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

- $\$ Damage
- Billions
- Millions
- Hundreds of Thousands
- Thousands
- Hundreds

- Cybervandalism
- Cybercrime
- Targeted Attacks and Cyberwarfare

Time
Malware Analysis
Malware Analysis
Malware Analysis
Malware Analysis
There is a lot of malware out there…
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

\[c:\text{sample.exe}\]

\[\text{net: 192.168.0.1} \rightarrow \text{evil.com:80}\]
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)?

Where does the data come from that is written into these files?

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

Does the program “leak” information from the secret file to the network? And if so, under which circumstances (triggers)?
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
Dynamic Analysis Approaches

Visibility

Process Emulation

Full System Emulation

System call hooking (Virtual machines)

Fidelity
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autostart</td>
<td>Registering for autostart using the Windows start menu</td>
</tr>
<tr>
<td>Evasion</td>
<td>Possibly stalling against analysis environment (loop)</td>
</tr>
<tr>
<td>File</td>
<td>Modifying executable in user-shared data directory</td>
</tr>
<tr>
<td>Signature</td>
<td>Identified trojan code</td>
</tr>
<tr>
<td>Steal</td>
<td>Keystroke logging capabilities</td>
</tr>
<tr>
<td>Stealth</td>
<td>Creating executables masquerading system files</td>
</tr>
<tr>
<td>Stealth</td>
<td>Deleting the sample after execution</td>
</tr>
</tbody>
</table>
Detecting Keyloggers

**Analysis Subject 2**

<table>
<thead>
<tr>
<th>MD5</th>
<th>21f809d9e6f33e3d3a30646360b09</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>0352725130ce8f1ee482b771e39ab3e3e2</td>
</tr>
<tr>
<td>Command Line</td>
<td>C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup\spoolsv.exe C:\Users\jchewbacca.exe</td>
</tr>
<tr>
<td>File Type</td>
<td>PE executable, application, 32-bit</td>
</tr>
<tr>
<td>File Size (bytes)</td>
<td>5224645</td>
</tr>
<tr>
<td>Analysis Reason</td>
<td>Process started</td>
</tr>
</tbody>
</table>

**Libraries**

**File System Activity**

**Registry Activity**

**Network Activity**

**Process Interactions**

**Keyboard Monitoring**

**Keylogging**

<table>
<thead>
<tr>
<th>Content Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credit Card</td>
<td>T805-3835-1100-9326</td>
</tr>
<tr>
<td>Password</td>
<td>gr4fandv</td>
</tr>
<tr>
<td>Social Security Number</td>
<td>616B-66-6413</td>
</tr>
<tr>
<td>Username</td>
<td>Username omitted from public report</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Process</th>
<th>Timestamp</th>
<th>Dump Type</th>
<th>Snapshot Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Subject 1</td>
<td>17 s</td>
<td>Process Dump</td>
<td>Observed API function invocation from untrusted memory region</td>
</tr>
<tr>
<td>Analysis Subject 1</td>
<td>20 s</td>
<td>Process Dump</td>
<td>Observed API function invocation from untrusted memory region</td>
</tr>
<tr>
<td>Analysis Subject 1</td>
<td>296 s</td>
<td>Process Dump</td>
<td>Analysis terminated</td>
</tr>
<tr>
<td>Analysis Subject 2</td>
<td>22 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 2</td>
<td>22 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 2</td>
<td>297 s</td>
<td>Process Dump</td>
<td>Analysis terminated</td>
</tr>
<tr>
<td>Analysis Subject 3</td>
<td>27 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 3</td>
<td>28 s</td>
<td>Process Dump</td>
<td>Observed API function invocation from untrusted memory region</td>
</tr>
<tr>
<td>Analysis Subject 3</td>
<td>30 s</td>
<td>Process Dump</td>
<td>Process terminated</td>
</tr>
<tr>
<td>Analysis Subject 4</td>
<td>30 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 4</td>
<td>30 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 4</td>
<td>39 s</td>
<td>Process Dump</td>
<td>Observed API function invocation from untrusted memory region</td>
</tr>
<tr>
<td>Analysis Subject 6</td>
<td>42 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 6</td>
<td>42 s</td>
<td>Process Dump</td>
<td>Observed code execution in memory region allocated by untrusted code</td>
</tr>
<tr>
<td>Analysis Subject 6</td>
<td>297 s</td>
<td>Process Dump</td>
<td>Analysis terminated</td>
</tr>
</tbody>
</table>
Supporting Static Analysis

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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x003df9bb</td>
<td>Code execution in untrusted memory region after interesting system-call</td>
</tr>
<tr>
<td>0x003d3124</td>
<td>Code execution in untrusted memory region after interesting system-call</td>
</tr>
<tr>
<td>0x0034066</td>
<td>Original entry point of c:\document<del>1\miller\locals</del>1\temp\rasfx0\emprx.exe</td>
</tr>
<tr>
<td>0x1001160</td>
<td>Original entry point of c:\document<del>1\miller\locals</del>1\temp\rasfx0\emprxres.dll</td>
</tr>
</tbody>
</table>

