



Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware

Christopher Kruegel
Lastline, Inc.

Who am I?



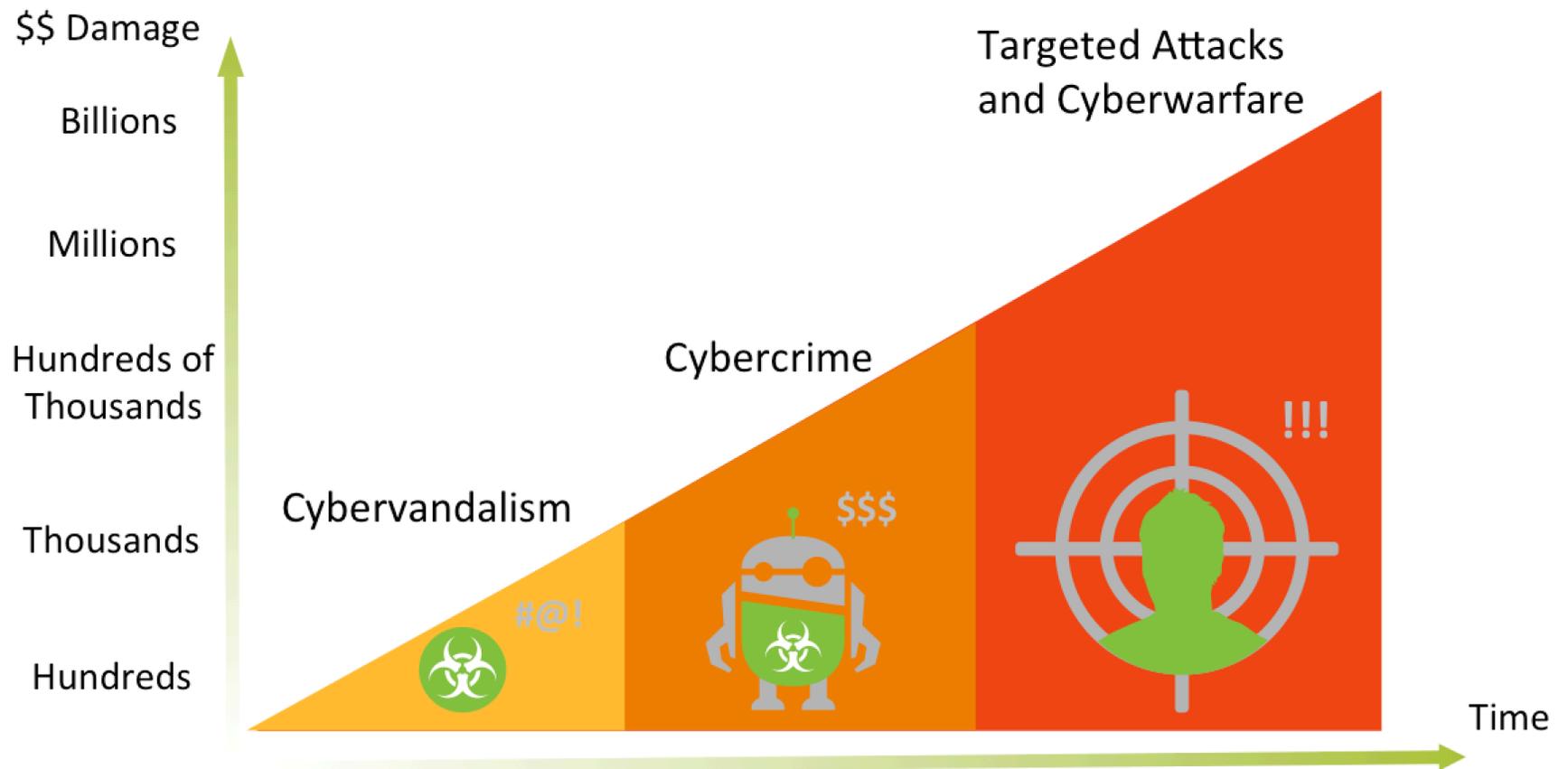
- Co-founder and Chief Scientist at Lastline, Inc.
 - Lastline offers protection against zero-day threats and advanced malware
 - effort to commercialize our research
- Professor in Computer Science at UC Santa Barbara (on leave)
 - many systems security papers in academic conferences
 - started malware research in about 2004
 - built and released practical systems (Anubis, Wepawet, ...)

What are we talking about?



- Automated malware analysis
 - how can we implement dynamic malware analysis systems
- Evasion as a significant threat to automated analysis
 - detect analysis environment
 - detect analysis system
 - avoid being seen by automated analysis
- Improvements to analysis systems
 - automate defenses against classes of evasion approaches

Evolution of Malware



Malware Analysis



OllyDbg - 601e77d9.exe
File View Debug Plugins Options Window Help

CPU - main thread, module ntdll

| Address | Hex dump | ASCII | Comment |
|----------|------------------|-------|--------------------------------|
| 7C90E8A8 | 68 00E9907C | | PUSH ntdll.7C90E900 |
| 7C90E8B0 | 64:A1 00000000 | | MOV EAX, DWORD PTR FS:[0] |
| 7C90E8B6 | 50 | | PUSH EAX |
| 7C90E8B7 | 8B4424 10 | | MOV EAX, DWORD PTR SS:[ESP+10] |
| 7C90E8B8 | 896C24 10 | | MOV DWORD PTR SS:[ESP+10], EBP |
| 7C90E8BF | 8D6C24 10 | | LEA EBP, DWORD PTR SS:[ESP+10] |
| 7C90E8C3 | 2BE0 | | SUB ESP, EAX |
| 7C90E8C5 | 53 | | PUSH EBX |
| 7C90E8C6 | 56 | | PUSH ESI |
| 7C90E8C7 | 57 | | PUSH EDI |
| 7C90E8C8 | 8B45 F8 | | MOV EAX, DWORD PTR SS:[EBP-8] |
| 7C90E8CB | 8965 E8 | | MOV DWORD PTR SS:[EBP-18], ESP |
| 7C90E8CE | 50 | | PUSH EAX |
| 7C90E8CF | 8B45 FC | | MOV EAX, DWORD PTR SS:[EBP-4] |
| 7C90E8D2 | C745 FC FFFFFFFF | | MOV DWORD PTR SS:[EBP-4], -1 |
| 7C90E8D9 | 8945 F8 | | MOV DWORD PTR SS:[EBP-8], EAX |
| 7C90E8DC | 8D45 F0 | | LEA EAX, DWORD PTR SS:[EBP-10] |
| 7C90E8DF | 64:A3 00000000 | | MOV DWORD PTR FS:[0], EAX |
| 7C90E8E5 | C3 | | RETN |
| 7C90E8E6 | 8B4D F0 | | MOV ECX, DWORD PTR SS:[EBP-10] |

EBP=0012FF3C
Stack SS:[0012FF00]=00000018

Registers (FPU)

| Register | Value |
|----------|--------------------------|
| EAX | 00000018 |
| ECX | 0012FFB0 |
| EDX | 7C90E4F4 ntdll.KiFastSys |
| EBX | 7FFD4000 |
| ESP | 0012FEF0 |
| EBP | 0012FF3C |
| ESI | 00020000 |
| EDI | 7C910208 ntdll.7C910208 |
| EIP | 7C90E8BB ntdll.7C90E8BB |

Threads

| Ident | Entry | Data block | Last error |
|----------|----------|------------|---------------|
| 000007CC | 004010B8 | 7FFDF000 | ERROR_SUCCESS |

Call stack of main thread

| Address | Stack | Procedure / arguments | Called from |
|----------|----------|----------------------------|-------------------|
| 0012FEF8 | 7C9103F9 | ? ntdll.7C90E8AB | ntdll.7C9103F4 |
| 0012FF04 | 7C801F10 | ? ntdll.RtlAcquirePebLock | kernel32.7C801F0A |
| 0012FF40 | 734235E3 | ? kernel32.GetStartupInfoA | MSUBUM60.734235D0 |
| 0012FF44 | 0012FF58 | pStartupinfo = 0012FF58 | |
| 0012FFBC | 004010C2 | ? <JMP.&MSUBUM60.#100> | 601e77d9.004010B0 |

