RSAConference2015
San Francisco | April 20-24 | Moscone Center


SESSION ID: CRWD-T08

# Evasive Malware
# Exposed and Deconstructed

**Christopher Kruegel**

Chief Scientist
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?

- What is evasion and why should I care?
- Evasion as a significant threat to automated malware analysis
  - detect analysis environment
  - avoid being seen by automated analysis
- Improvements to analysis systems
  - automate defenses against classes of evasion approaches

# Evasive Malware

## e·vade

/əˈvād/

*verb*

gerund or present participle: **evading**

escape or avoid, especially by cleverness or trickery.
"friends helped him to evade capture for a time"
*synonyms:* elude, avoid, dodge, escape (from), steer clear of, keep at arm's length, sidestep; More

- (of an abstract thing) elude (someone).
  "sleep still evaded her"
- avoid giving a direct answer to (a question).
  "he denied evading the question"
  *synonyms:* avoid, dodge, sidestep, bypass, shirk, hedge, skirt around, fudge, be evasive about; *informal* duck
  "he evaded the question"

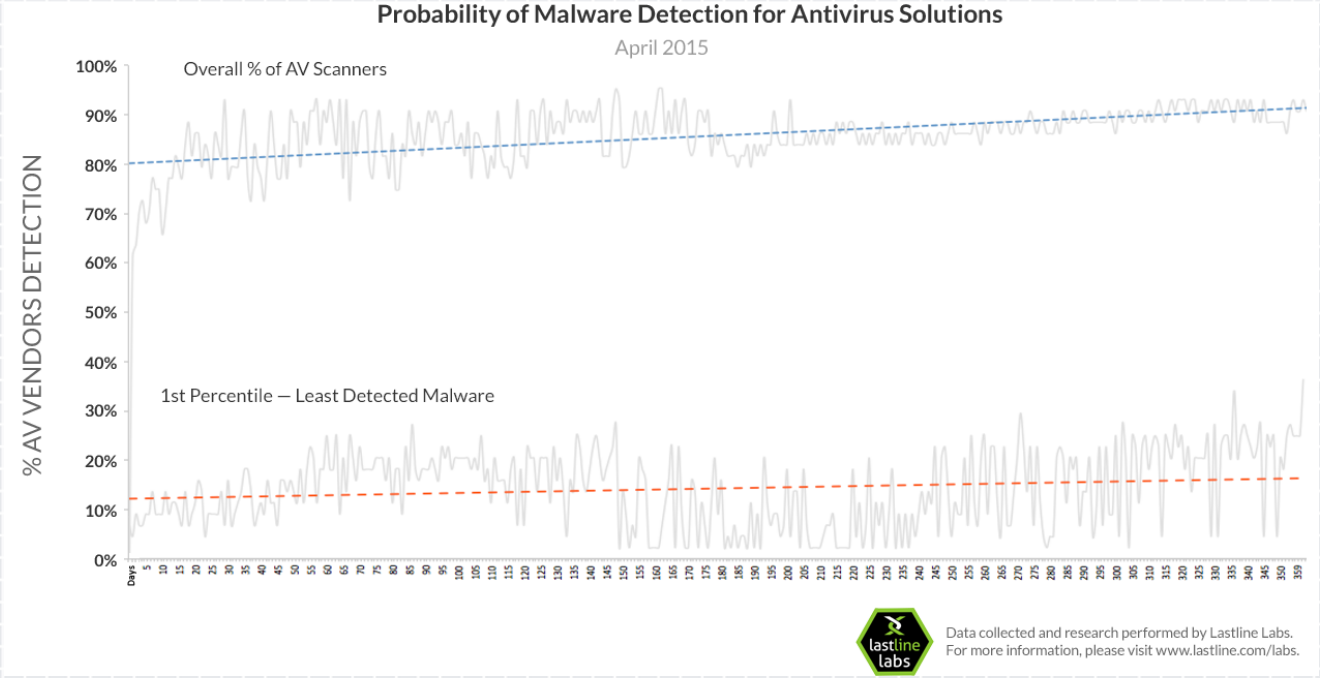
# Evasive Malware

- Attackers have always tried to escape or avoid detection
  - as we build new defenses, attackers try to bypass them
  - result is the arms race in computer security
- Evasion has been used by malware authors for decades
  - initially, evasion was targeting anti-virus (AV) solutions
  - AV systems relied heavily on signatures and static analysis

# Evading Static Analysis

- Make (relevant) code unavailable
  - packing / encrypting
  - delay inclusion of code (run-time code loading or generation)
- Exploit differences in the parsing capabilities
  - parsing of executable (the target is the OS)
  - parsing of document (the target is, for example, Office application)
- Make operations dependent on values known only at run-time
  - table lookups based on user-provided input

# Evading Static Analysis




64% of AV scanners fail to identify “1% hardest to detect” malware after 1 yr.

# Evading Static Analysis

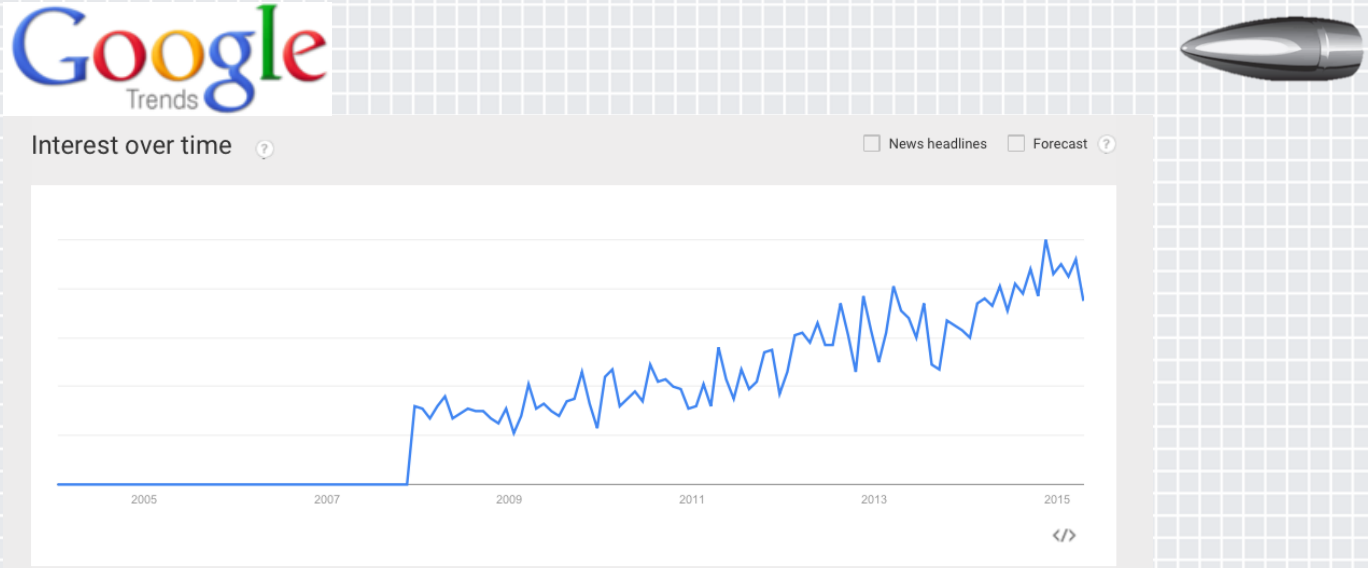
# Dynamic Malware Analysis

# Dynamic Malware Analysis

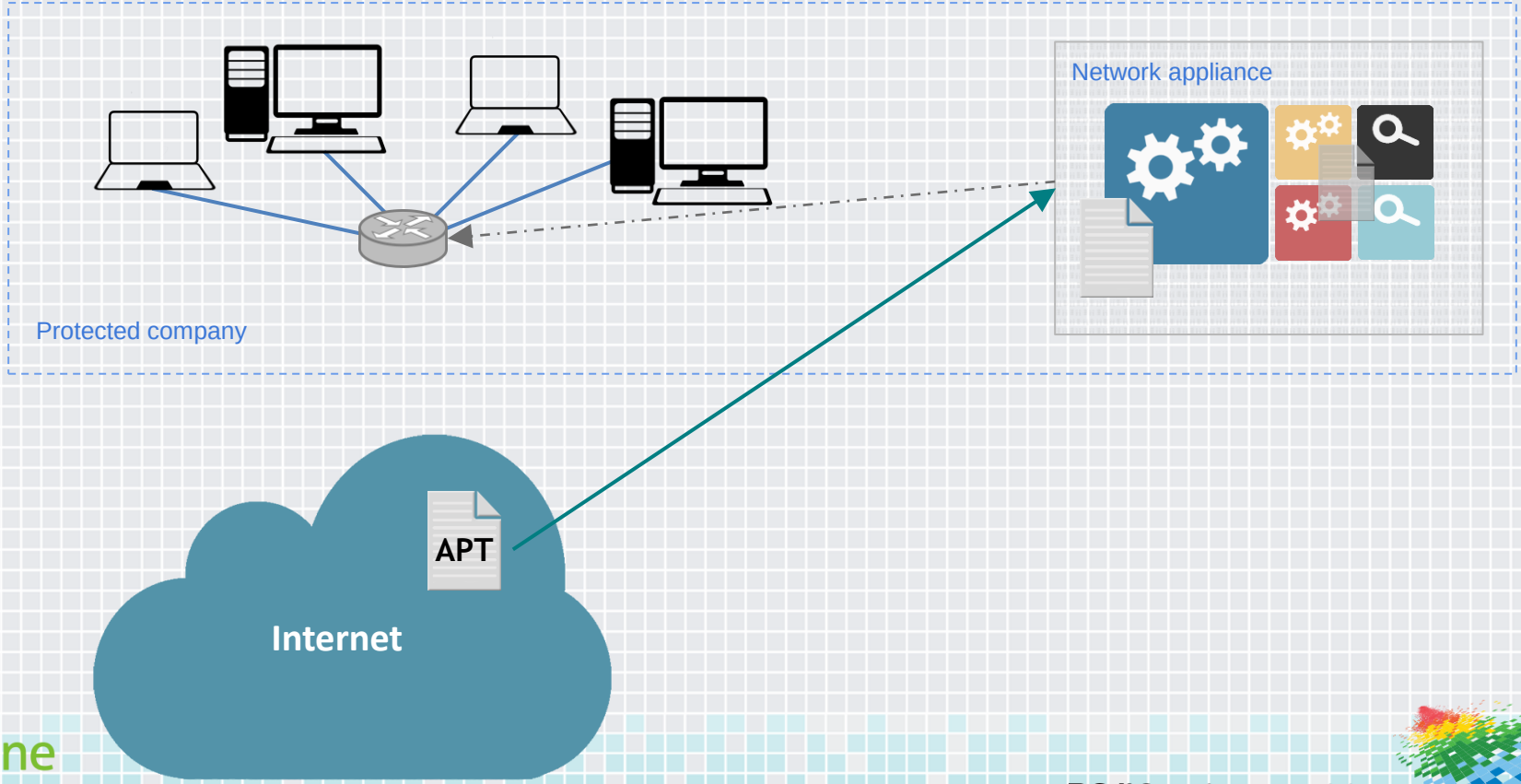
- Also known as malware analysis sandbox
- Implemented as instrumented execution environment
  - run program and observe its activity
  - make determination whether code is malicious or not
- Sandboxes are great!
  - can handle zero day threats (signature-less defense)
  - automate tasks done by human analysts and reverse engineers

