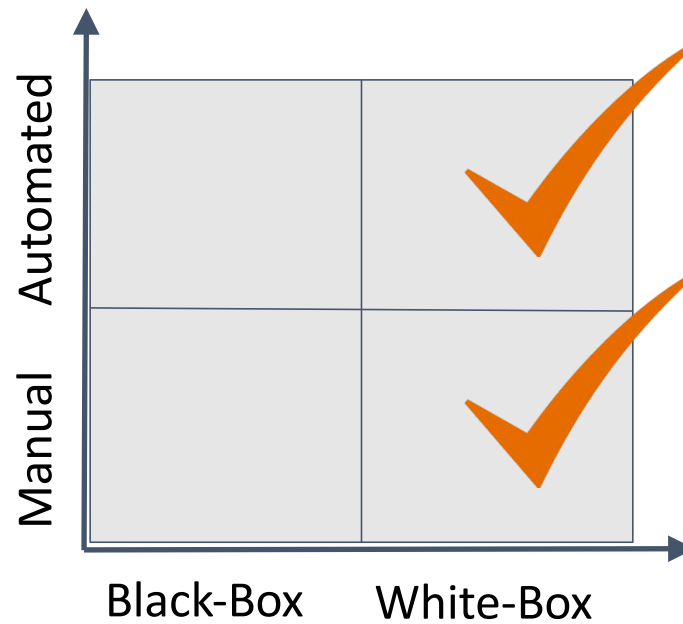
Manual white-box testing

- 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



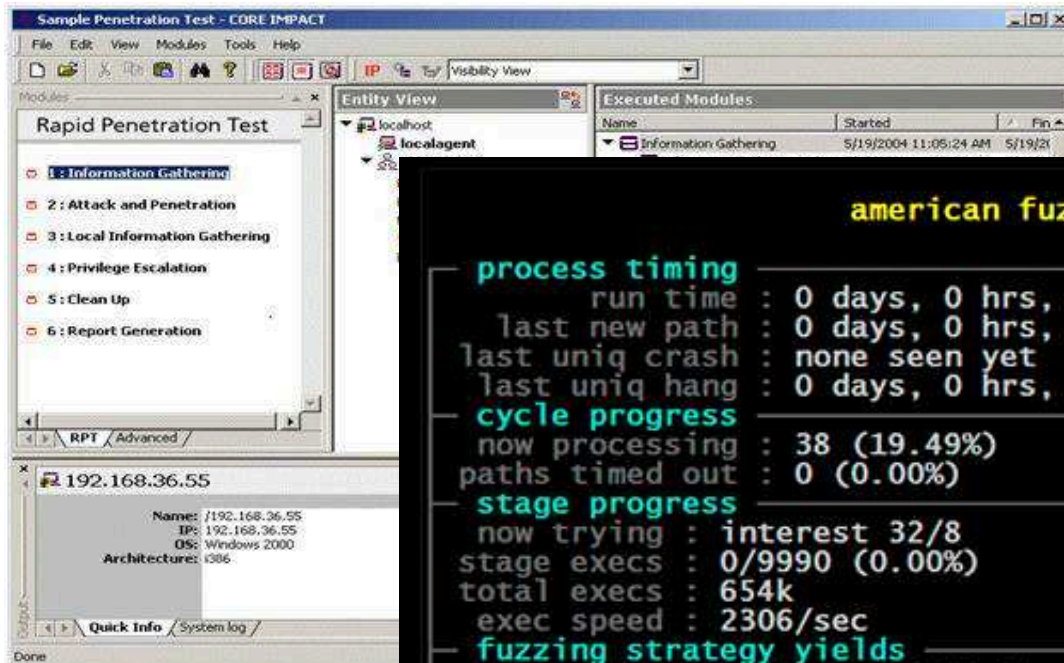
Classification of Testing Approaches



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

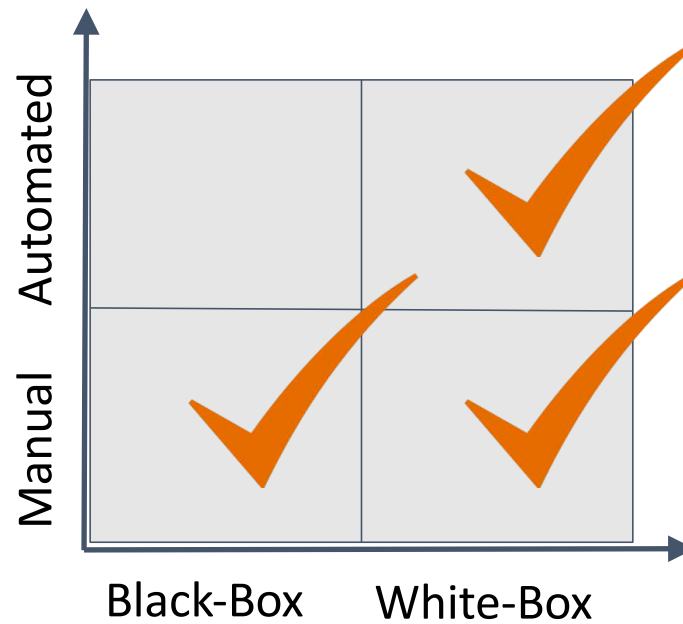


```

american fuzzy lop 0.47b (readpng)

process timing
  run time      : 0 days, 0 hrs, 4 min, 43 sec
  last new path : 0 days, 0 hrs, 0 min, 26 sec
  last uniq crash : none seen yet
  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec
cycle progress
  now processing : 38 (19.49%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : interest 32/8
  stage execs : 0/9990 (0.00%)
  total execs : 654k
  exec speed : 2306/sec
fuzzing strategy yields
  bit flips : 88/14.4k, 6/14.4k, 6/14.4k
  byte flips : 0/1804, 0/1786, 1/1750
  arithmetics : 31/126k, 3/45.6k, 1/17.8k
  known ints : 1/15.8k, 4/65.8k, 6/78.2k
  havoc : 34/254k, 0/0
  trim : 2876 B/931 (61.45% gain)
overall results
  cycles done : 0
  total paths : 195
  uniq crashes : 0
  uniq hangs : 1
map coverage
  map density : 1217 (7.43%)
  count coverage : 2.55 bits/tuple
findings in depth
  favored paths : 128 (65.64%)
  new edges on : 85 (43.59%)
  total crashes : 0 (0 unique)
  total hangs : 1 (1 unique)
path geometry
  levels : 3
  pending : 178
  pend fav : 114
  imported : 0
  variable : 0
  latent : 0
  