Malware Analysis



OllyDbg - 601e77d9.exe

File View Debug Plugins Options Window Help

Windows Task Manager

Applications Processes Performance Networking Users

Image Name User Name CPU Mem Usage

| Image Name | User Name | CPU | Mem Usage |
|---------------------|-----------------|-----|-----------|
| wuauclt.exe | SYSTEM | 00 | 2,420 K |
| wscntfy.exe | user | 00 | 680 K |
| wpabaln.exe | user | 00 | 2,784 K |
| winlogon.exe | SYSTEM | 00 | 1,732 K |
| lurdx.exe | user | 00 | 388 K |
| taskmgr.exe | user | 02 | 4,296 K |
| System Idle Process | SYSTEM | 98 | 16 K |
| System | SYSTEM | 00 | 36 K |
| svchost.exe | LOCAL SERVICE | 00 | 740 K |
| svchost.exe | NETWORK SERVICE | 00 | 1,340 K |
| svchost.exe | SYSTEM | 00 | 8,176 K |
| svchost.exe | NETWORK SERVICE | 00 | 1,628 K |
| svchost.exe | SYSTEM | 00 | 1,308 K |
| spoolsv.exe | SYSTEM | 00 | 1,488 K |
| smss.exe | SYSTEM | 00 | 56 K |
| services.exe | SYSTEM | 00 | 1,376 K |
| OLLYDBG.EXE | user | 00 | 7,588 K |
| lsass.exe | SYSTEM | 00 | 968 K |
| jusched.exe | user | 00 | 520 K |
| explorer.exe | user | 00 | 13,452 K |

Processes: 24 CPU Usage: 2% Commit Charge: 98140K / 11820K

CPU - main thread, module ntdll

| Address | Hex dump | ASCII | 0012FEF0 |
|----------|-------------------------|-------|-----------|
| 00404000 | 00 00 00 00 00 00 00 00 | | 0012FEF4 |
| 00404008 | 00 00 00 00 00 00 00 00 | | 0012FEF8 |
| 00404010 | 00 00 00 00 00 00 00 00 | | 0012FEFC |
| 00404018 | 00 00 00 00 00 00 00 00 | | 0012FFF0 |
| 00404020 | 00 00 00 00 00 00 00 00 | | 0012FFF4 |
| 00404028 | 00 00 00 00 00 00 00 00 | | 0012FFF8 |
| 00404030 | 00 00 00 00 00 00 00 00 | | 0012FFF0C |
| 00404038 | 00 00 00 00 00 00 00 00 | | 0012FFF10 |
| 00404040 | 00 00 00 00 00 00 00 00 | | 0012FFF14 |
| 00404048 | 00 00 00 00 00 00 00 00 | | 0012FFF18 |
| 00404050 | 00 00 00 00 00 00 00 00 | | 0012FFF1C |
| 00404058 | 00 00 00 00 00 00 00 00 | | 0012FFF20 |
| 00404060 | 00 00 00 00 00 00 00 00 | | 0012FFF24 |
| 00404068 | 00 00 00 00 00 00 00 00 | | 0012FFF28 |
| 00404070 | 00 00 00 00 00 00 00 00 | | 0012FFF2C |

EBP=0012FF3C
Stack SS:[0012FF00]=00000018

Registers (FPU)

| Register | Value |
|----------|-----------------|
| EAX | 00000018 |
| ECX | 0012FFB0 |
| EDX | 7C90E4F4 ntdll. |
| EBX | 7FFD4000 |
| ESP | 0012FEF0 |
| EBP | 0012FF3C |
| ESI | 00020000 |
| EDI | 7C910208 ntdll. |
| EIP | 7C90E8BB ntdll. |
| C 0 | ES 0023 32bit |
| P 1 | CS 001B 32bit |
| A 0 | SS 0023 32bit |
| Z 0 | DS 0023 32bit |

Threads

| Ident | Entry | Data block |
|----------|----------|------------|
| 000007CC | 004010B8 | 7FFDF006 |

Call stack of main thread

| Address | Stack | Procedure / arguments | Called from |
|----------|----------|----------------------------|-------------------|
| 0012FEF8 | 7C9103F9 | ? ntdll.7C90E8AB | ntdll.7C9103F4 |
| 0012FF04 | 7C801F10 | ? ntdll.RtlAcquirePebLock | kernel32.7C801F0A |
| 0012FF40 | 734235E3 | ? kernel32.GetStartupInfoA | MSUBUM60.734235D0 |
| 0012FF44 | 0012FF58 | pStartupInfo = 0012FF58 | |
| 0012FFB0 | 004010C2 | ? <JMP.&MSUBUM60.#100> | 601e77d9.004010B0 |

Malware Analysis



The screenshot displays a Windows desktop environment used for malware analysis. The primary window is OllyDbg, which is debugging the process '601e77d9.exe'. The CPU window shows assembly instructions for the main thread in the ntdll module, with the instruction at address 896C24 (MOV EDI, EDI) highlighted. The registers window shows EAX at 00000018 and EIP at 7C90E8BB. The threads window shows the current thread at address 00000000. The stack window shows the current stack frame at address 0012FF3C. The network traffic window shows a list of captured packets, with the selected packet (No. 11) being an HTTP GET request for '/images/led/hg.php'. The Windows Task Manager window is open in the background, showing a list of running processes. The 'Processes' tab is active, and 'urdx.exe' is highlighted, showing it is running under the 'user' account with 0% CPU usage and 388 K of memory usage. Other visible processes include 'wuauclt.exe', 'wscntfy.exe', 'wpabaln.exe', 'winlogon.exe', 'taskmgr.exe', 'System Idle Process', 'System', 'svchost.exe', 'spoolsv.exe', 'smsv.exe', 'services.exe', 'OLLYDBG.EXE', 'lsass.exe', 'jusched.exe', and 'explorer.exe'. The Task Manager shows overall system statistics: Processes: 24, CPU Usage: 2%, and Commit Charge: 98140K / 11820C.

Malware Analysis



The screenshot displays a Windows desktop environment used for malware analysis. Three main windows are visible:

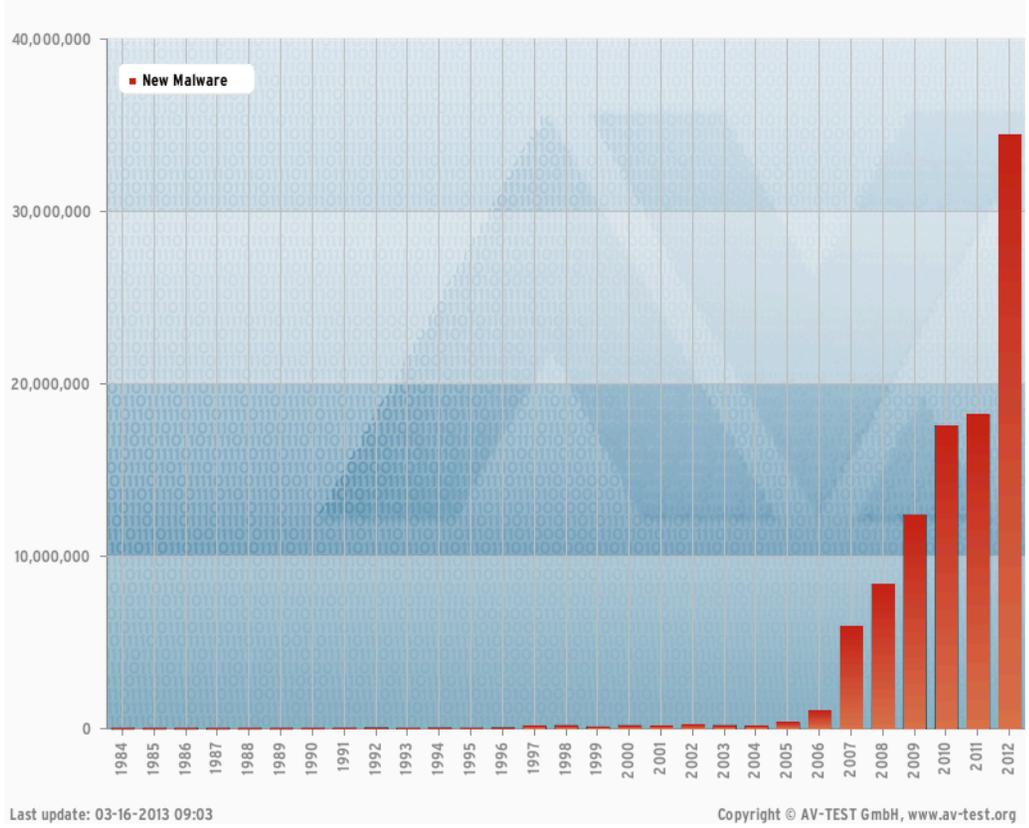
- Registry Editor:** Shows the system registry tree on the left and a list of registry values on the right. The path is `My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters`. Values include `Domain`, `HostName`, `IPv6Enabled`, `NameServer`, `NW HostName`, and `SearchList`.
- Windows Task Manager:** Shows the 'Processes' tab. The process `lrdyxc.exe` is selected, showing it is running under the 'user' account with 388 K of memory usage.
- Wireshark:** Shows a network capture of traffic. The selected packet is an HTTP GET request from `192.168.0.2` to `192.168.0.2` for the file `led/hg.php`. The request includes headers for `Accept-Encoding`, `User-Agent`, and `Host`.

There is a lot of malware out there ...