# Dynamic Malware Analysis

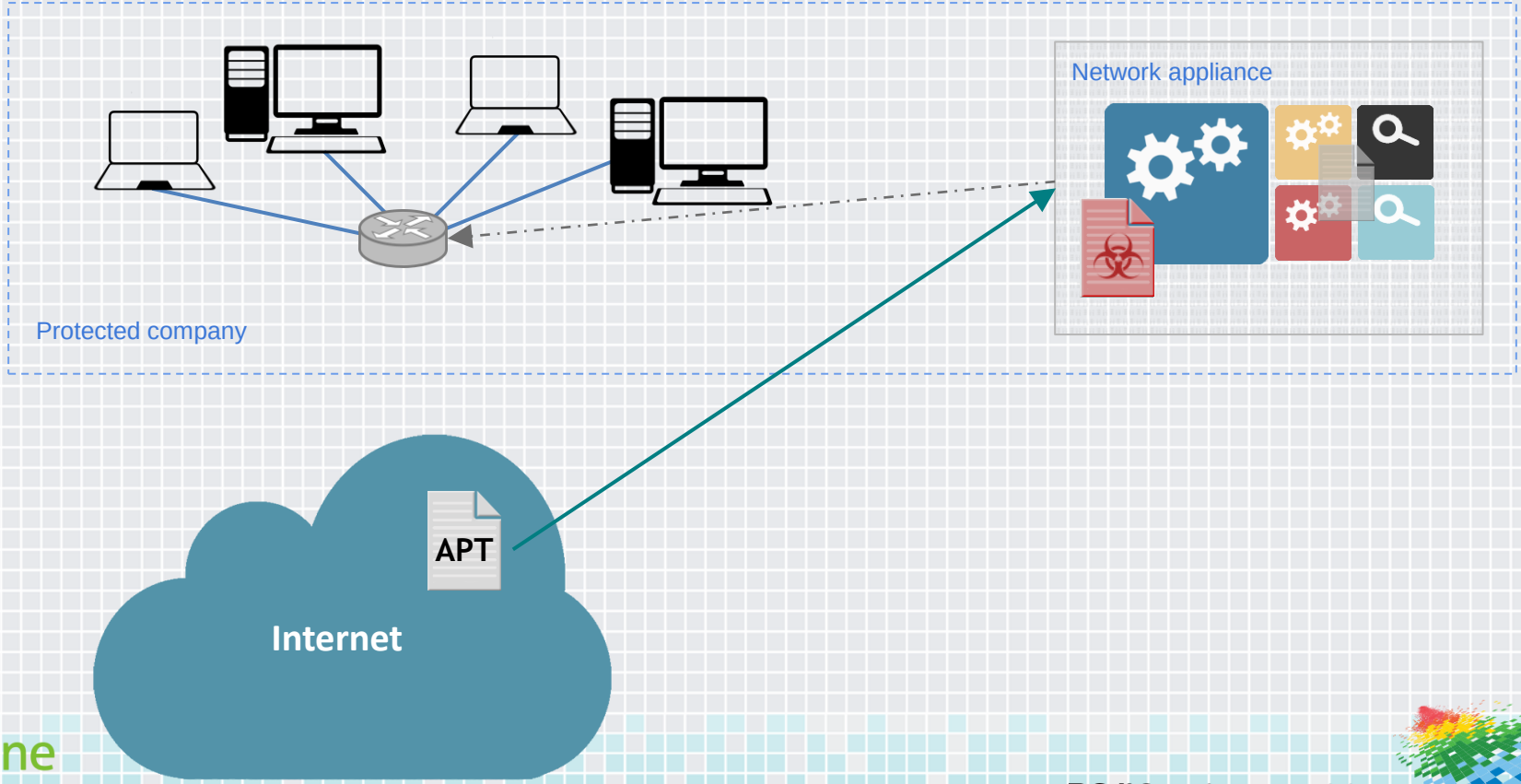
- Recently emerged as a new silver bullet in security