```

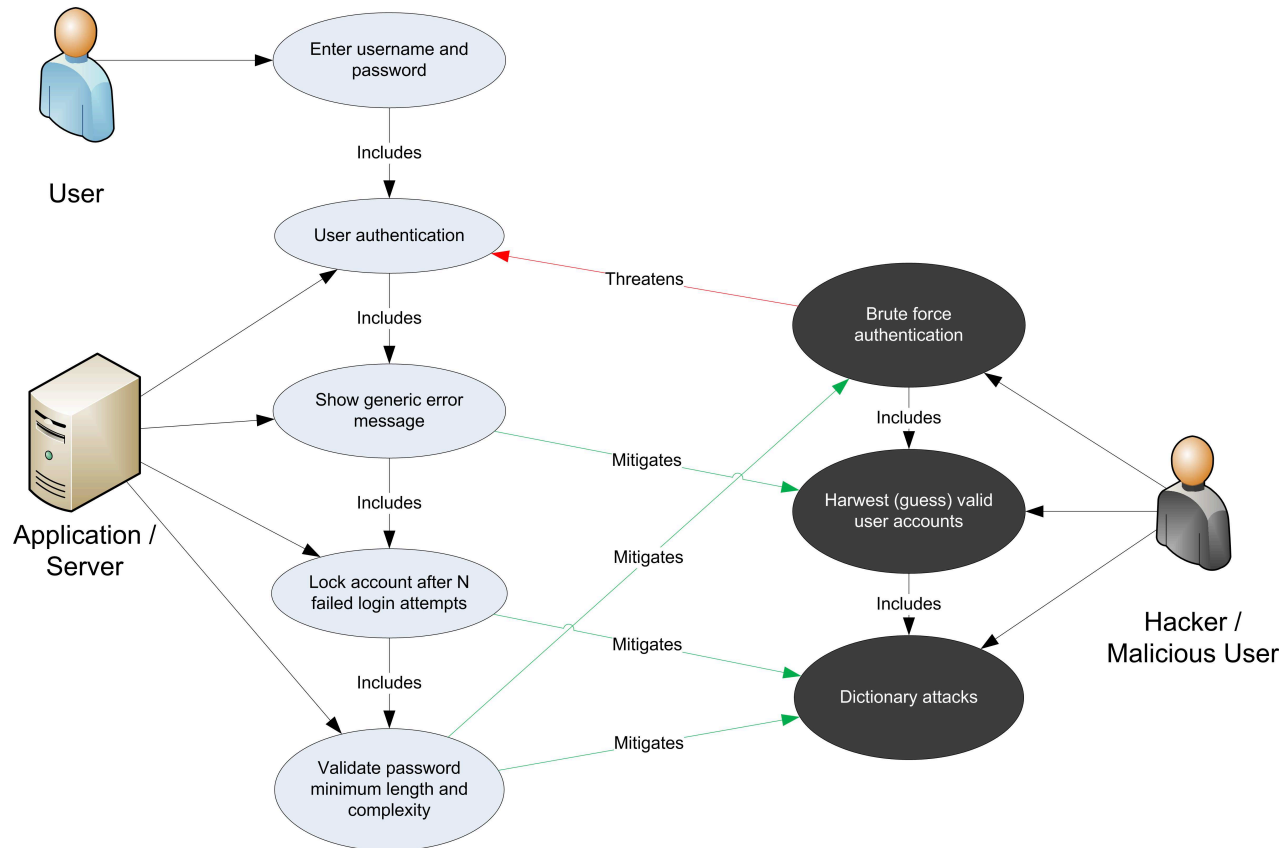
Classification of Testing Approaches



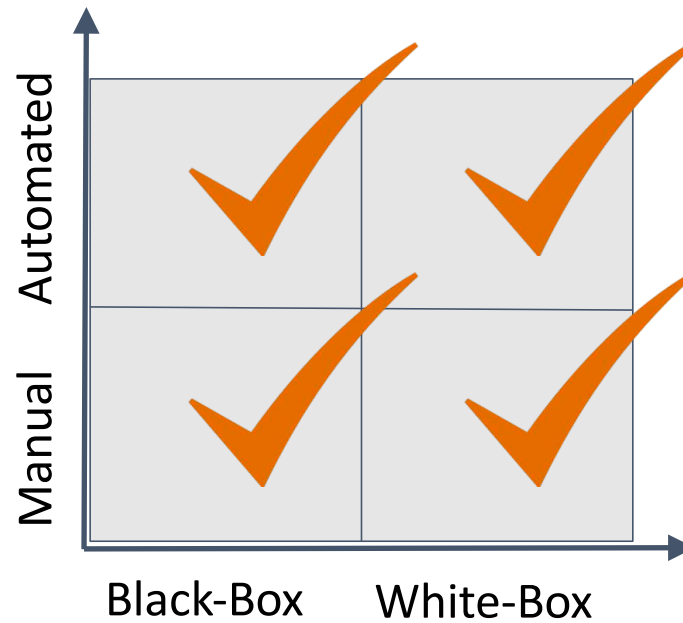
Manual black-box testing

- 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

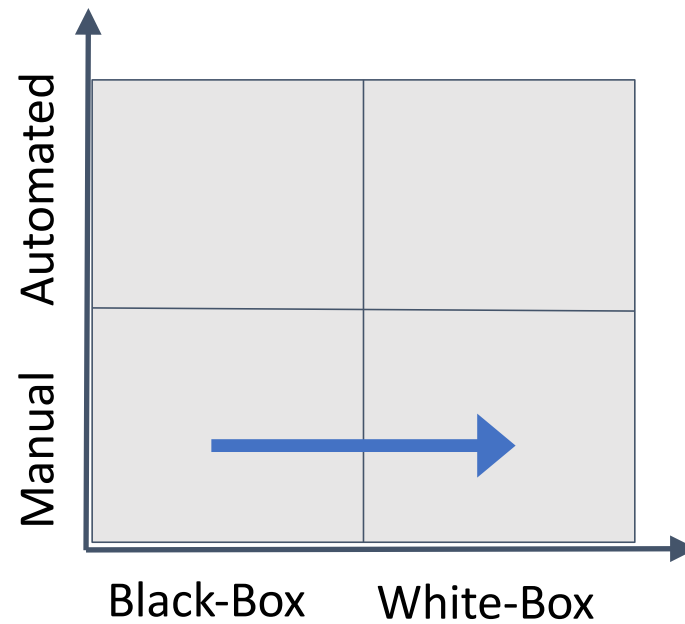
- Random Fuzzing
- “Dumb” (mutation-based) Fuzzing
