UNDERSTANDING AND FIGHTING EVASIVE MALWARE

Christopher Kruegel
Lastline Inc. and UC Santa Barbara
Who am I?

► Professor in Computer Science at UC Santa Barbara
  ► 100+ systems security papers in academic conferences
  ► started malware research in about 2004
  ► built and released practical systems (Anubis, Wepawet, …)

► Co-founder and Chief Scientist at Lastline, Inc.
  ► Lastline offers protection against zero-day threats and advanced malware
  ► venue to commercialize our academic research
What are we talking about?

- Evolution of malicious code and automated malware analysis
- 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 common evasion approaches
Evolution of Malware

- **Cybercrime**: $$$
- **Cybervandalism**: @!
- **Targeted Attacks and Cyberwarfare**: !!!

- **$ Damage**: Billions
- **Millions**: Hundreds of Thousands
- **Thousands**: Hundreds
Malware Analysis
Malware Analysis

---

### Resource Usage

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Usage</td>
<td>2%</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>98140K</td>
</tr>
</tbody>
</table>

---

### Process Information

- **Process Name**: myProcess.exe
- **User**: user
- **Priority**: Low
- **CPU Usage**: 2%
- **Memory Usage**: 98140K/11870C

---

### Windows Task Manager

- **Image Name**: myProcess.exe
- **User Name**: user
- **CPU**: 00
- **Memory Usage**: 98140K/11870C

---

### Memory Dump

- **Address**: 0x4D000000
- **Value**: 0x00000000

---

### Debugging Tools

- **OllyDbg**: Used for debugging the malware.
- **Windows Task Manager**: Used for resource monitoring.

---

### Additional Information

- **RSAC Conference Europe 2013**: Event where this document was generated.
- **Lastline**: Malware analysis tool used in the research.
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
Automated Malware Analysis

Not all sandboxes are equal!

It is easy to build a sandbox, it is hard to build an effective sandbox!

Lawrence Orans
“The Executive's Guide to Cyberthreats”
(Gartner Symposium, October 2013)
Automated Malware Analysis

► Ask your vendor questions about their sandbox
  ► what files are supported (executables, documents, more …)
  ► how effective is classification of malicious behaviors
  ► how effective is sandbox in eliciting behaviors (evasion!)
Automated Malware Analysis

► Anubis: ANalyzing Unknown Binaries
(dynamic malware analysis environment)
Automated Malware Analysis

- Anubis: ANalyzing Unknown BInarieS (dynamic malware analysis environment)
  - based on system/CPU emulator (Qemu)
  - can see every instruction!
  - monitors system activity from the outside (stealthier)
  - requires mechanisms to handle semantic gap
  - general platform on which additional components can be built
  - supports dynamic data flow analysis (taint tracking)
Automated Malware Analysis
VM Engine versus CPU Emulation

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,0xffffffffc0(%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 Data Flow Analysis

► Data tainting
  ► if any byte of any input value is tainted, then all bytes of the output are tainted
  (e.g., add %eax, %ebx)

► Address tainting
  ► in addition, if any byte of any input value that is involved in the address computation of a source memory operand is tainted, then the output is tainted
  (e.g., mov %eax, (%ecx, %ebx, 2))
Evasions

Security in knowledge
Evasion

Malware authors are not stupid
► 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 clicks 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

Joanna Rutkowska

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

```c
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

.loc_401E39:          ; CODE XREF: .text:00401DCC
          mov    eax, [ebp-270h]

.loc_401E3F:          ; CODE XREF: .text:00401DD1
          mov    [ebp-170h], eax

.loc_401E45:          ; CODE XREF: .text:00401E2B
          push   dword ptr [ebp-16Ch]
          call   dword ptr [ebp-34h]
          cmp    dword ptr [ebp-170h], 'awm'
          ; search known sandbox names
          ; substring in registry key value
            vbox
          ; qemu
            vmwa
          jz      short loc_401E95
          cmp    dword ptr [ebp-170h], 'xobv'
          jz      short loc_401E95
          cmp    dword ptr [ebp-170h], 'umeq'
          jz      short loc_401E95

.loc_401E72:          ; CODE XREF: .text:00401D55
          rdtsc
Detect Analysis Engine
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

► 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
SYSTEMTIME SystemTime;

DisableThreadLibraryCalls(hdl1);
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

- Escape 32-bit address space (on 64-bit Windows)
  - 32-bit Windows processes actually live in 64-bit address space
  - code can modify segment register to point outside “normal” 32-bit address space
  - Windows uses this trick to call 64-bit system calls from 32-bit code (basically, 32-bit system calls are trampolines to 64-bit versions)
  - malware uses this to bypass systems that monitor 32-bit addresses of system calls
Avoid Monitoring

► Sleep for a while (analysis systems have time-outs)
  ► typically, a few minutes will do this

► “Sleep” in a smarter way (stalling code – example on the next slide)
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!=0xe4e1c1);
16 }
```