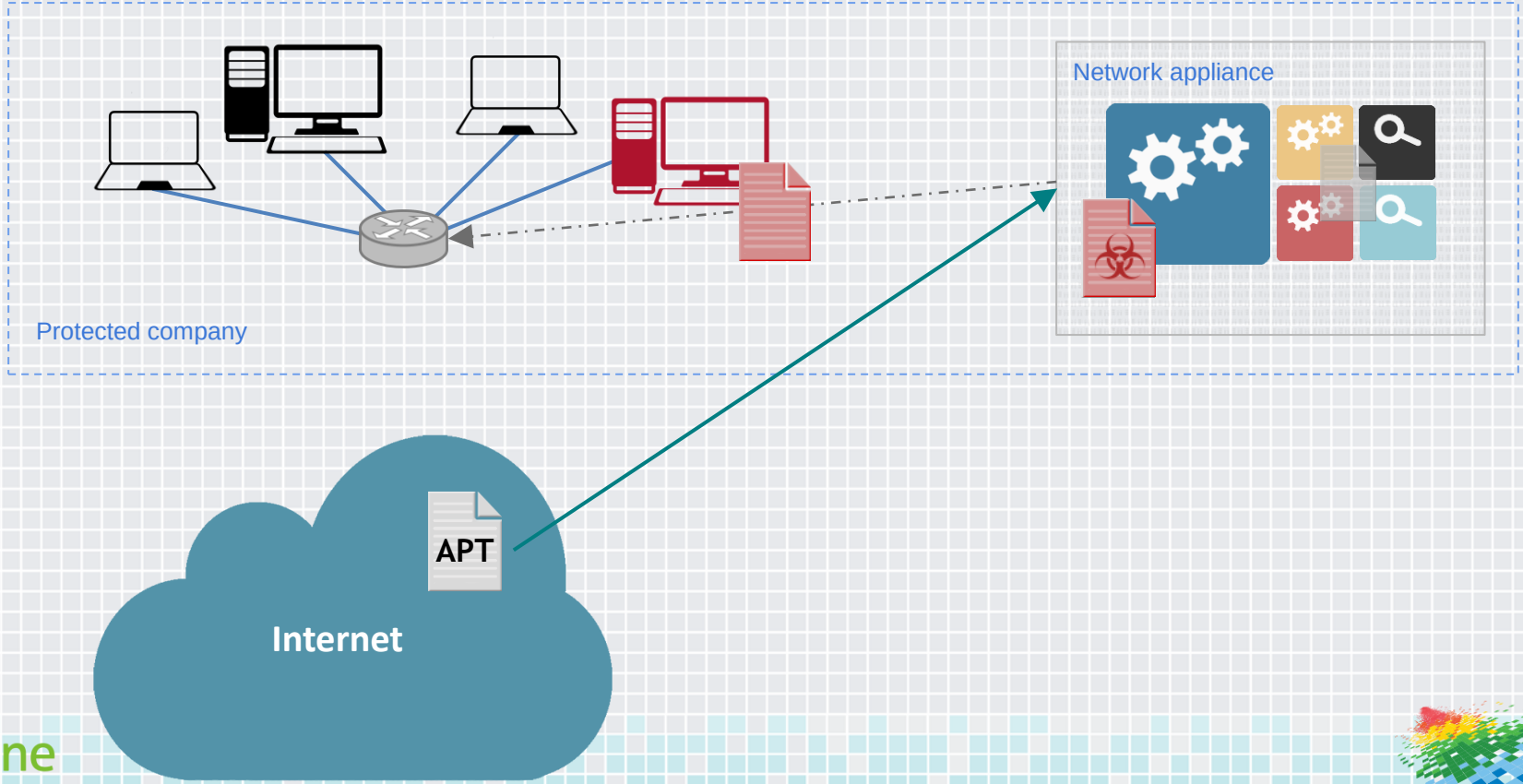
# Dynamic Malware Analysis



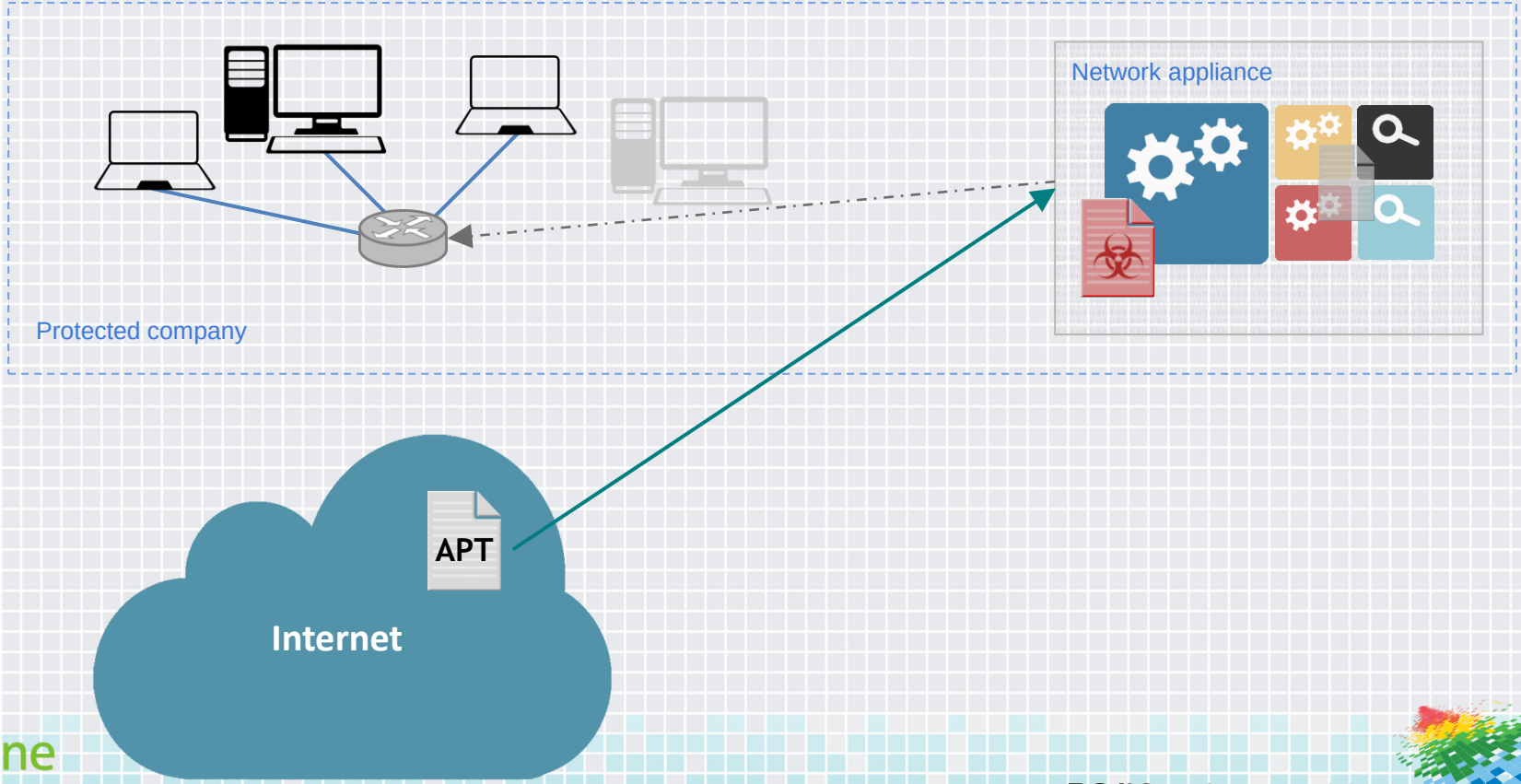
# Dynamic Malware Analysis



# Dynamic Malware Analysis



# Dynamic Malware Analysis

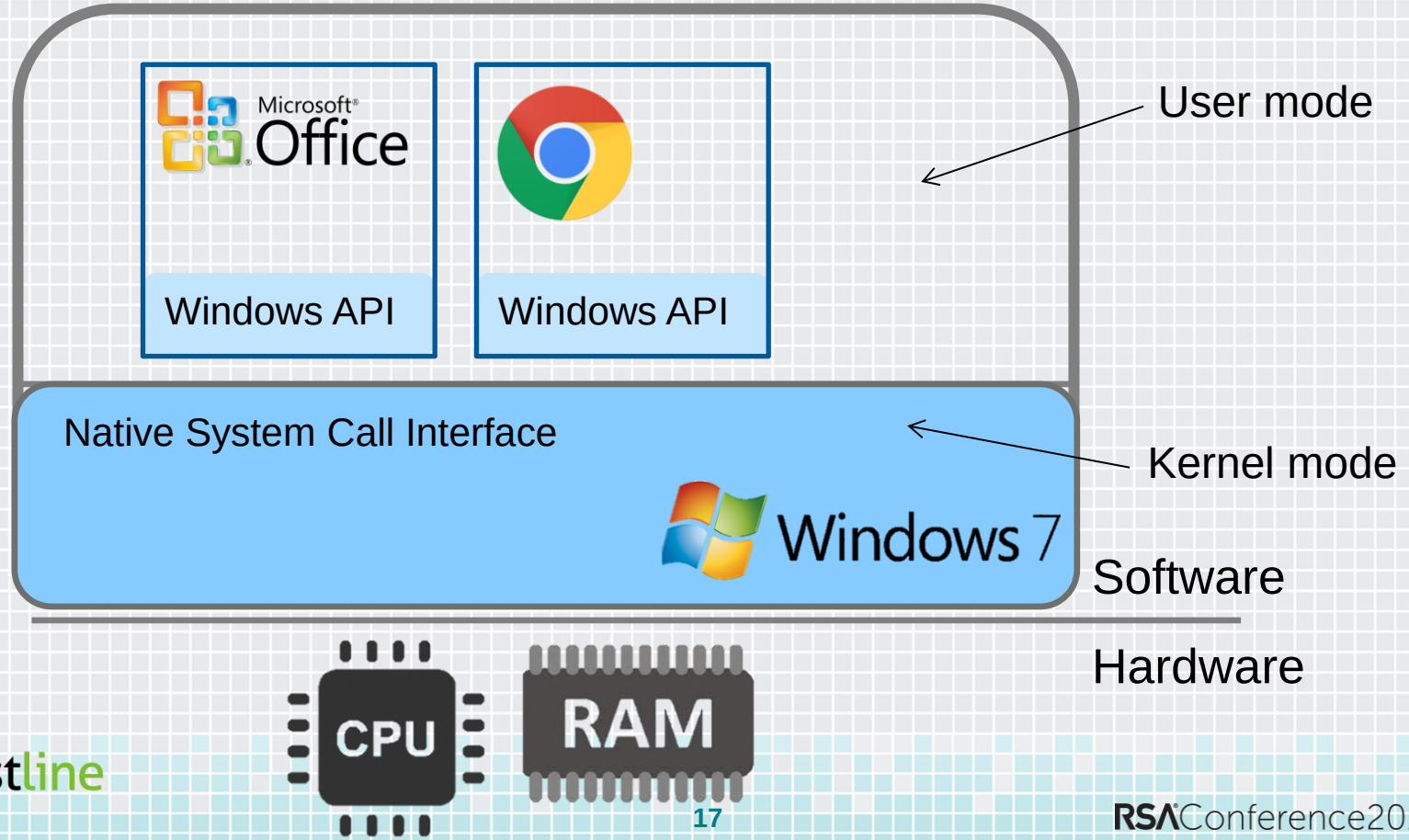


# Not All Sandboxes Are Equal

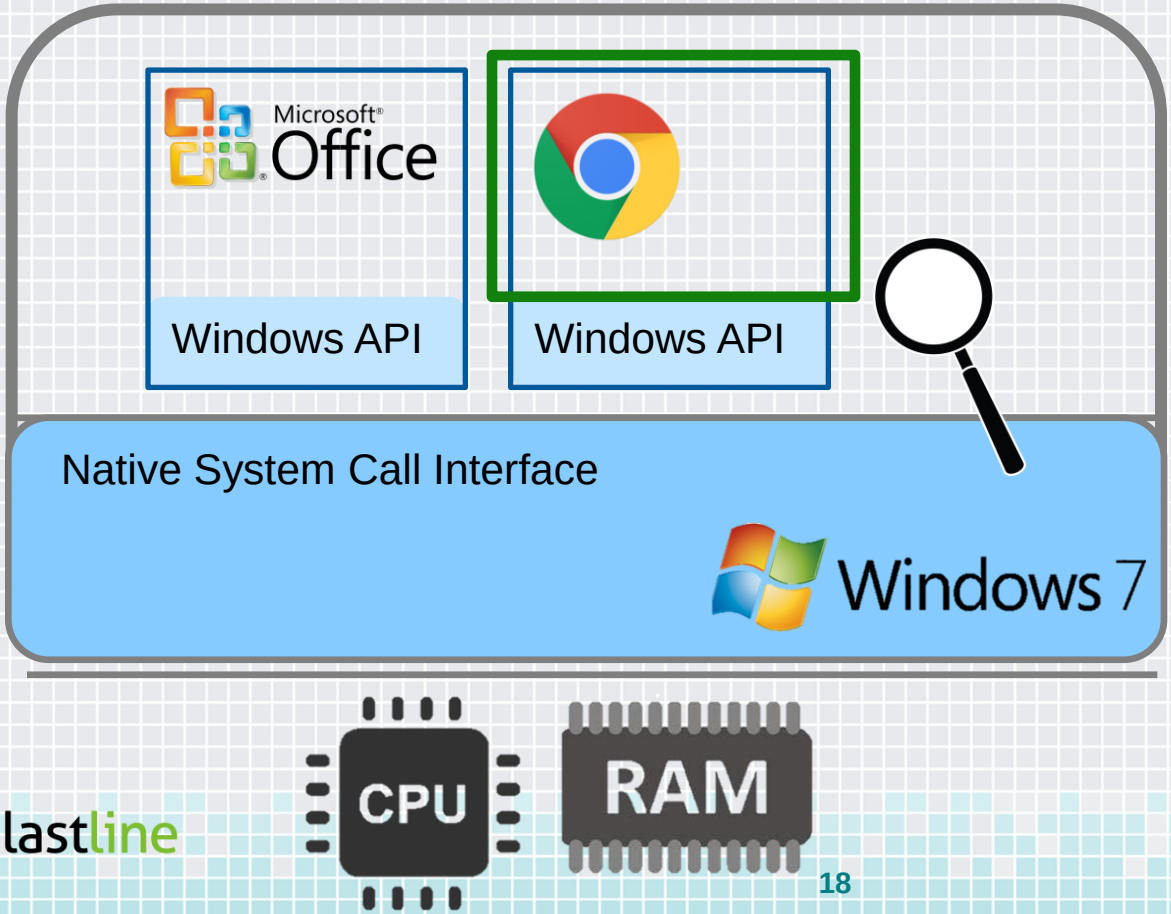
*"It is easy to build a sandbox,*
*it is hard to build an effective sandbox!"*

Lawrence Orans
"The Executive's Guide to Cyberthreats"
(Gartner Symposium, October 2013)