New Malware

▶ All years ▶ Last 10 years ▶ Last 5 years ▶ Last 24 months ▶ Last 12 months



Automated Malware Analysis



- Aka sandbox
- Automation is great!
 - analysts do not need to look at each sample by hand (debugger)
 - only way to stem flood of samples and get scalability
 - can handle zero day threats (signature-less defense)
- Implemented as instrumented execution environment
 - run program and observe its activity
 - make determination whether code is malicious or not

What do we want to monitor?



1. Persistent changes to the operating system, network traffic
 - a file was written, some data was exchanged over the network

A light purple rectangular notebox with a black border and a folded bottom-right corner. It contains the following text:

```
c:\sample.exe
```

```
net: 192.168.0.1  
-> evil.com:80
```

What do we want to monitor?



1. Persistent changes to the operating system, network traffic
 - a file was written, some data was exchanged over the network

- Can be done with post hoc monitoring of file system and external capturing of network traffic
 - easy to implement
 - allow malware to run on bare metal and unmodified OS (stealthy)
 - quite poor visibility (no temporary effects, sequence of actions, memory snapshots, data flows, ...)

What do we want to monitor?



2. Interactions between the program (malware) and the environment (operating system)

```
open c:\sample.exe
read c:\secret.exe
write c:\tmp\a.txt
net: 192.168.0.1
-> evil.com:80
delete c:\tmp\a.txt
write c:\sample.exe
```

What do we want to monitor?

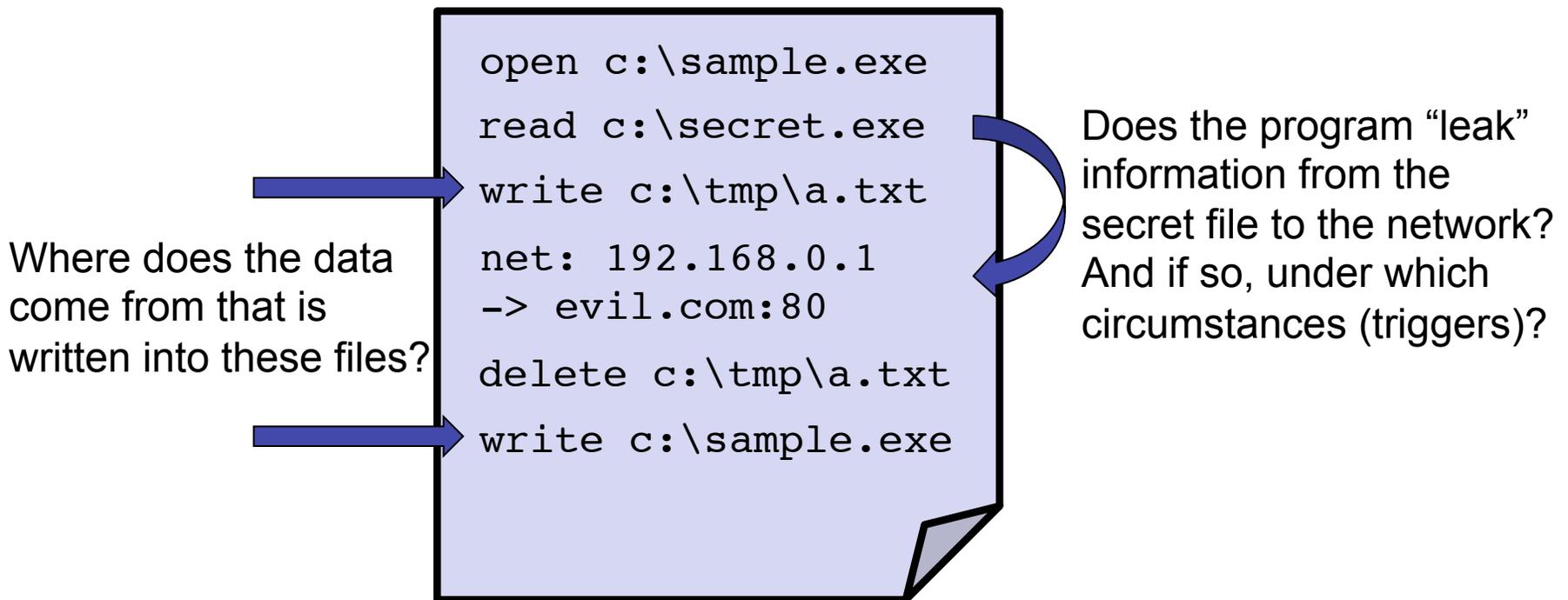


2. Interactions between the program (malware) and the environment (operating system)
 - Can be done by instrumenting the operating system or libraries (install system call or library call hooks)
 - typically done by running modified OS image inside virtual machines, used by many (most) vendors
 - can see temporary effects, sequence of operations, more details
 - very limited visibility into program operations (instructions)
 - limited visibility of memory (where does data value come from?)

What do we want to monitor?



3. Details of the program execution (how does the program process certain inputs, how are outputs produced, which checks are done)?



What do we want to monitor?



3. Details of the program execution (how does the program process certain inputs, how are outputs produced, which checks are done)?
 - Can be implemented through process emulation (CPU instructions + some Windows API calls) or a debugger
 - provides single instruction visibility
 - can potentially detect triggers and *data flows*
 - poor fidelity (**some** Windows API calls)
 - very slow and easy to detect (debugger)
 - produces a lot of data, so analysis must be able to leverage it

What do we want to monitor?



4. Details of the program execution while maintaining good fidelity?

What do we want to monitor?



4. Details of the program execution while maintaining good fidelity?
 - Can be implemented through full system emulation (running a real OS on top of emulated hardware – CPU / memory)
 - provides single instruction visibility
 - can detect triggers and data flows
 - much better fidelity (real Windows)
 - not as fast as native execution (or VM), but pretty fast
 - produces a lot of data, so analysis must be able to leverage it

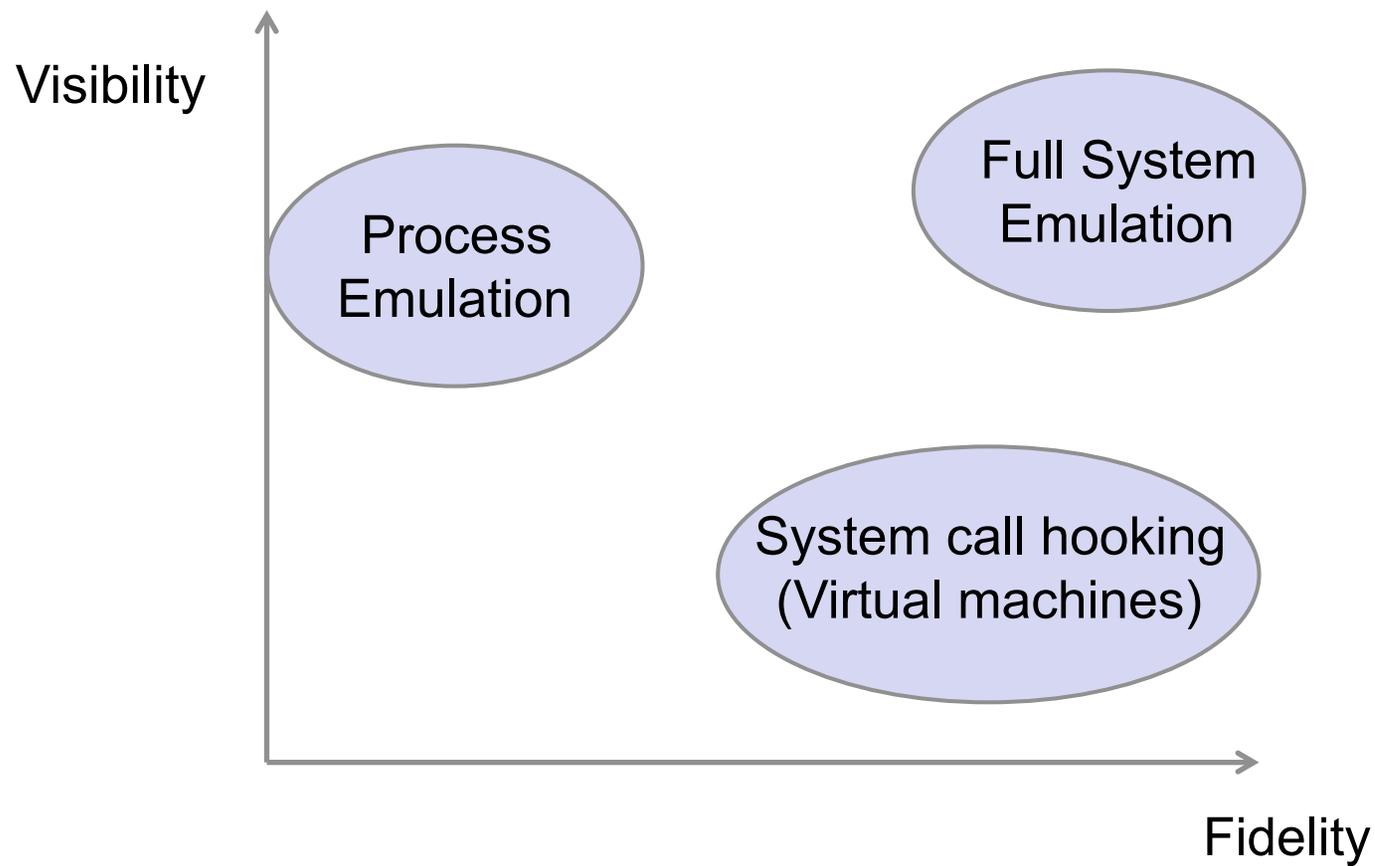
VM Approach versus CPU Emulation



```
callq 0x100070478 ; symbol stub for: _open
callq 0x1000704b4 ; symbol stub for: _read
callq 0x1000702b6 ; symbol stub for: _close
```

```
cmpl $0x0c,%ebx
je 0x10000f21e
xorl %esi,%esi
movq %r15,%rdi
xorl %eax,%eax
callq 0x100070478 ; symbol stub for: _open
movl %eax,%r12d
testl %eax,%eax
js 0x10000f21e
leaq 0xffffffff70(%rbp),%rcx
movq %rcx,0xfffffec0(%rbp)
movl $0x00000050,%edx
movq %rcx,%rsi
movl %eax,%edi
callq 0x1000704b4 ; symbol stub for: _read
movq %rax,%r13
movl %eax,%r14d
movl %r12d,%edi
callq 0x1000702b6 ; symbol stub for: _close
cmpl $0x02,%r13d
jle 0x10000f21e
```