 - Mutate an existing input
- “Smart” (generation-based) Fuzzing
 - Generate an input based on a model (grammar)

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

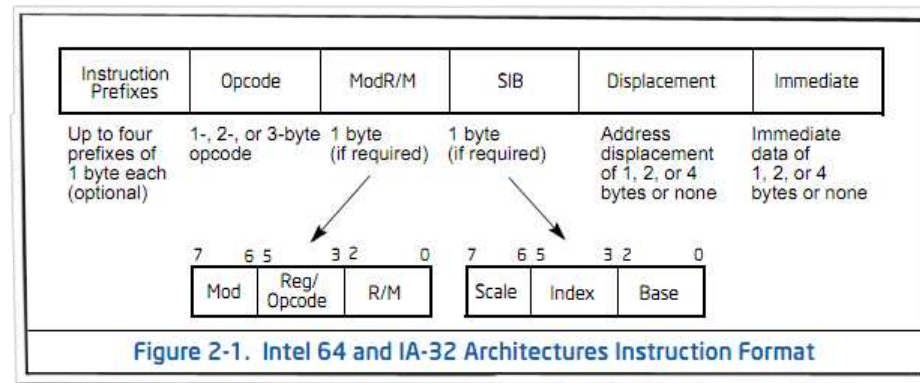
Legality

- 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”