# Sandbox Designs



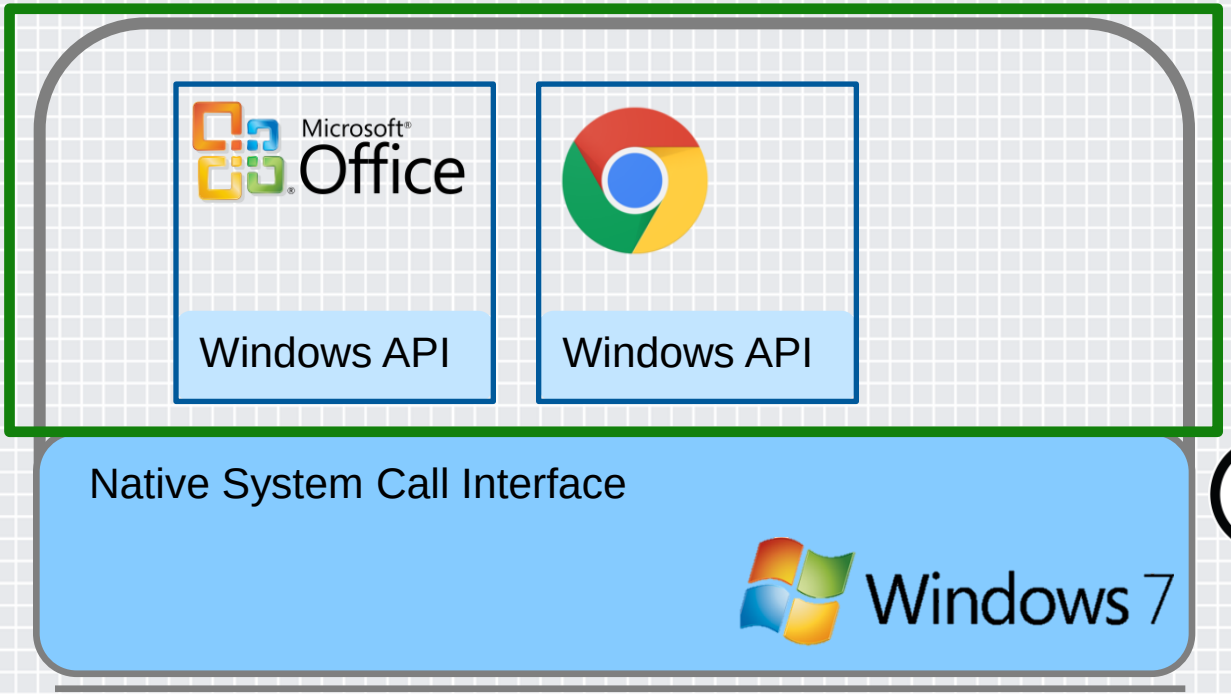
# Sandbox Designs




Hook API functions

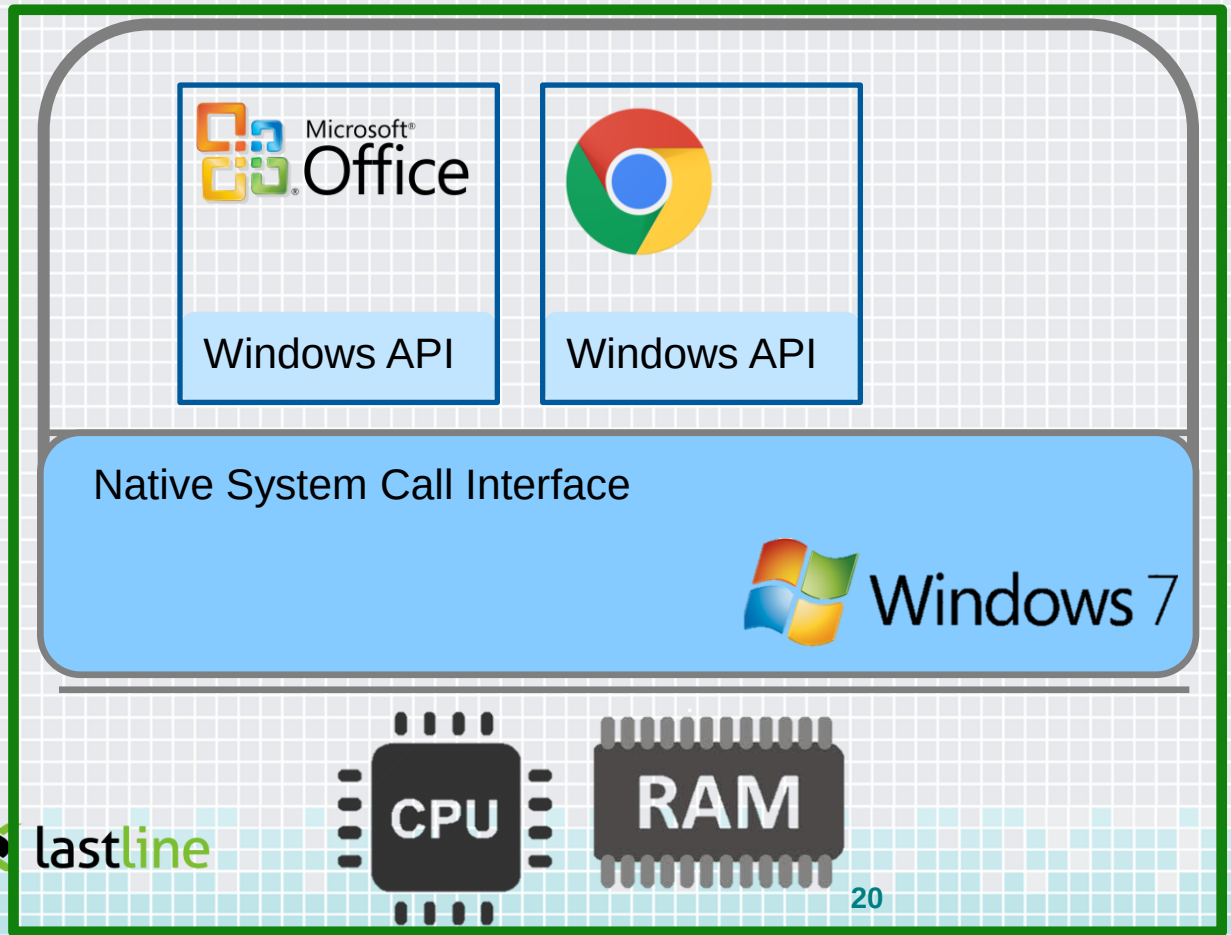
- monitor interactions with OS
- easy to implement
- needs process modifications
- no kernel visibility

# Sandbox Designs




Hook system calls
- monitor interactions with OS
- easy to implement
- minimal kernel visibility

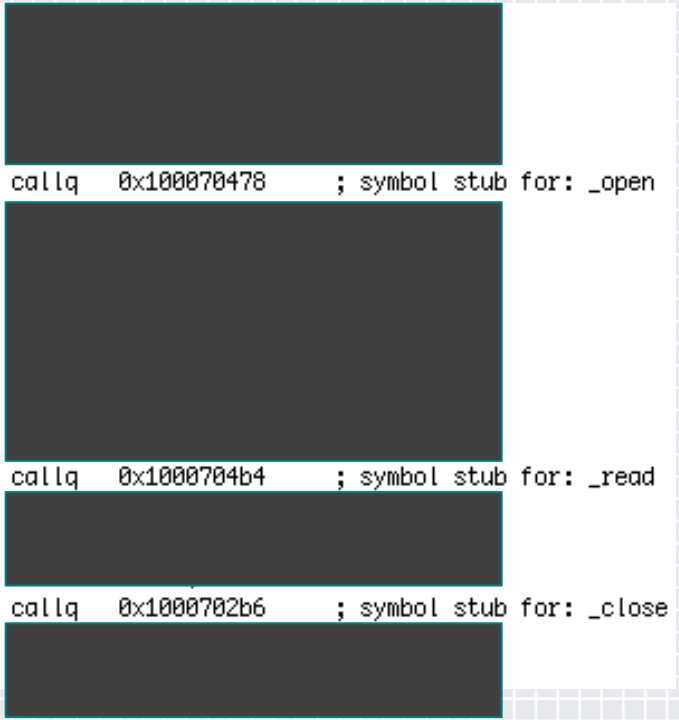
# Sandbox Designs




Full system emulation

- monitor interactions with OS and all instructions
- full kernel visibility
- implementation is more difficult