Dynamic Analysis Approaches



Our Automated Malware Analysis



Anubis: *ANalyzing Unknown BInarieS* (university project)
and its successor (which was built from scratch)

llama: *LastLine Advanced Malware Analysis*

- based on full system emulation
- can see every instruction!
- monitors system activity from the outside (stealthier)
- runs real operating system
 - requires mechanisms to handle semantic gap
- general platform on which additional components can be built

Visibility Does Matter



- See more types of behavior
 - which connection is used to leak sensitive data
 - allows automated detection of C&C channels
 - how does the malware process inputs from C&C channels
 - enumeration of C&C commands (and malware functionality)
 - insights into keyloggers (often passive in sandbox)
 - take memory snapshots after decryption for forensic analysis
- Combat evasion
 - detect triggers
 - bypass stalling code
 - much more about this later ...

Detecting Keyloggers



- Software-based keyloggers
 - `SetWindowsHook`: intercepts events from the system, such as keyboard and mouse activity
 - `GetAsyncKeyState` or `GetKeyState`
- User simulation module that triggers actions likely to be monitored by keyloggers
 - Type on keyboard
 - Insert special data values (e.g., “valid” credit card numbers, passwords, email addresses, etc.)
- Track sensitive data and how it is used by the malware

Detecting Keyloggers



Threat Level

The file was found to be **malicious** at 2014-05-09 01:38:35.

Risk Assessment

Maliciousness score: **100/100**

Risk estimate: High Risk - Malicious behavior detected

Malicious Activity Summary

| Type | Description |
|-----------|--|
| Autostart | Registering for autostart using the Windows start menu |
| Evasion | Possibly stalling against analysis environment (loop) |
| File | Modifying executable in user-shared data directory |
| Signature | Identified trojan code |
| Steal | Keystroke logging capabilities |
| Stealth | Creating executables masquerading system files |
| Stealth | Deleting the sample after execution |

Detecting Keyloggers



Analysis Subject 2

| | |
|--------------------------|---|
| MD5 | 21f8b9d9a6fa3a0cd3a3f0644636bf09 |
| SHA1 | 0392f25130ce88fdee482b771e38a3eaae90f3e2 |
| Command Line | "C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup\spoolsv.exe" C:\Users\...\chewbacca.exe |
| File Type | PE executable, application, 32-bit |
| File Size (bytes) | 5,224,645 |
| Analysis Reason | Process started |

Libraries

File System Activity

Registry Activity

Network Activity

Process Interactions

Keyboard Monitoring

Keylogging

| Content Type | Content |
|------------------------|-------------------------------------|
| Credit Card | TR05-2005 -1100-9326 |
| Password | grafsndv |
| Social Security Number | 614 -06-6413 |
| Username | Username omitted from public report |

Supporting Static Analysis



- Recognize interesting points in time during the analysis of a malware
 - a sensitive system call has been executed
 - malware has unpacked itself
- Take a snapshot of the process memory and annotate interesting regions
- Import snapshot into IDA Pro (together with the annotations) for manual analysis

<https://user.lastline.com/malscape#/task/f7b5c2293e574d069e0a48bcd7691b16>

Supporting Static Analysis



Process Dumps ?

| Process | Timestamp | Dump Type | Snapshot Reason |
|--------------------|-----------|--------------|--|
| Analysis Subject 1 | 17 s | Process Dump | Observed API function invocation from untrusted memory regi... |
| Analysis Subject 1 | 20 s | Process Dump | Observed API function invocation from untrusted memory regi... |
| Analysis Subject 1 | 296 s | Process Dump | Analysis terminated |
| Analysis Subject 2 | 22 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 2 | 22 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 2 | 297 s | Process Dump | Analysis terminated |
| Analysis Subject 3 | 27 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 3 | 28 s | Process Dump | Observed API function invocation from untrusted memory regi... |
| Analysis Subject 3 | 30 s | Process Dump | Process terminated |
| Analysis Subject 4 | 30 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 4 | 30 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 4 | 39 s | Process Dump | Observed API function invocation from untrusted memory regi... |
| Analysis Subject 6 | 42 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 6 | 42 s | Process Dump | Observed code execution in memory region allocated by untr... |
| Analysis Subject 6 | 297 s | Process Dump | Analysis terminated |

Supporting Static Analysis



Windows Process Snapshots

This section lists process snapshots that were taken during the analysis. Please refer to the API documentation for more information on how to use these files (e.g., how to load them into IDA Pro).

For additional help, click [here](#)

| Address | Description |
|------------|--|
| 0x003df9bb | Code execution in untrusted memory region after interesting system-call |
| 0x003d3124 | Code execution in untrusted memory region after interesting system-call |
| 0x00434066 | Original entry point of c:\docume~1\miller\locals~1\temp\rarsfx0\emprpx.exe |
| 0x10001160 | Original entry point of c:\docume~1\miller\locals~1\temp\rarsfx0\emprpxres.dll |

```
mov     eax, [esp+arg 4]
```

Evasion



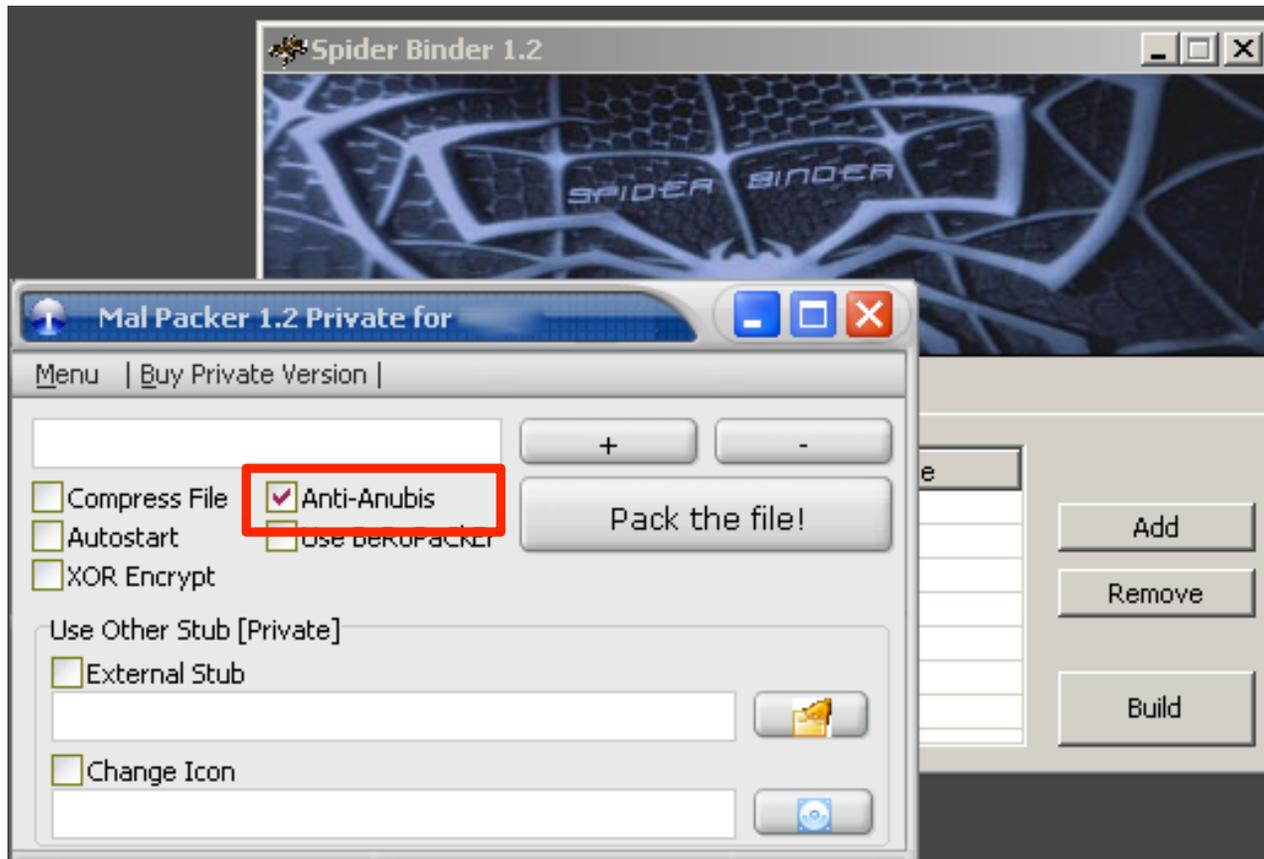
- Malware authors are not sleeping
 - they got the news that sandboxes are all the rage now
 - since the code is executed, malware authors have options ..
- Evasion
 - develop code that exhibits no malicious behavior in sandbox, but that infects the intended target
 - can be achieved in various ways