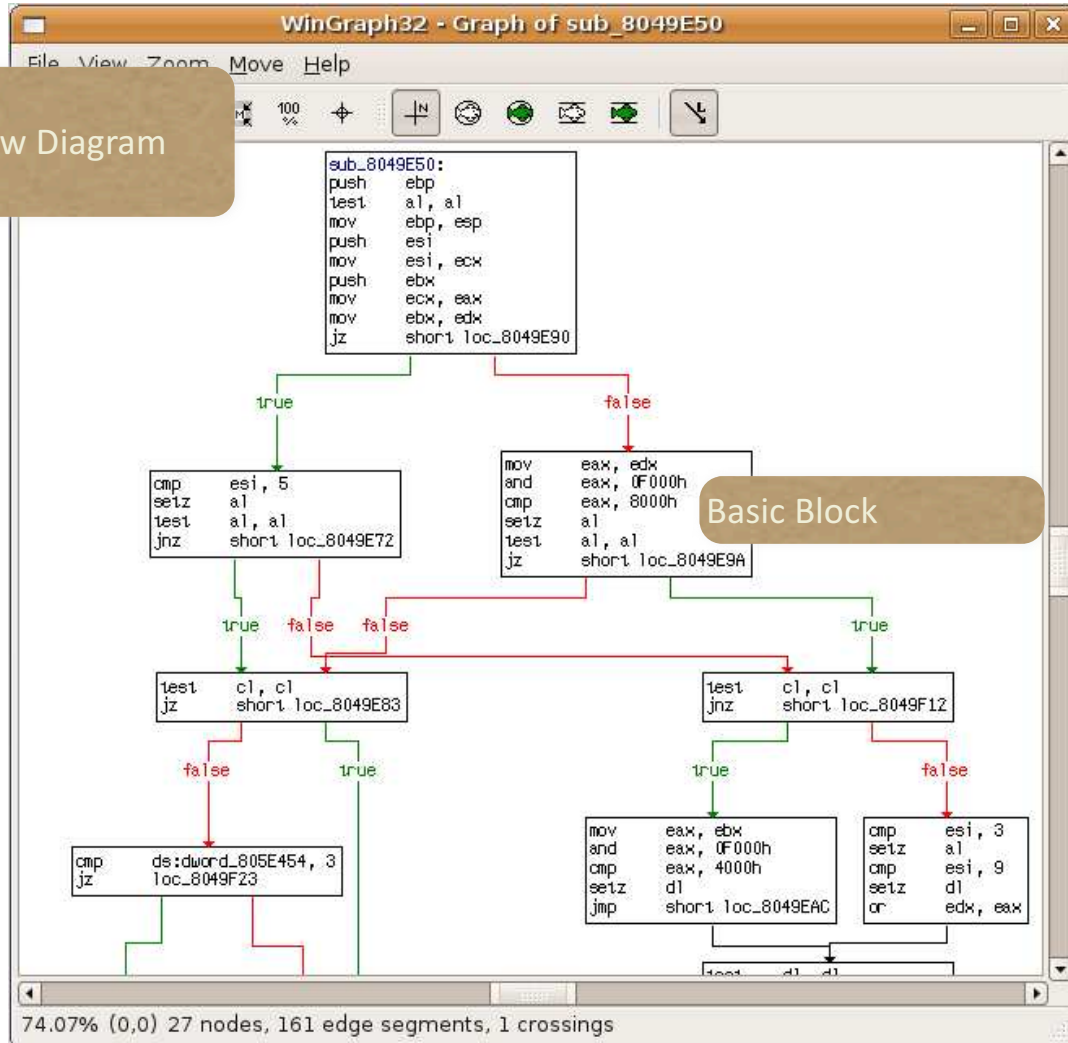
Two Techniques

- Static Code Analysis (structure)
 - Disassemblers
- Dynamic Code Analysis (operation)
 - Tracing / Hooking
 - Debuggers
- Combination of the two works best in my experience