# VM Approach 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    0xffffff70(%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
```

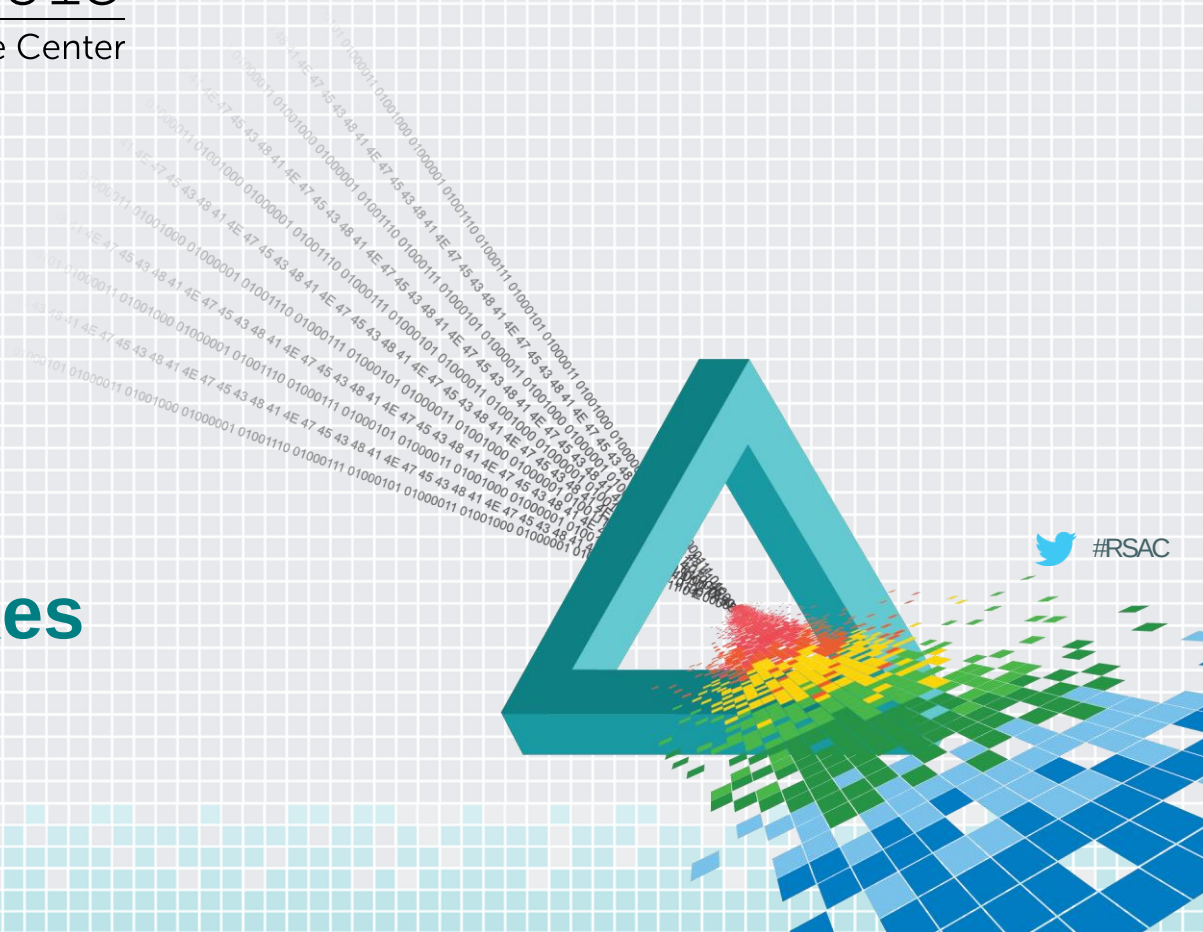
# Visibility Does Matter

- 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
  - see everything and adapt to attacker's threats
  - detect triggers
  - bypass stalling code

RSAConference2015

San Francisco | April 20-24 | Moscone Center

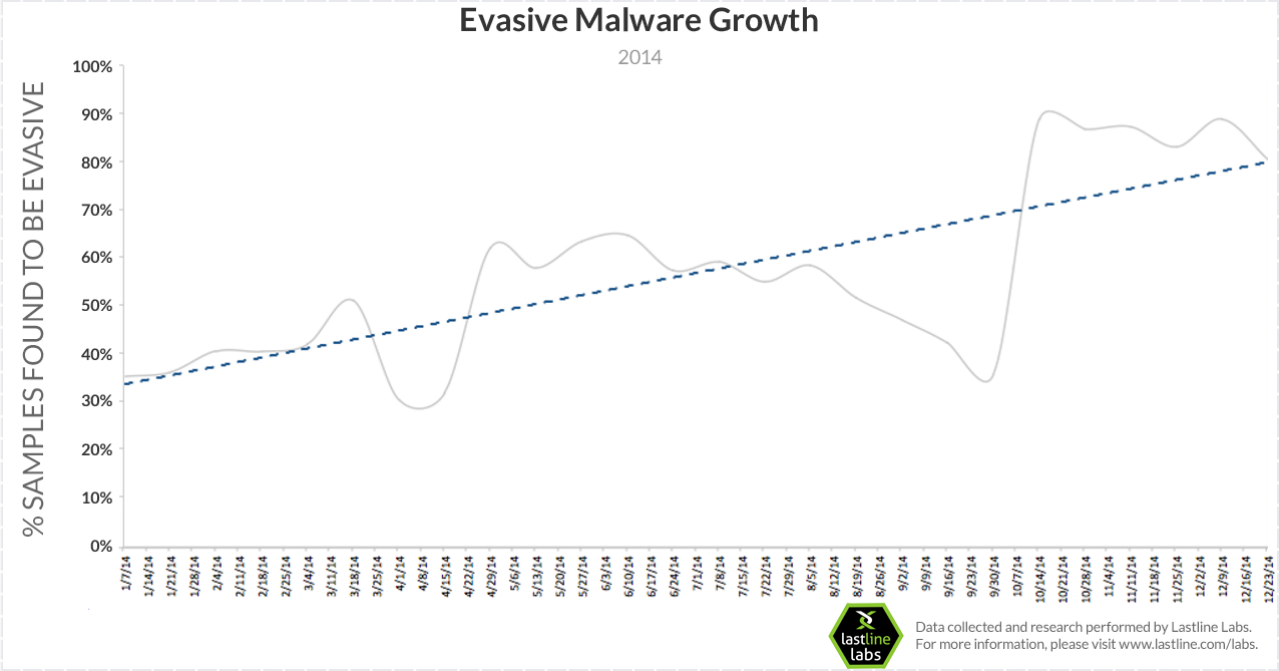
# Evading Sandboxes



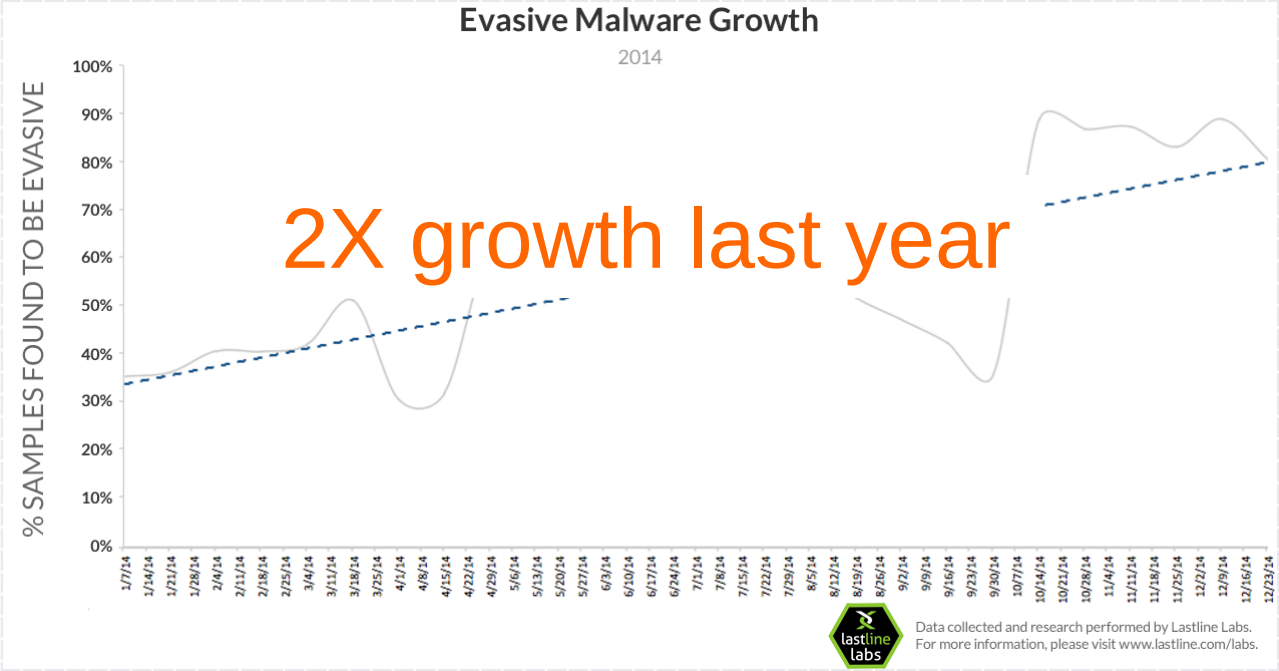
# Evading Dynamic Analysis

- Malware authors don’t sleep
  - 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

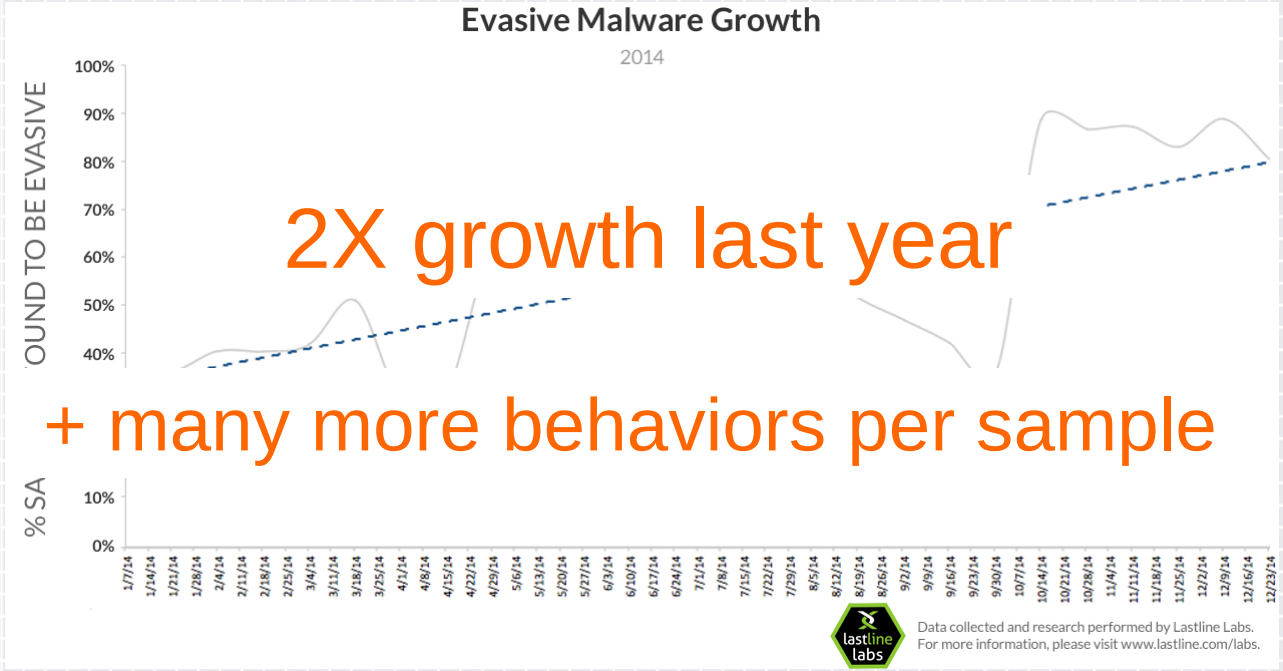
# Evasion Going Mainstream



# Evasion Going Mainstream



# Evasion Going Mainstream



# Evading Dynamic Analysis

- Malware can detect runtime or analysis environment
  - differences between virtualized and bare metal environment
  - checks based on system (CPU) features
  - checks based on operating system artifacts (files, processes, …)
- Malware can exploit limited context
- 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
  - move code into kernel space (rootkits)

Environmental Awareness

Timing-based Evasion

# Detect Analysis Environment

- Check Windows 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, …
- WMI queries

# Detect Analysis Environment

# Detect Analysis Environment

Enigma Group's Hacking Forum