Evasion



- Malware can detect underlying runtime environment
 - differences between virtualized and bare metal environment
 - checks based on system (CPU) features
 - artifacts in the operating system
- Malware can detect signs of specific analysis environments
 - checks based on operating system artifacts (files, processes, ...)
- Malware can avoid being analyzed
 - tricks in making code run that analysis system does not see
 - wait until someone does something
 - time out analysis before any interesting behaviors are revealed
 - simple sleeps, but more sophisticated implementations possible

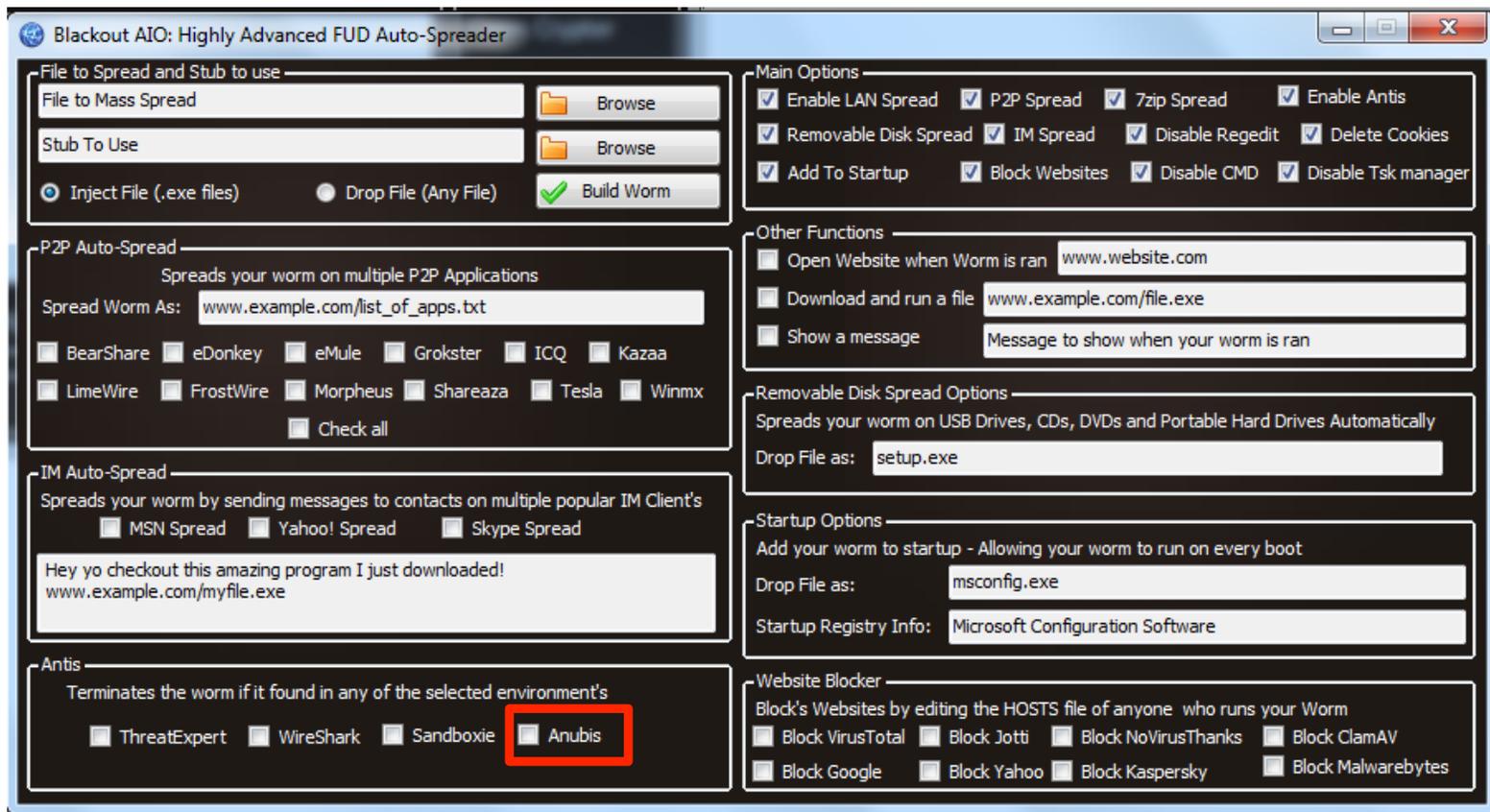
Evasion



Evasion



Evasion



Detect Runtime Environment



- Insufficient support from hardware for virtualization
 - J. Robin and C. Irvine: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor; Usenix Security Symposium, 2000
 - famous RedPill code snippet

Joanna Rutkowska

Swallowing the **Red Pill** is more or less equivalent to the following code (returns non zero when in Matrix):

```
int swallow_redpill () {
    unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();
    return (m[5]>0xd0) ? 1 : 0;
}
```

Detect Runtime Environment



- Insufficient support from hardware for virtualization
 - J. Robin and C. Irvine: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor; Usenix Security Symposium, 2000
 - famous RedPill code snippet
- hardware assisted virtualization (Intel-VT and AMD-V) helps
- but systems can still be detected due to timing differences

Detect Runtime Environment



- CPU bugs or unfaithful emulation
 - invalid opcode exception, incorrect debug exception, ...
 - later automated in: R. Paleari, L. Martignoni, G. Roglia, D. Bruschi: A fistful of red-pills: How to automatically generate procedures to detect CPU emulators; Usenix Workshop on Offensive Technologies (WOOT), 2009
 - recently, we have seen malware make use of (obscure) math instructions
- The question is ... can malware really assume that a generic virtual machine implies an automated malware analysis system?

Detect Analysis Engine



- Check Windows XP Product ID
`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductID`
- Check for specific user name, process names, hard disk names
`HKLM\SYSTEM\CURRENTCONTROLSET\SERVICES\DISK\ENUM`
- Check for unexpected loaded DLLs or Mutex names
- Check for color of background pixel
- Check of presence of 3-button mouse, keyboard layout, ...

Detect Analysis Engine



```
.text:00401E37  
.text:00401E39 loc_401E39: ; CODE XREF: .text:00401DCC↑j  
.text:00401E39 ; .text:00401DC3↑j  
.text:00401E39 mov eax, [ebp-270h]  
.text:00401E3F  
.text:00401E3F loc_401E3F: ; CODE XREF: .text:00401DD1↑j  
.text:00401E3F mov [ebp-170h], eax  
.text:00401E45  
.text:00401E45 loc_401E45: ; CODE XREF: .text:00401E2B↑j  
.text:00401E45 push dword ptr [ebp-16Ch]  
.text:00401E48 call dword ptr [ebp-34h]  
.text:00401E4E cmp dword ptr [ebp-170h], 'awmv' ;  
.text:00401E4E ; search known sandboxes'  
.text:00401E4E ; substring in registry key value  
.text:00401E4E ; vbox  
.text:00401E4E ; qemu  
.text:00401E4E ; vmla  
.text:00401E58 jz short loc_401E95  
.text:00401E5A cmp dword ptr [ebp-170h], 'xobv'  
.text:00401E64 jz short loc_401E95  
.text:00401E66 cmp dword ptr [ebp-170h], 'umeq'  
.text:00401E70 jz short loc_401E95  
.text:00401E72  
.text:00401E72 loc_401E72: ; CODE XREF: .text:00401D55↑j  
.text:00401E72 ; .text:00401D6D↑j ...  
.text:00401E72 rdtsc
```

Detect Analysis Engine



Enigma Group's Hacking Forum

HOME FORUMS EXTRA DONATIONS LOGIN REGISTER

User Info
Welcome, **Guest**. Please [login](#) or [register](#).
Did you miss your [activation email](#)?
January 31, 2013, 02:42:53 PM

Login with username, password and session length

Search: _____ Search [Advanced search](#)

News
Need a hash cracked? Use the Enigma Group [Hash Cracker](#)! It's the largest hash library on the interwebz.