MOV EAX, [ESP+ARQ 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
  - famous RedPill code snippet

```c
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
  - 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, …
  - 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
Detect Analysis Engine

```c
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(hdl);
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, &f1OldProtect);

    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, f1OldProtect, &f1OldProtect);
    
}
Avoid Monitoring

Code execution after initial call to \texttt{NtTerminateProcess}

\begin{verbatim}
01535  ExitProcess(IN UINT uExitCode)
01536  {
01537      BASE_API_MESSAGE ApiMessage;
01539
01540      ASSERT(!BaseRunningInServerProcess);
01541
01542      __SEH2_TRY
01543      {
01544          /* Acquire the PEB lock */
01545              RtlAcquirePebLock();
01546
01547          /* Kill all the threads */
01548              NtTerminateProcess(NULL, 0);
01549
01550          /* Unload all DLLs */
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);
\end{verbatim}

Stop monitoring here

Interesting stuff happens here ...
Avoid Monitoring

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FF90000</td>
<td>7FF80560</td>
</tr>
</tbody>
</table>

7FF80560  8>MOV EDI,EDI  <- copied from 77DDEFFC
7FF80562  -E>JMP ADVAPI32.77DDEFFE

AdjustTokenPrivileges
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

```c
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!=0xe1cl);
16 }
```

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

Real host - A few milliseconds
Anubis - Ten hours
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

```c
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

```c
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

```c
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

```c
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

```c
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

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

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

This prints \( x = 0 \)!

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

```c
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

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

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

\[ x = \text{input} \]
\[ y = 2 \times x + 1 \]
\[ \text{magic} = y \]
Bypassing Triggers

• Constraint set

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

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

```plaintext
x = 0
x == input
y = 2*x + 1
magic == y
magic == 47
```
Bypassing Triggers

- Constraint set

```c
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

```plaintext
x = read_input();

if (x > 10)
  if (x < 15)
    interesting();

exit();
```
Bypassing Triggers

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

<table>
<thead>
<tr>
<th>Interesting input sources</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Internet connectivity</td>
<td>20</td>
</tr>
<tr>
<td>Check for mutex object</td>
<td>116</td>
</tr>
<tr>
<td>Check for existence of file</td>
<td>79</td>
</tr>
<tr>
<td>Check for registry entry</td>
<td>74</td>
</tr>
<tr>
<td>Read current time</td>
<td>134</td>
</tr>
<tr>
<td>Read from file</td>
<td>106</td>
</tr>
<tr>
<td>Read from network</td>
<td>134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional code coverage</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>136</td>
</tr>
<tr>
<td>0% - 10%</td>
<td>21</td>
</tr>
<tr>
<td>10% - 50%</td>
<td>71</td>
</tr>
<tr>
<td>50% - 200%</td>
<td>37</td>
</tr>
<tr>
<td>&gt; 200%</td>
<td>43</td>
</tr>
</tbody>
</table>

Additional code is likely for error handling

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

### Table: Experimental Results

<table>
<thead>
<tr>
<th>Description</th>
<th># samples</th>
<th>%</th>
<th># AV families</th>
</tr>
</thead>
<tbody>
<tr>
<td>base run</td>
<td>29,102</td>
<td></td>
<td>1329</td>
</tr>
<tr>
<td>stalling</td>
<td>9,826</td>
<td>33.8%</td>
<td>620</td>
</tr>
<tr>
<td>loop found</td>
<td>6,237</td>
<td>21.4%</td>
<td>425</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Passive</th>
<th></th>
<th>Active</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># samples</td>
<td># AV families</td>
<td># samples</td>
<td># AV families</td>
</tr>
<tr>
<td>Runs total</td>
<td>3,770</td>
<td>319</td>
<td>2,467</td>
<td>231</td>
</tr>
<tr>
<td>Added behavior (any activity)</td>
<td>1,003</td>
<td>119</td>
<td>549</td>
<td>105</td>
</tr>
<tr>
<td>- Added file activity</td>
<td>949</td>
<td>113</td>
<td>359</td>
<td>79</td>
</tr>
<tr>
<td>- Added network activity</td>
<td>444</td>
<td>52</td>
<td>108</td>
<td>31</td>
</tr>
<tr>
<td>- Added GUI activity</td>
<td>24</td>
<td>15</td>
<td>260</td>
<td>51</td>
</tr>
<tr>
<td>- Added process activity</td>
<td>499</td>
<td>55</td>
<td>90</td>
<td>41</td>
</tr>
<tr>
<td>- Added registry activity</td>
<td>561</td>
<td>82</td>
<td>184</td>
<td>52</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>21</td>
<td>13</td>
<td>273</td>
<td>48</td>
</tr>
<tr>
<td>Ignored (possibly random) activity</td>
<td>1,447</td>
<td>128</td>
<td>276</td>
<td>72</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>0</td>
<td>0</td>
<td>82</td>
<td>27</td>
</tr>
<tr>
<td>No new behavior</td>
<td>1,320</td>
<td>225</td>
<td>1,642</td>
<td>174</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>0</td>
<td>0</td>
<td>277</td>
<td>63</td>
</tr>
</tbody>
</table>
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.