```
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 == "Schmidti") || (user == "schmidti") ){
  return true; // Detected CW Sandbox.
} else if( (snd = FindWindow("Afx:400000:0", NULL)) ){
  return true; // Detected WinJail Sandbox.
} else {
  return false;
}
```

HWND snd;

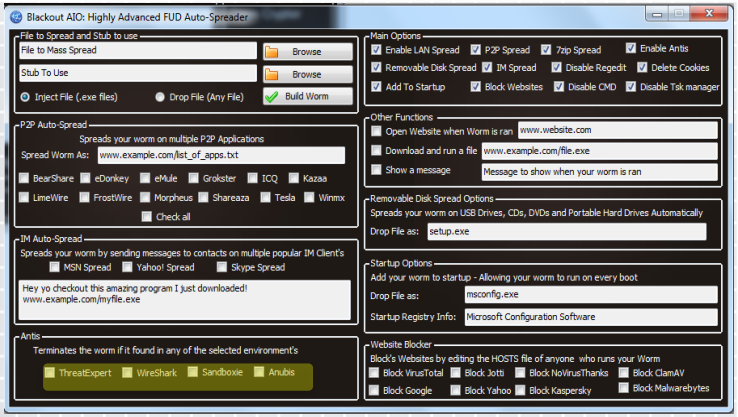
if( (snd = FindWindow("SandboxieControlWndClass", NULL)) ){
  return true; // Detected Sandboxie.

lastline

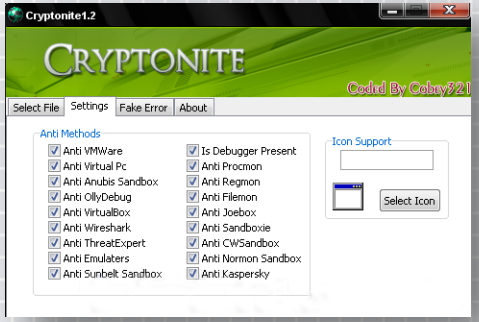
# Detect Analysis Environment

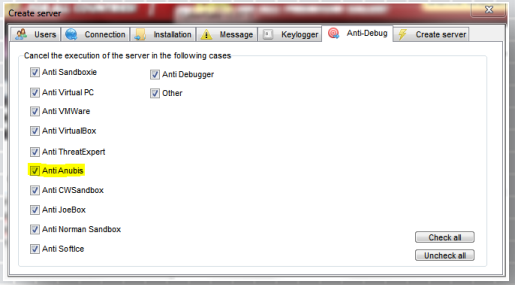
- Current usage of both physical and virtual memory
  - `GlobalMemoryStatus`
- CPU properties
  - `NtOpenKey (Hardware\Description\System\CentralProcessor\0)`
- Check for hard drive properties
  - `DeviceIoControl (IOCTL_STORAGE_QUERY_PROPERTY)`
  - `DeviceIoControl (IIOCTL_DISK_GET_LENGTH_INFO`
- Device name
  - `SetupDiGetDeviceRegistryProperty (SPDRP_FRIENDLYNAME)`
- Check for number of processors
  - `GetSystemInfo`

# Detect Analysis Environment






# Exploit Limited Context

- In certain cases, malware is targeted for specific organization
  - malware doesn't need to detect analysis environment
  - instead, only run on very specific, intended target

- This idea has become more popular in APT attacks
  - attacker can leverage much of previously discussed techniques
  - additional information could come from local network environment

# Avoid Monitoring

- Open window and wait for user to click
  - or, as 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
- Bypass in-process hooks (e.g., of library functions)

# 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   <- second instruction of AdjustTokenPrivlages

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
```

jump to second instruction of library function

# Avoid Monitoring

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

# The Simple Sleep Attack

```
push  20000000h
call  Sleep
```

Sleep(x) - sleeps x milliseconds

# Sandbox Controls Time APIs

- `Sleep (NtDelayExecution)`
- `SetTimer (NtSetTimer)`
- `NtWaitforSingleObject (NtWaitFor*)`
- `WaitForMultipleObjects (NtWaitFor*)`

# Avoid Monitoring

Anti-sleep-acceleration

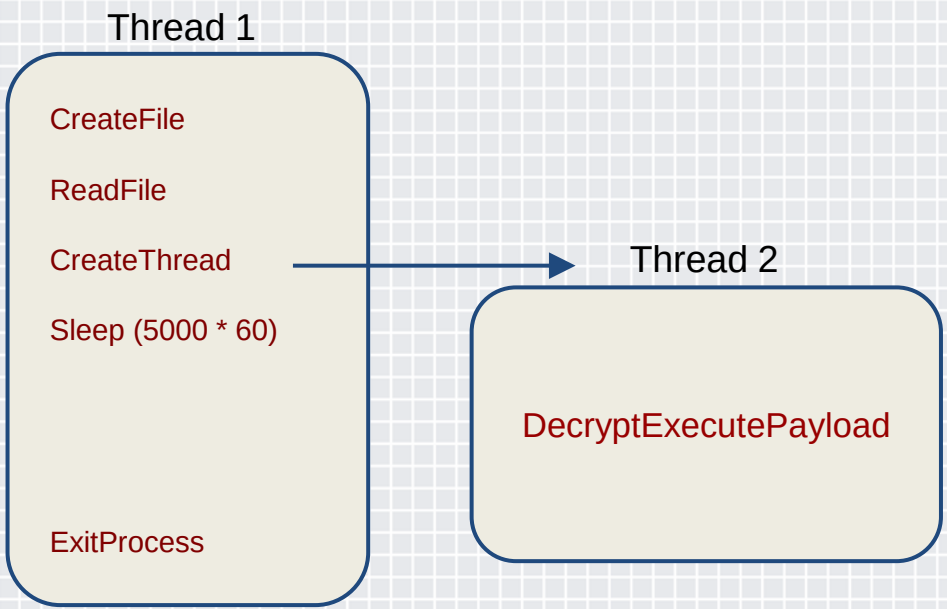
- introduce a race condition that involves sleeping

- Sample creates two threads
  1. `sleep()` + `NtTerminateProcess()`
  2. decrypts and runs payload

- Another variation
  1. `sleep()` + `DeleteFileW(`<name>.bat`)`
  2. start <name>.bat file

# Timing Attack: Race Condition

```
push edi ;hTemplateFile
push edi ;dwFlagsAndAttributes
push 3   ;dwCreattionDesposition
push edi ;lpSecurityAttributes
push 1   ;dwShareMode
push 80000000h ;dwDesiredAccess
push ebx ;lpFileName
call CreateFileA
cmp eax, 0FFFFFFFFh
jz fail
lea eax, [ebp + NumberOfBytesRead]
push edi  lpOverlapped
push ecx ; plNuberOfBytesRead
push esi ; nNumberOfBytesToRead
push [ebp + DecryptExecutePayload]
push eax
call ReadFile
cmp [ebp + NumberOfBytesRead], edi
jbe fail
; Decrypts and execute an encypted code,
: which creates a new thread for a payload function
call [ebp + DecryptExecutePayload]
push 0x493e0
call Sleep
fail:
push edi
call ExitProcess
```



# Avoid Monitoring

Anti-sleep-acceleration

- explicitly check for time that has passed
- sometimes using and comparing multiple time sources

# Timing Attack: Sleep and TSC

```
    rdtsc
    mov             [ebp+RDTSC1_EAX], eax
    mov  [ebp+RDTSC1_EDX], edx
    push 20000h
    call   Sleep
    rdtsc
    sub   edx, [ebp+RDTSC1_EDX]
    cmp  edx, 0
    jg short return_success
    sub   eax, [ebp+RDTSC1_EAX]
    cmp  eax, 20000h
    jge short return success
    mov  eax, 1
    retn
return success:
    mov  eax, 0
    retn
```

```
int detect_time_manipulation()
{
        rdtsc_value1 = get_rdtsc_value();
        Sleep (0x20000);
        rdtsc_value2 = get_rdtsc_value();

        if (rdtsc_value2 - rdtsc_value1 >= 0x20000)
           return 0;
        return 1;
}
```

# Timing Attack: Sleep, TSC and Ticks

```
rdtsc
mov    [esp+RDTSC1_EAX], eax
mov    [esp+RDTSC1_EDX], edx
call   GetTickCount
mov    ebx, eax ; EBX contains Tick Counter 1
push   10000
call   Sleep
rdtsc
; Calculate RDTSC difference
sub    eax, [esp+RDTSC1_EAX]
sbb    edx, [esp+RDTSC1_EDX]
mov    [esp+RDTSC_DIFF_EAX], eax
mov    [esp+RDTSC_DIFF_EDX], edx
call   GetTickCount
; Calculate GetTickCount difference
mov    ecx, eax
sub    ecx, ebx
cmp    [esp+RDTSC_DIFF_EDX], 0
jnz    short fail
cmp    [esp+RDTSC_DIFF_EAX], 50000000
jb     short return1_fail
jmp    short return_0_sucess
jb   short return1_fail
jmp    short return_0_sucess
fail:
jl   short return1_fail
```

```
int detect_time_manipulation()
{
        rdtsc_value1 = get_rdtsc_value();
        tick_cout1 = GetTickCount();
        Sleep(10000);
        rdtsc_value2 = get_rdtsc_value();
        tick_cout2 = GetTickCount();

        if (rdtsc_value2 - rdtsc_value1 < 50000000)
            return 1;
        if (tick_cout2 - tick_cout1 < 50)
            return 1;
        return 0;
}
```