Forum Stats
39005 Posts in 4766 Topics by 23414 Members
Latest Member: [young12dre](#)

Enigma Group's Hacking Forum | Hacking | Undetection Techniques | [C++] Anti-Sandbox < previous next >

Pages: [1] PRINT

Author Topic: [C++] Anti-Sandbox (Read 2487 times)

blink_212
Global Moderator
Veteran
★★★★★
Offline
Posts: 1438
• Respect: +6
EG Fanatic.

[C++] Anti-Sandbox
* on: January 28, 2011, 01:46:21 AM * 0

This is basicky a combination of my old work, and some other code have ported over from VB. I'll release the current source for what im working on somewhere else... 😊

Code: [Select](#)

```
bool detectSandbox(char* exeName, char* user){
// Used for detecting sandboxes. So far it detects
// Armbis, CO, Sunbelt, Sandboxie, Norman, WinJail.

char* str = exeName;
char * pch;

HWND snd;

if( (snd = FindWindow("SandboxieControlWndClass", NULL)) ){
return true; // Detected Sandboxie.
```

Detect Analysis Engine



Enigma Group's Hacking Forum

[HOME](#) [FORUMS](#) [EXTRA](#) [DONATIONS](#) [LOGIN](#) [REGISTER](#)

```
if( (snd = FindWindow("SandboxieControlWndClass", NULL)) ){
    return true; // Detected Sandboxie.
} else if( (pch = strstr (str,"sample")) || (user == "andy") || (user == "Andy") ){
    return true; // Detected Anubis sandbox.
} else if( (exeName == "C:\file.exe") ){
    return true; // Detected Sunbelt sandbox.
} else if( (user == "currentuser") || (user == "Currentuser") ){
    return true; // Detected Norman Sandbox.
} else if( (user == "Schmidt") || (user == "schmidt") ){
    return true; // Detected CW Sandbox.
} else if( (snd = FindWindow("Afx:400000:0", NULL)) ){
    return true; // Detected WinJail Sandbox.
} else {
    return false;
}
```

Avoid Monitoring



- Open window and wait for user to click
 - or, as recently discovered by our competitor, click multiple times ;-)
- Only do bad things after system reboots
 - system could catch the fact that malware tried to make itself persistent
- Only run before / after specific dates
- Code execution after initial call to `NtTerminateProcess`
- Bypass in-process hooks (e.g., of library functions)

Avoid Monitoring



```
SYSTEMTIME SystemTime;

DisableThreadLibraryCalls(hdll);
GetSystemTime(&SystemTime);
result = SystemTime.wMonth;
if (SystemTime.wDay + 100 * (SystemTime.wMonth + 100 * (unsigned int)SystemTime.wYear)
    >= 20120101)
{
    uint8_t* pmain_image = (uint8_t*)GetModuleHandleA(0);
    IMAGE_DOS_HEADER *pdos_header = (IMAGE_DOS_HEADER*)pmain_image;
    IMAGE_NT_HEADERS *pnt_header = \
        (IMAGE_NT_HEADERS*) (pdos_header->e_lfanew + pmain_image);
    uint8_t* entryPoint = pmain_image + pnt_header->OptionalHeader.AddressOfEntryPoint;
    result = VirtualProtect(entryPoint, 0x10u, 0x40u, &flOldProtect);

    if (result)
    {
        entryPoint[0] = 0xE9;
        entryPoint[1] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5);
        entryPoint[2] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 8);
        entryPoint[3] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 16);
        entryPoint[4] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 24);
        result = VirtualProtect((LPVOID)entryPoint, 0x10u, flOldProtect, &flOldProtect);
    }
}
```

Avoid Monitoring



Code execution after initial call to NtTerminateProcess

```
01535 ExitProcess(IN UINT uExitCode)
01536 {
01537     BASE_API_MESSAGE ApiMessage;
01538     PBASE_EXIT_PROCESS ExitProcessRequest = &ApiMessage.Data.ExitProcessRequest;
01539
01540     ASSERT(!BaseRunningInServerProcess);
01541
01542     _SEH2_TRY
01543     {
01544         /* Acquire the PEB lock */
01545         RtlAcquirePebLock();
01546
01547         /* Kill all the threads */ ← Stop monitoring here
01548         NtTerminateProcess(NULL, 0);
01549
01550         /* Unload all DLLs */ ← Interesting stuff happens here ...
01551         LdrShutdownProcess();
01552
01553         /* Notify Base Server of process termination */
01554         ExitProcessRequest->uExitCode = uExitCode;
01555         CsrClientCallServer((PCSR_API_MESSAGE)&ApiMessage,
01556                             NULL,
01557                             CSR_CREATE_API_NUMBER(BASESRV_SERVERDLL_INDEX, BasepExitProcess),
01558                             sizeof(BASE_EXIT_PROCESS));
01559
01560         /* Now do it again */
01561         NtTerminateProcess(NtCurrentProcess(), uExitCode);
```



Avoid Monitoring



Bypass in-process hooks (e.g., of library functions)

```
Address  Pointer
7FF90000 7FF80560
      7FF80560 8>MOV EDI,EDI  ← copied from 77DDEFFC
      7FF80562 - E>JMP ADVAPI32.77DDEFFE  jump to second instruction of library
                                     function
AdjustTokenPrivlages
77DDEFFC > 8>MOV EDI,EDI  ← start
77DDEFFE 5>PUSH EBP
77DDEFFF 8>MOV EBP,ESP
77DDF001 5>PUSH ESI
77DDF002 F>PUSH DWORD PTR SS:[EBP+1C]
77DDF005 F>PUSH DWORD PTR SS:[EBP+18]
77DDF008 F>PUSH DWORD PTR SS:[EBP+14]
77DDF00B F>PUSH DWORD PTR SS:[EBP+10]
77DDF00E F>PUSH DWORD PTR SS:[EBP+C]
77DDF011 F>PUSH DWORD PTR SS:[EBP+8]
77DDF014 F>CALL DWORD PTR DS:[<&ntdll.NtAdjustPrivi>; ntdll.ZwAdjustPrivilegesToken
```

Avoid Monitoring



- Sleep for a while (analysis systems have time-outs)
 - typically, a few minutes will do this
- Anti-sleep-acceleration
 - some sandboxes skip long sleeps, but malware authors have figured that out ...
- “Sleep” in a smarter way (stalling code)

Avoid Monitoring



Anti-sleep-acceleration

- introduce a race condition that involves sleeping
- Sample creates two threads
 1. `sleep() + NtTerminateProcess`
 2. copies and restarts program
 - if `ZwDelayExecution` gets patched, `NtTerminateProcess` executes before second thread is done
- Another variation
 1. `sleep() + DeleteFileW(<name>.bat)`
 2. start `<name>.bat` file

Avoid Monitoring



```
1 unsigned count, tick;
2
3 void helper() {
4     tick = GetTickCount();
5     tick++;
6     tick++;
7     tick = GetTickCount();
8 }
9
10 void delay() {
11     count=0x1;
12     do {
13         helper();
14         count++;
15     } while (count!=0xe4e1c1);
16 }
```

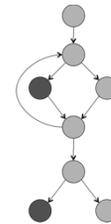
Real host - A few milliseconds
Anubis - Ten hours

Figure 1. Stalling code found in real-world malware (W32.DelfInj)

What can we do about evasion?