Disassembly



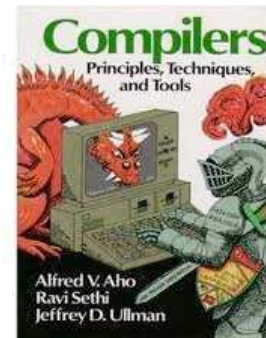
Control Flow Diagram



Basic Block

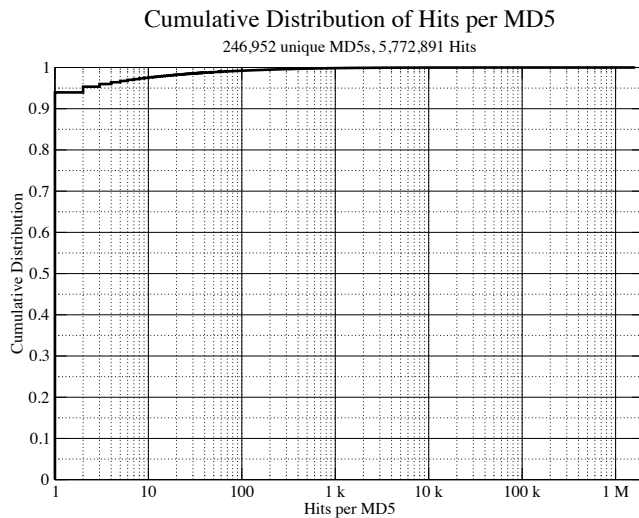
Difficulties

- 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

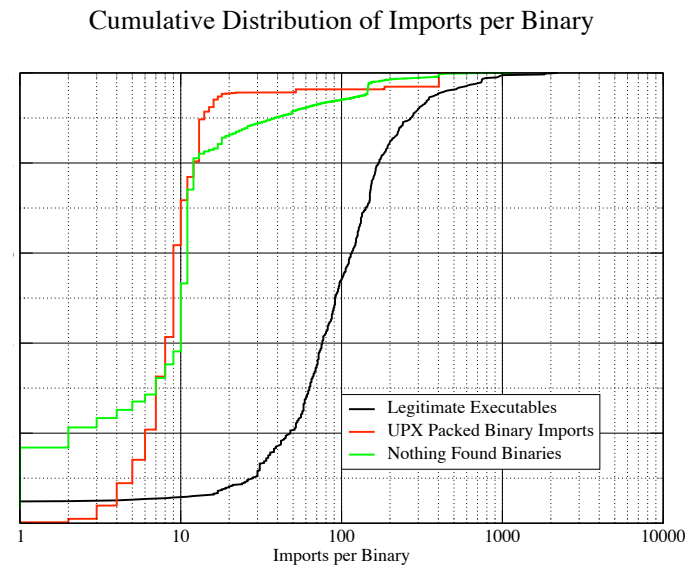
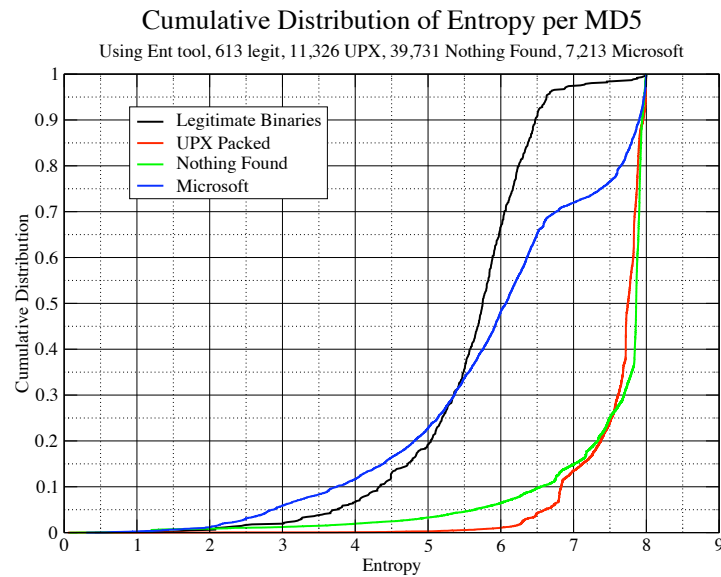
PEiD	Count
UPX	11244
Upack	6079
PECompact	4672
Nullsoft	2295
Themida	1688
FSG	1633
tElock	1398
NsPack	1375
ASpack	1283
WinUpack	1234

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

The screenshot displays the Process Monitor application window with a menu bar (File, Edit, Event, Filter, Tools) and a toolbar. The main area is a table of system events. The table has five columns: Time, Process Name, PID, Operation, and Path. The events are all for the process 'calc.exe' with PID 5400, occurring at 12:46:00. The operations include Process Start, Thread Create, QueryNameInfo, Load Image, CreateFile, QueryStandard, ReadFile, CloseFile, QueryInformation, FileSystemControl, QueryDirectory, and AppPatch.