lastline

# Timing Attack: Stalling Loops

```
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 }
```

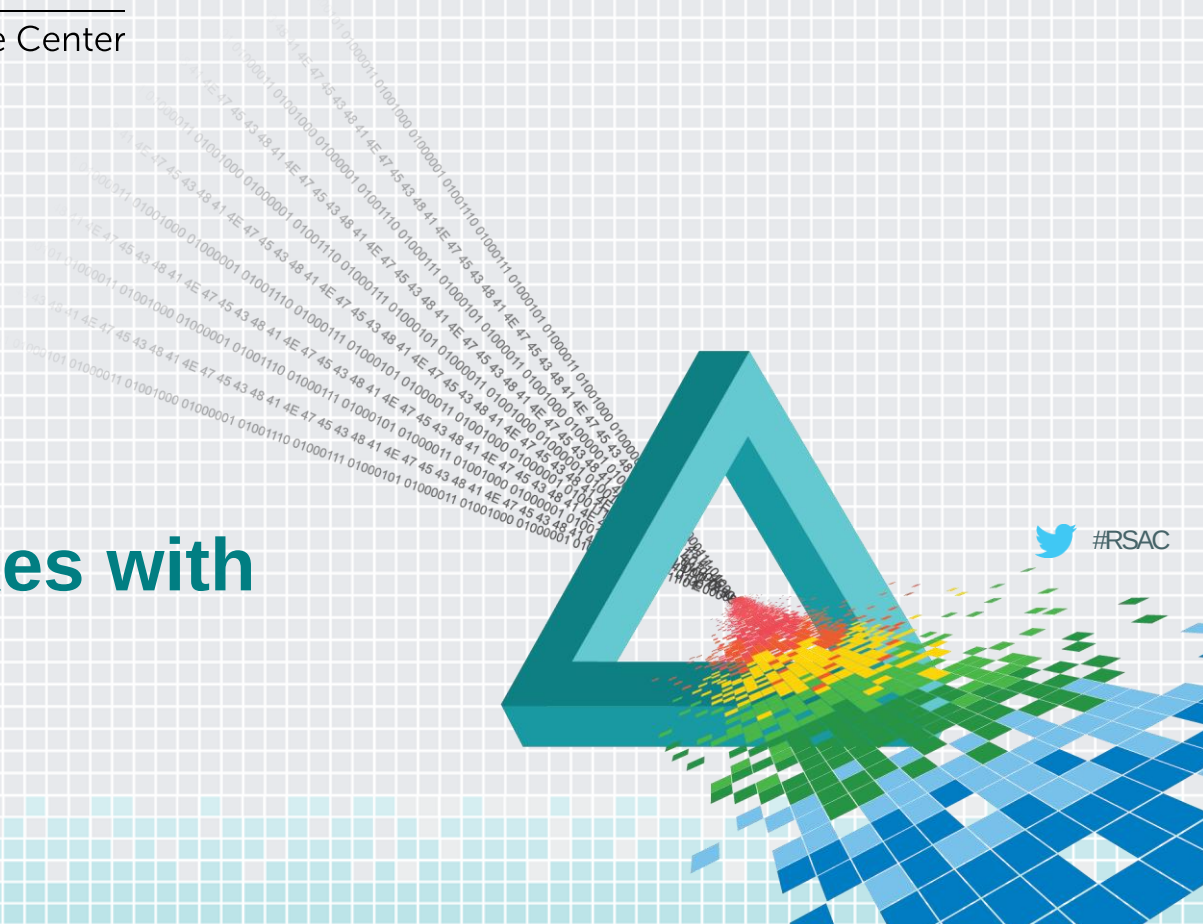
Figure 1. Stalling code found in real-world malware (W32.DelfInj)

# Example: Carbanak

- Used to infiltrate banks and takeover ATMs
- $1B raked in

- Stealth Behaviors
  - Hide .exe files
  - Unpacking behavior
  - Code injection to hide network activity
- Evasion Behaviors
  - Altered memory image of process
  - Virtual sandbox detection
  - Sleep calls
  - Forbid Debugging

# Evading Sandboxes with Kernel Malware



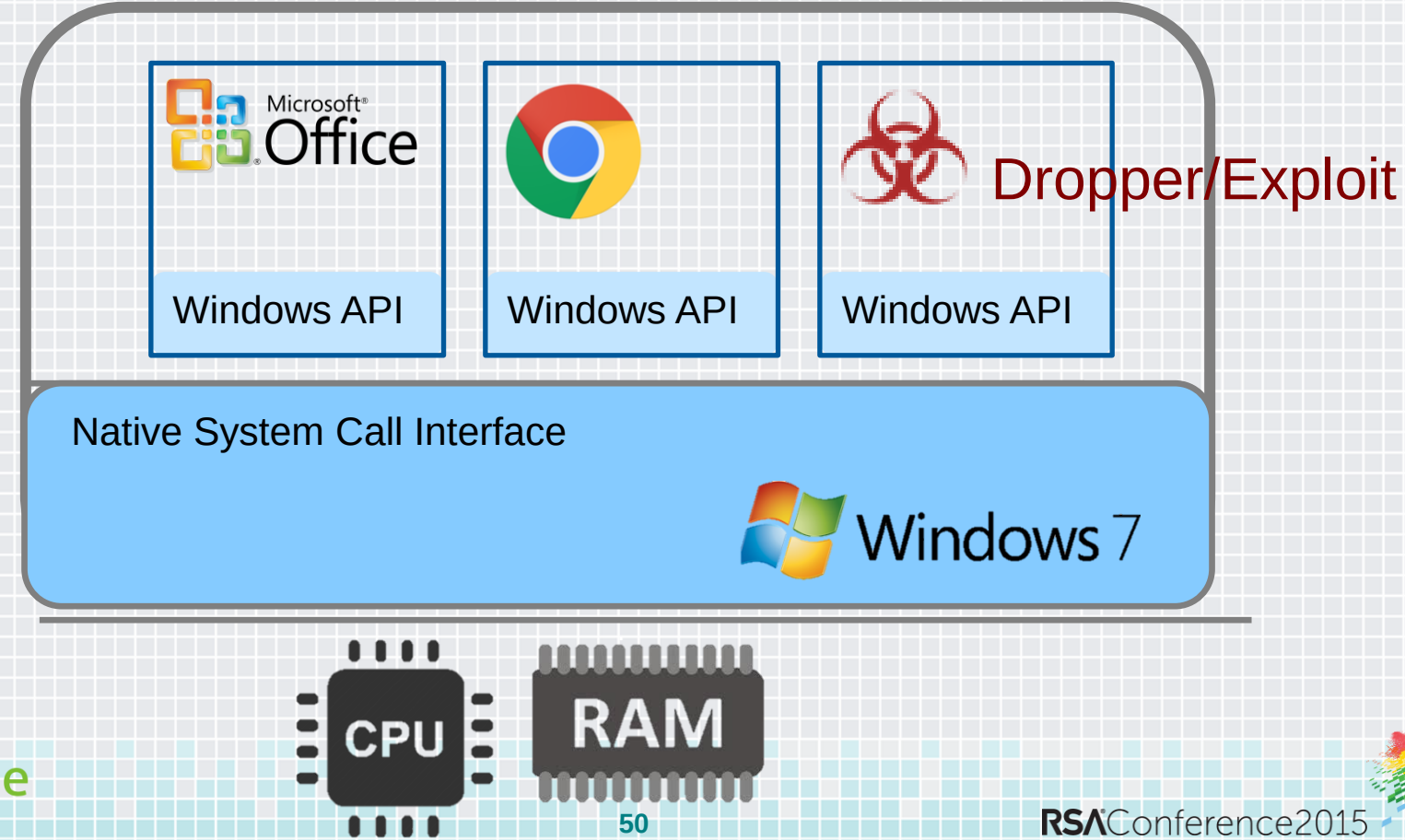
# Kernel Malware

- Problematic for many sandboxes
  - operates underneath the monitored interface
  - behaviors do not show up as system calls
- Critical component used in sophisticated APT attacks
  - Equation, Regin, Dark Hotel, Turla/Uroburos

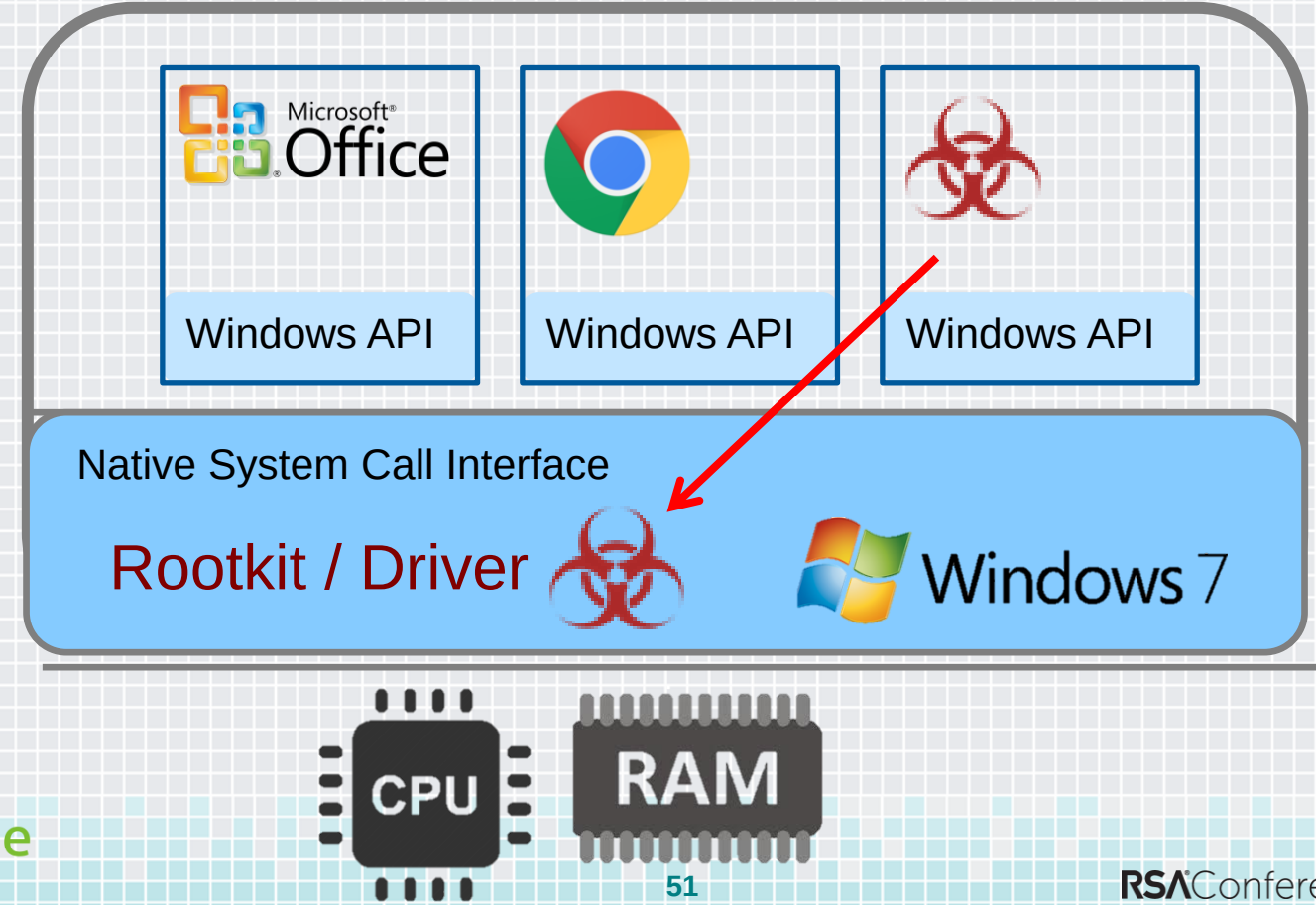
# Kernel Malware

- Three many steps
  1. inject malicious code into kernel
  2. make kernel execute malicious code
  3. implement malicious functionality