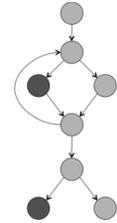
- One key evasive technique relies on checking for specific values in the environment (triggers)
 - we can randomize these values, if we know about them
 - we can detect (and bypass) triggers automatically



- Another key technique relies on timing out the sandbox
 - we can automatically profile code execution and recognize stalling

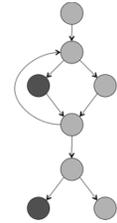


Bypassing Triggers



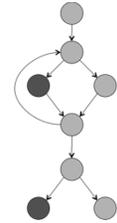
- Idea
 - explore multiple execution paths of executable under test
 - exploration is driven by monitoring how program uses certain inputs
 - system should also provide information under which circumstances a certain action is triggered
- Approach
 - track “interesting” input when it is read by the program
 - whenever a control flow decision is encountered that uses such input, two possible paths can be followed
 - save snapshot of current process and continue along first branch
 - later, revert back to stored snapshot and explore alternative branch

Bypassing Triggers



- Tracking input
 - we already know how to do this (tainting)
- Snapshots
 - we know how to find control flow decision points (branches)
 - snapshots are generated by saving the content of the process' virtual address space (of course, only used parts)
 - restoring works by overwriting current address space with stored image
- Explore alternative branch
 - restore process memory image
 - set the tainted operand (register or memory location) to a value that reverts branch condition
 - let the process continue to run

Bypassing Triggers

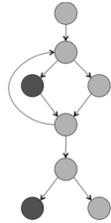


- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

```
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

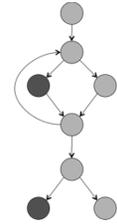
Bypassing Triggers



- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```



Bypassing Triggers

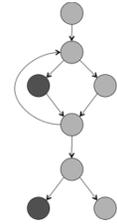


- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

```
x = 0;
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

Blue arrows in the original image point from the initial `x = 0` to the `x` in `read_input()`, from `read_input()` to `2*x` in `y = 2*x + 1`, from `y` in `check(y)` to the `magic` parameter in `check`, and from `x` in `print` to the `x` in `check`.



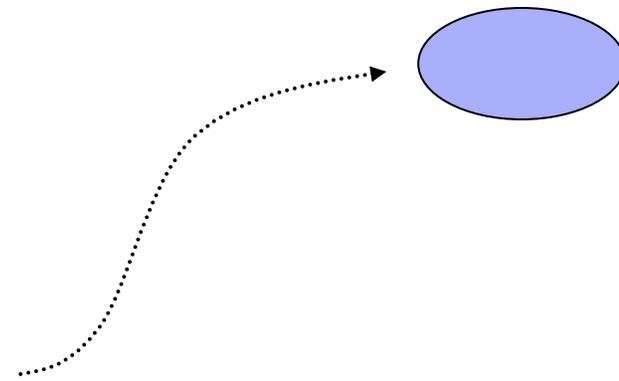
Bypassing Triggers

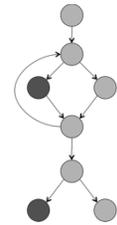


- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

```
x = 0;
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....
```

```
void check(int magic) {
    if (magic != 47)
        exit();
}
```





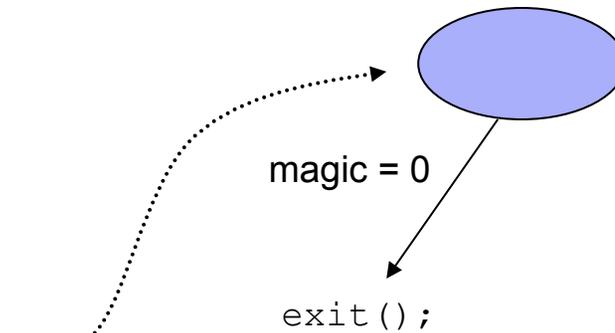
Bypassing Triggers

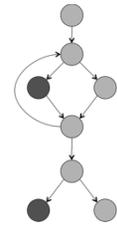


- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

```
x = 0;
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....
```

```
void check(int magic) {
    if (magic != 47)
        exit();
}
```





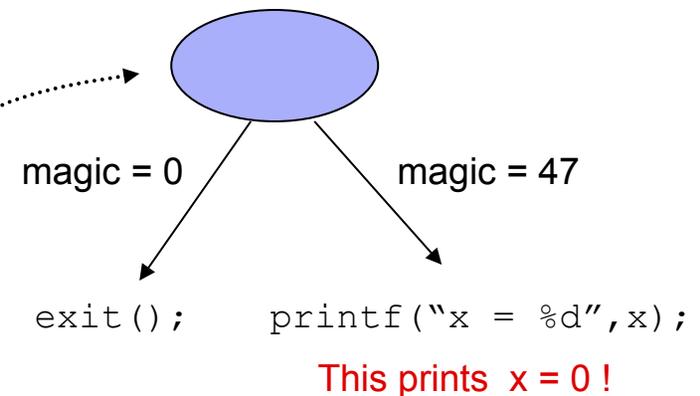
Bypassing Triggers



- Unfortunately, it is not that easy
 - when only rewriting the operand of the branch, process state can become inconsistent
 - input value might have been copied or used in previous calculations

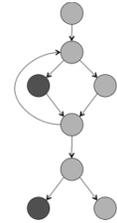
```
x = 0;
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```



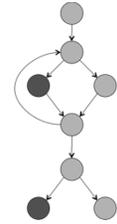
We have to remember that y depends on x, and that magic depends on y.

Bypassing Triggers



- Tracking of input must be extended
 - whenever a tainted value is copied to a new location, we must remember this relationship
 - whenever a tainted value is used as input in a calculation, we must remember the relationship between the input and the result
- **Constraint set**
 - for every operation on tainted data, a constraint is added that captures relationship between input operands and result
 - currently, we only model linear relationships
 - can be used to perform consistent memory updates when exploring alternative paths
 - provides immediate information about condition under which path is selected

Bypassing Triggers

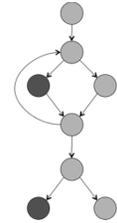


- Constraint set

```
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
.....
```

```
void check(int magic) {
    if (magic != 47)
        exit();
}
```

Bypassing Triggers



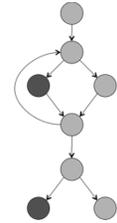
- Constraint set

```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

```
x == input
y == 2*x + 1
magic == y
```

Bypassing Triggers



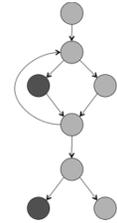
- Constraint set

```
x = 0  
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

```
x == input  
y == 2*x + 1  
magic == y  
magic == 47
```

Bypassing Triggers



- Constraint set

```
x = 0  
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

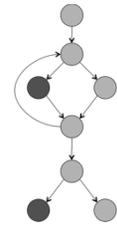
```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

```
x == input  
y == 2*x + 1  
magic == y  
magic == 47
```

solve for alternative
branch

```
y == magic == 47  
x == input == 23
```

Now, print outputs "x = 23"

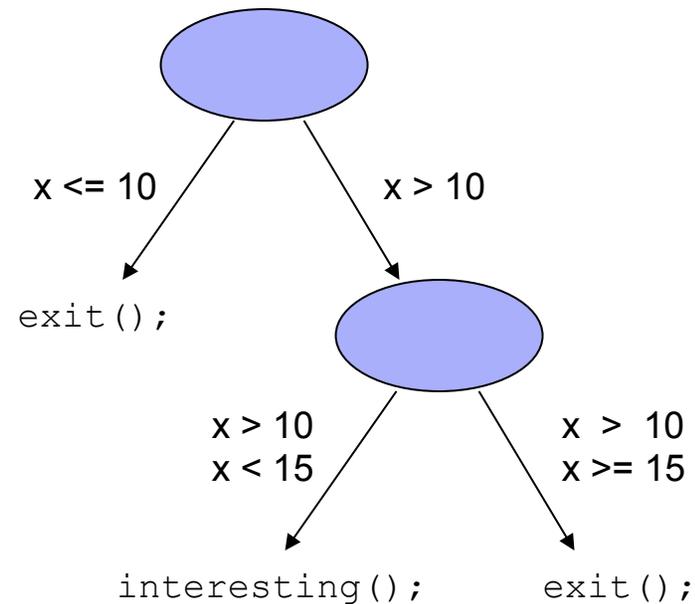


Bypassing Triggers

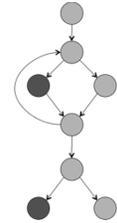


- Path constraints
 - capture effects of conditional branch operations on tainted variables
 - added to constraint set for certain path