Figure 1. Stalling code found in real-world malware (W32.DelfInj)
Handling Evasions
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 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
void check(int magic) {
    if (magic != 47)
        exit();
}
```
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();
}
```
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
int x = 0;

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

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

magic = 0
exit();
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
void check(int magic) {
  if (magic != 47)
    exit();
}
```

This prints \( x = 0 \)!

We have to remember that \( y \) depends on \( x \), and that \( \text{magic} \) 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
  - can be used to perform consistent memory updates when exploring alternative paths
  - provides immediate information about condition under which path is selected
Constraint Set

\[
\begin{align*}
x &= \text{read_input}(); \\
y &= 2 \times x + 1; \\
\text{check}(y); \\
\text{print}("x = \%d, \ x"); \\
\end{align*}
\]

\[
\begin{align*}
\text{void check(int magic) \{} \\
\quad \text{if (magic \!= 47)} \\
\quad \quad \text{exit();} \\
\text{\}}
\end{align*}
\]
Bypassing Triggers

**Constraint Set**

\[ x = 0 \]

\[
\begin{align*}
x &= \text{read\_input}(); \\
y &= 2 \cdot x + 1 \\
\text{check}(y) \\
\text{print}(\text{"x = %d, x"}); \\
\ldots
\end{align*}
\]

\[
\text{void check}(\text{int magic}) \{ \\
\quad \text{if (magic != 47)} \\
\quad \quad \text{exit}(); \\
\}
\]

\[ x == \text{input} \]

\[ y == 2 \cdot x + 1 \]

\[ \text{magic} == y \]
Constraint Set

\[
x = 0 \\
x = \text{read\_input}(); \\
y = 2 \times x + 1; \\
\text{check}(y); \\
\text{print}("x = \%d, x"); \\
\ldots \\
\text{void check(int magic) \{ \\
\qquad \text{if (magic \(!= 47) \\
\qquad \quad \text{exit}(); \\
\qquad \}} \\
\]

\[
x = \text{input} \\
y = 2 \times x + 1 \\
magic = y \\
magic = 47\]
Bypassing Triggers

Constraint Set

\[ \begin{align*}
    x &= 0 \\
    x &= \text{read_input}() \\
    y &= 2x + 1 \\
    \text{check}(y) \\
    \text{print}(\text{"x = x"}) \\
    \text{...}
\end{align*} \]

\begin{verbatim}
    void check(int magic) {
        if (magic != 47) 
            exit();
    }
\end{verbatim}

\[ \begin{align*}
    x &= \text{input} \\
    y &= 2x + 1 \\
    \text{magic} &= y \\
    \text{magic} &= 47 \\
\end{align*} \]

solve for alternative branch

\[ \begin{align*}
    y &= \text{magic} = 47 \\
    x &= \text{input} = 23 \\
\end{align*} \]

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

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

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

### Interesting input sources

<table>
<thead>
<tr>
<th>Source</th>
<th>Count</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>

### Additional code coverage

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Count</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

- 1,552 / 6,237 stalling samples reveal additional behavior
- At least 543 had obvious signs of malicious (deliberate) stalling

### 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>

### Additional Analysis:

<table>
<thead>
<tr>
<th>Description</th>
<th>Passive</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># samples</td>
<td>%</td>
</tr>
<tr>
<td>Runs total</td>
<td>3,770</td>
<td>—</td>
</tr>
<tr>
<td>Added behavior (any activity)</td>
<td>1,003</td>
<td>26.6%</td>
</tr>
<tr>
<td>- Added file activity</td>
<td>949</td>
<td>25.2%</td>
</tr>
<tr>
<td>- Added network activity</td>
<td>444</td>
<td>11.8%</td>
</tr>
<tr>
<td>- Added GUI activity</td>
<td>24</td>
<td>0.6%</td>
</tr>
<tr>
<td>- Added process activity</td>
<td>499</td>
<td>13.2%</td>
</tr>
<tr>
<td>- Added registry activity</td>
<td>561</td>
<td>14.9%</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>21</td>
<td>0.6%</td>
</tr>
<tr>
<td>Ignored (possibly random) activity</td>
<td>1,447</td>
<td>38.4%</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>No new behavior</td>
<td>1,320</td>
<td>35.0%</td>
</tr>
<tr>
<td>- Exception cases</td>
<td>0</td>
<td>0.0%</td>
</tr>
</tbody>
</table>
Malware is key component in many security threats on the Internet

Automated analysis of malicious code faces number of challenges
  - evasion is one critical challenge!

Types of evasion
  - detect analysis environment
  - detect analysis system
  - avoid analysis

We shouldn’t simply give up, it is possible to address certain techniques in very general ways
Thank you!

Christopher Kruegel
Lastline Inc. / UCSB

chris@lastline.com
http://www.lastline.com