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	QueryStandard...	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	FileSystemControl	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

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

C CPU - main thread, module ollydbg

Address	Hex dump	Command	Comment	Registers (FPU)
00401020	• 6A 00	PUSH 0		EAX 00000000
00401022	• E8 85C60E00	CALL <JMP.&KERNEL32.Get	Module: KERNEL	ECX 0012FFB4
00401027	• 8B00	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 0012FFF0
00401034	• E8 FBE20D00	CALL 004DF334		ESI 00000000
00401039	• 6A 00	PUSH 0	ollydbg [Arg1 =	EDI 00000000
0040103B	• E8 14F80D00	CALL 004E0854	ollydbg	EIP 0040103B ollydbg
00401040	• 59	POP ECX		
00401041	• 60 00F14F00	PUSH 00F14F00		

Dest=ollydbg.004E0854

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 00 91 4E 00 00 01 B0 93 4E 00 00 00 34 95	0012FFC4	7C81604F
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 08 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

L Log data

Address	Message
773D0000	Module C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b641-...
00401000	Entry point of main module
0040103B	INT3: EAX = 0 EBX = 7FFDA000 (2147328000) CDW0
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*

Taxonomy of malware evasion

Easier
↑
↓
Harder

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

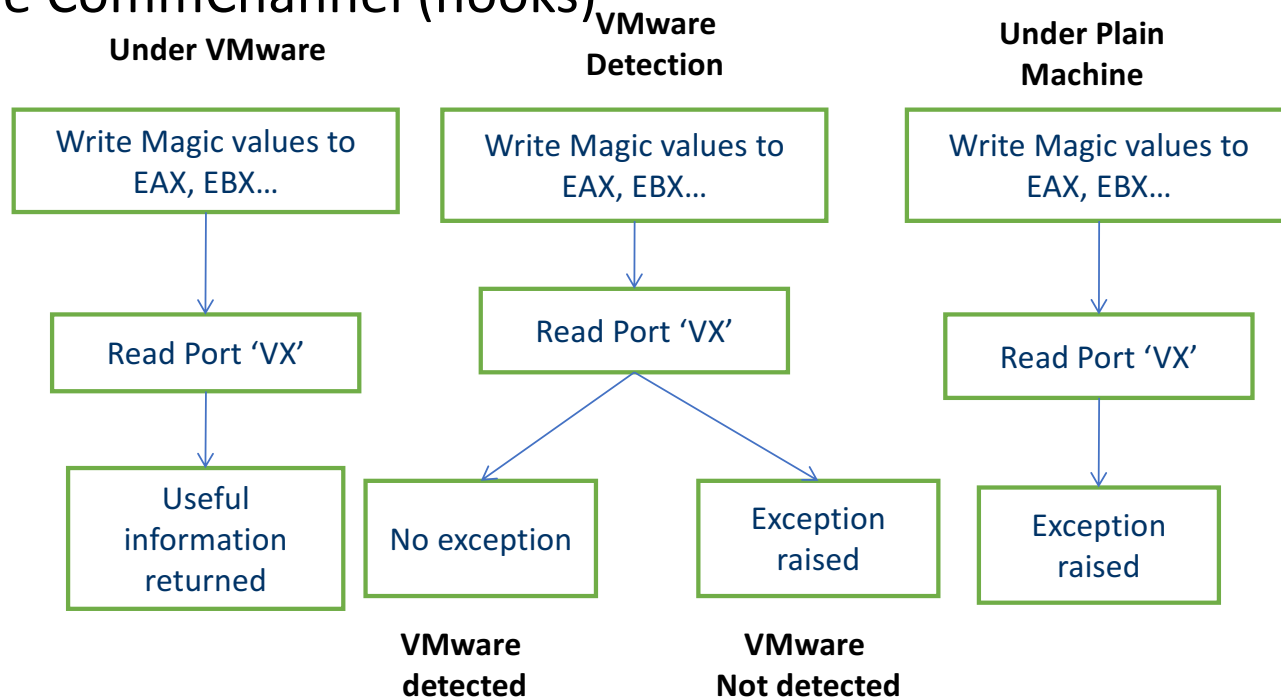
Example 1

- Device driver strings
 - Network cards

```
Ethernet adapter Local Area Connection:  
Connection-specific DNS Suffix . :  
Description . . . . . : VMware Accelerated AMD PCNet Adapter  
Physical Address. . . . . : 00-0C-29-8B-88-1A  
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)



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