```
x = read_input();  
  
if (x > 10)  
    if (x < 15)  
        interesting();  
  
exit();
```



Bypassing Triggers



- 308 malicious executables
 - large variety of viruses, worms, bots, Trojan horses, ...

Additional code is likely for error handling

| Interesting input sources | |
|---------------------------------|-----|
| Check for Internet connectivity | 20 |
| Check for mutex object | 116 |
| Check for existence of file | 79 |
| Check for registry entry | 74 |
| Read current time | 134 |
| Read from file | 106 |
| Read from network | 134 |

| Additional code coverage | |
|--------------------------|-----|
| none | 136 |
| 0% - 10% | 21 |
| 10% - 50% | 71 |
| 50% - 200% | 37 |
| > 200% | 43 |

Relevant behavior:
time-triggers
filename checks
bot commands

Combating Evasion

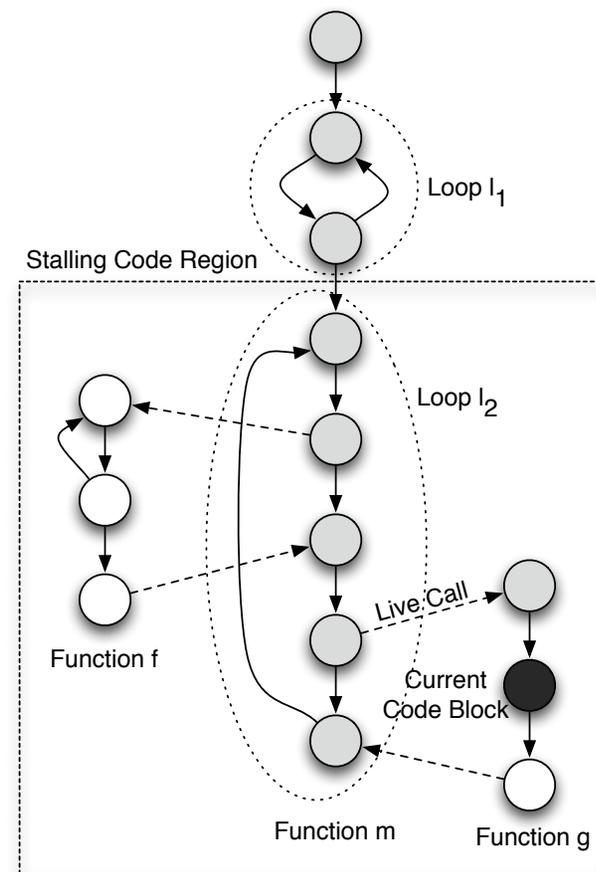


- Mitigate stalling loops
 1. detect that program does not make progress
 2. passive mode
 - find loop that is currently executing
 - reduce logging for this loop (until exit)
 3. active mode
 - when reduced logging is not sufficient
 - actively interrupt loop
- Progress checks
 - based on system calls
 - too many failures, too few, always the same, ...

Passive Mode



- Finding code blocks (white list) for which logging should be reduced
 - build dynamic control flow graph
 - run loop detection algorithm
 - identify live blocks and call edges
 - identify first (closest) *active* loop (loop still in progress)
 - mark all regions reachable from this loop



Active Mode



- Interrupt loop
 - find conditional jump that leads out of white-listed region
 - simply invert it the next time control flow passes by
- Problem
 - program might later use variables that were written by loop but that do not have the proper value and fail
- Solution
 - mark all memory locations (variables) written by loop body
 - dynamically track all variables that are marked (taint analysis)
 - whenever program uses such variable, extract slice that computes this value, run it, and plug in proper value into original execution

Experimental Results

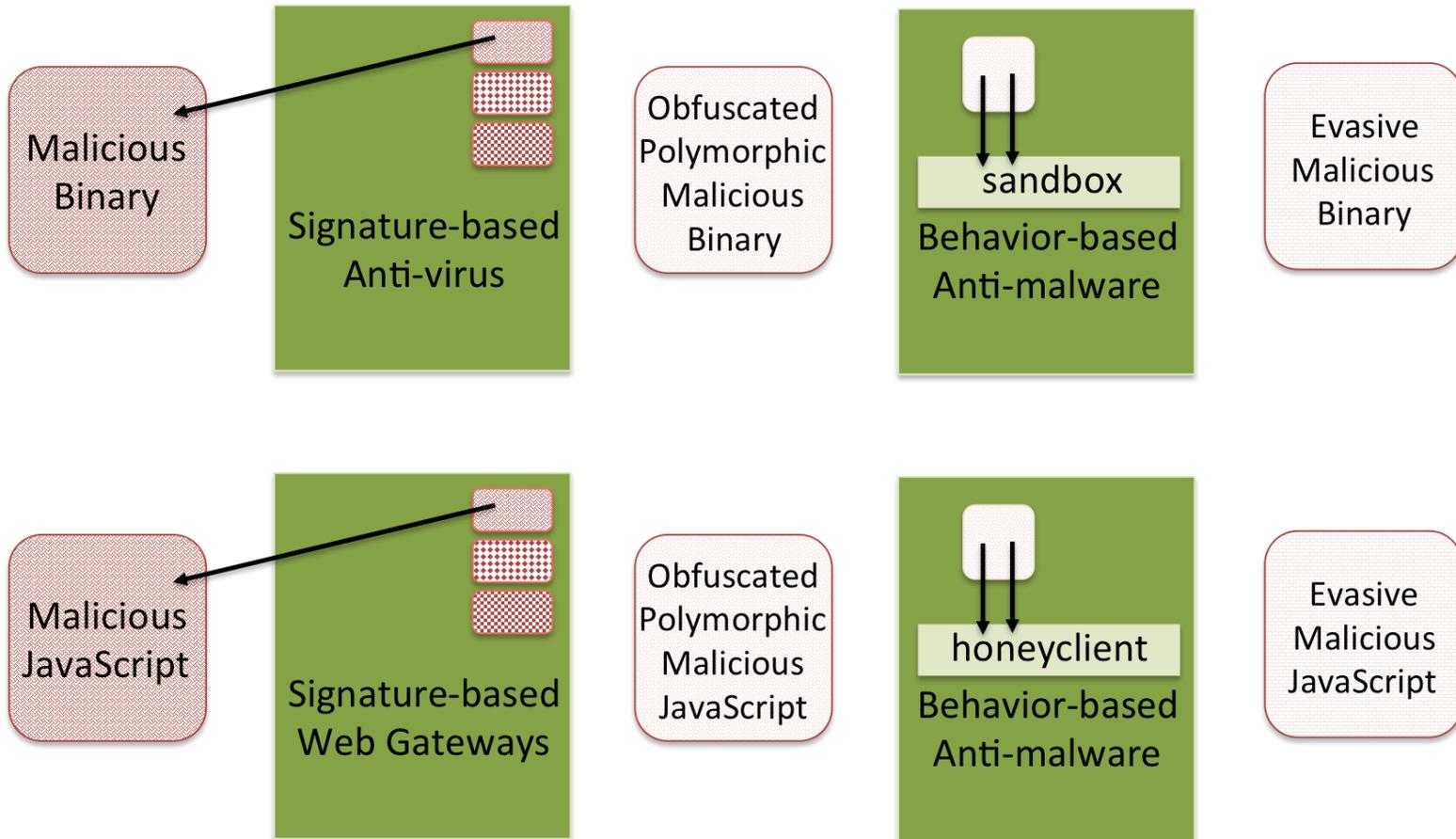


| Description | # samples | % | # AV families |
|-------------------|-----------|-------|---------------|
| <i>base run</i> | 29,102 | — | 1329 |
| <i>stalling</i> | 9,826 | 33.8% | 620 |
| <i>loop found</i> | 6,237 | 21.4% | 425 |

- 1,525 / 6,237 stalling samples reveal additional behavior
- At least 543 had obvious signs of malicious (deliberate) stalling

| Description | Passive | | | Active | | |
|---|-----------|-------|---------------|-----------|-------|---------------|
| | # samples | % | # AV families | # samples | % | # AV families |
| <i>Runs total</i> | 3,770 | — | 319 | 2,467 | — | 231 |
| <i>Added behavior (any activity)</i> | 1,003 | 26.6% | 119 | 549 | 22.3% | 105 |
| - Added file activity | 949 | 25.2% | 113 | 359 | 14.6% | 79 |
| - Added network activity | 444 | 11.8% | 52 | 108 | 4.4% | 31 |
| - Added GUI activity | 24 | 0.6% | 15 | 260 | 10.5% | 51 |
| - Added process activity | 499 | 13.2% | 55 | 90 | 3.6% | 41 |
| - Added registry activity | 561 | 14.9% | 82 | 184 | 7.5% | 52 |
| - Exception cases | 21 | 0.6% | 13 | 273 | 11.1% | 48 |
| <i>Ignored (possibly random) activity</i> | 1,447 | 38.4% | 128 | 276 | 11.2% | 72 |
| - Exception cases | 0 | 0.0% | 0 | 82 | 3.3% | 27 |
| <i>No new behavior</i> | 1,320 | 35.0% | 225 | 1,642 | 66.6% | 174 |
| - Exception cases | 0 | 0.0% | 0 | 277 | 11.2% | 63 |

Evasion in a Broader Context



Conclusions



- Visibility and fidelity are two critical factors when building successful dynamic analysis systems
 - full system emulation is a great point in the design spectrum
- Automated analysis of malicious code faces number of challenges
 - evasion is one critical challenge
- We shouldn't simply give up; it is possible to address many evasion techniques in very general ways

THANK YOU!



For more information visit www.lastline.com
or contact us at info@lastline.com.