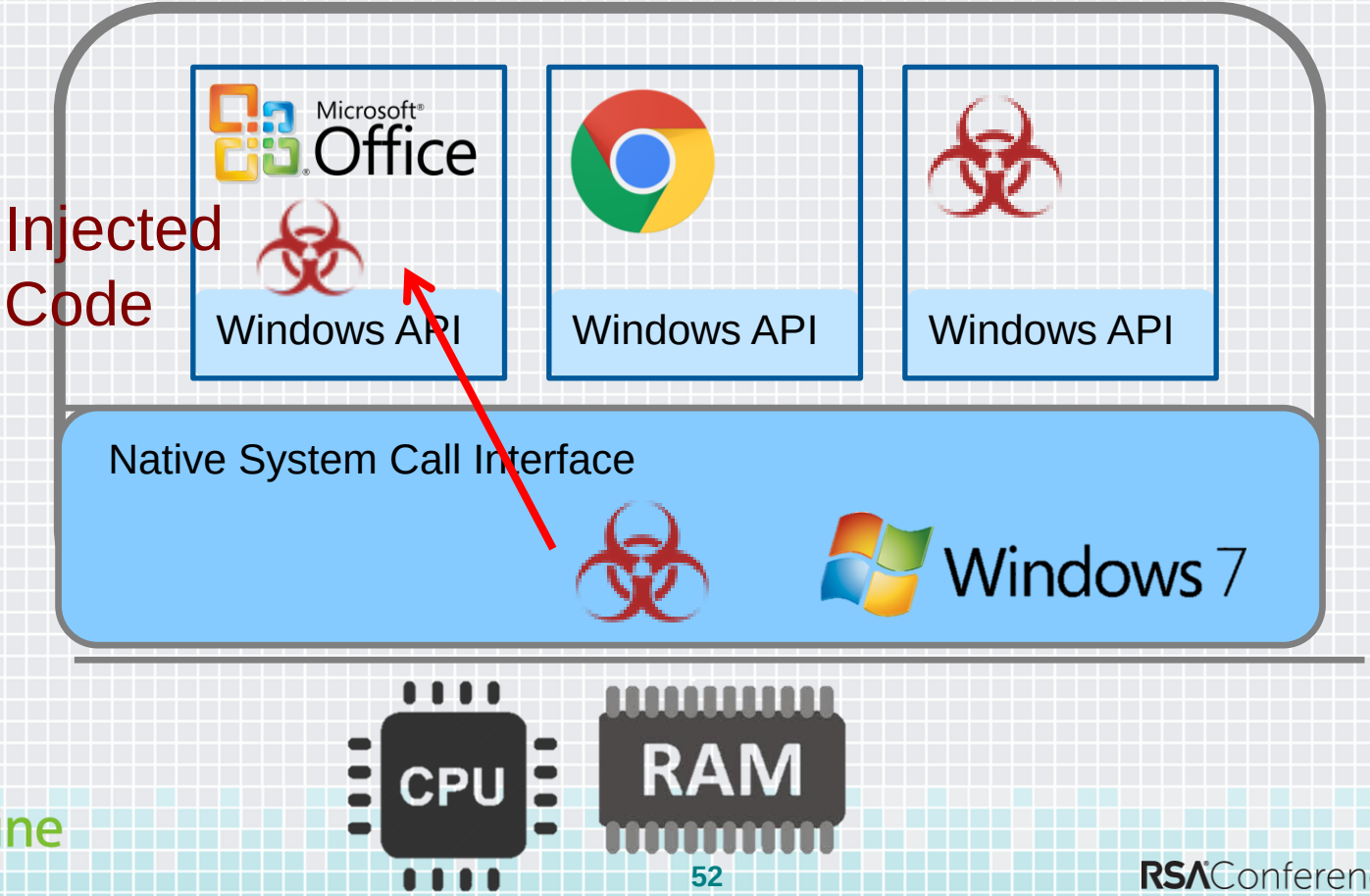
# Kernel Malware



# Kernel Malware



# Kernel Malware



# Kernel Malware

- Inject code into kernel
  - load a driver into the kernel
  - problem: newer versions of Windows only load signed drivers
  - solution: steal certificate and sign your own driver
  - solution: reboot OS into mode where driver checks are disabled
  - solution: load vulnerable driver and exploit it

# Kernel Malware

- Make kernel execute new code
  - redirect (change) code pointer to point to malicious code
    - system call and interrupt tables are classic targets
  - problem: Windows PatchGuard monitors integrity of system-critical data structures such as *SSDT*, *IDT*
  - solution: tamper with PatchGuard and disable its functionality
  - solution: redirect code pointers that PatchGuard doesn't monitor

# Kernel Malware

- Implement malicious functionality
  - you are in the kernel, you can do anything you want
  - problem: kernel programming is not trivial, and mistakes crash the system
  - solution: inject malicious code into legitimate apps or libraries
    - this can be done by changing directly their memory
    - alternatively, one can simply change code in libraries or on disk

# Example: Turla

- Load and exploit vulnerable VirtualBox driver
- Disable check for signed driver loading (*g_CiEnabled*)
- Load whatever you want

```
fffff880`03327b4e 498bd2          mov     rdx,r10
fffff880`03327b51 e8aa060000      call    usbehub!AssertMsg1+0x9a0 (fffff880`03328200)
fffff880`03327b56 0fb6cb          movzx   ecx,bl
fffff880`03327b59 440f22c1        mov     cr8,rcx
fffff880`03327b5d 33db            xor     ebx,ebx
fffff880`03327b5f 895f30          mov     dword ptr [rdi+30h],ebx
fffff880`03327b62 48c7473804000000 mov     qword ptr [rdi+38h],4
fffff880`03327b6a 488b4f70        mov     rcx,qword ptr [rdi+70h] ds:002b:fffffa80`02a03250={nt!g_CiEnabled (fffff800`02c72eb8)}
fffff880`03327b6e 8901            mov     dword ptr [rcx],eax
fffff880`03327b70 eb0a            jmp     usbehub!AssertMsg1+0x31c (fffff880`03327b7c)
```

# Example: Turla

- Tamper with data structures that PatchGuard monitors
- Then, deal with the consequences (blue screen of death)

- PatchGuard invokes *KeBugCheckEx*
  - hook *KeBugCheckEx* function and simply return
- Updated PatchGuard includes its own copy of *KeBugCheckEx*
  - hook RtlCaptureContext and simply return

# Example: Turla

- Traditional rootkit behavior
  - redirect interesting system calls into single interrupt handler
  - dispatch and make desired changes to system call functionality

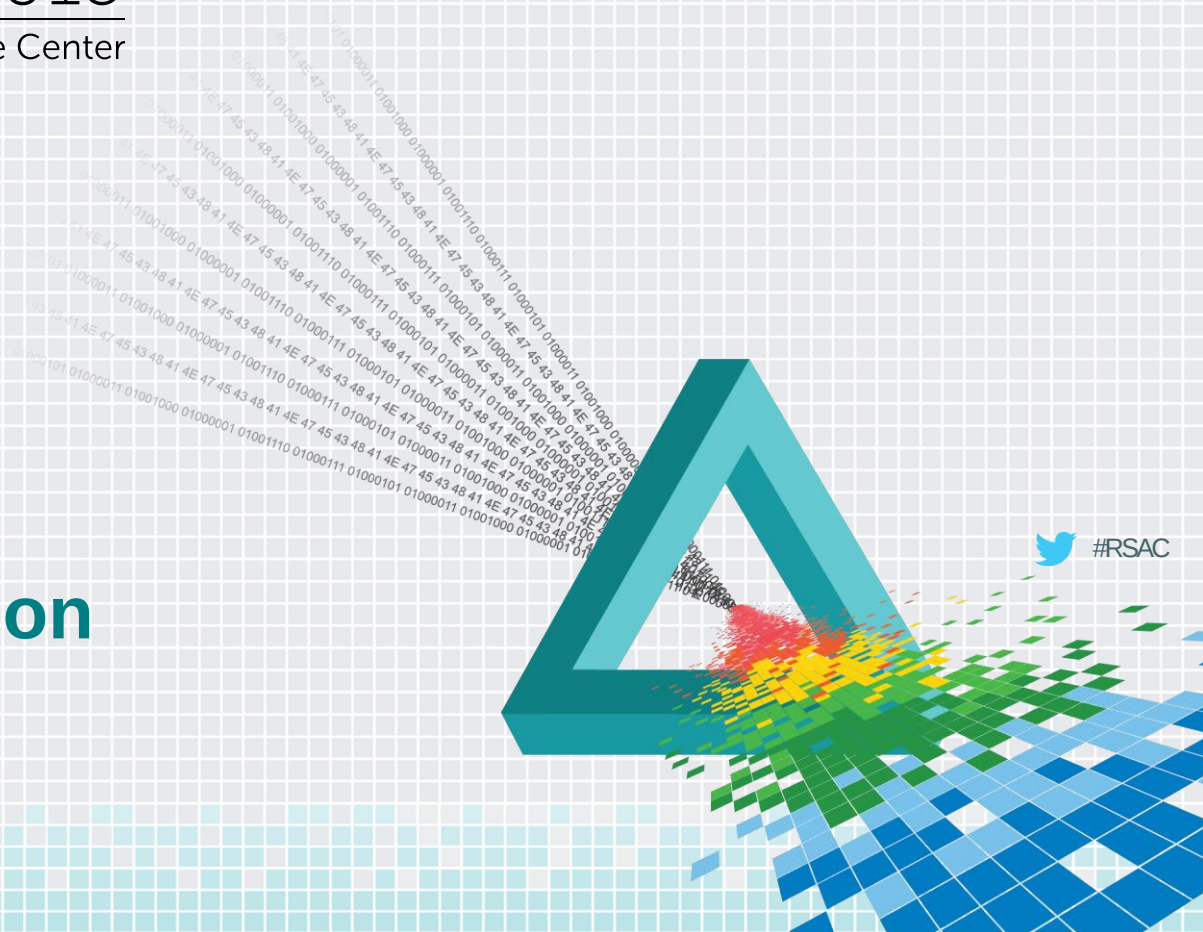
# Example: Turla

**Maliciousness score** **95/100**
**Risk estimate** High Risk - Malicious behavior detected

## Analysis Overview

| Type | Description |
|---|---|
| File | Modifying executable in Windows directory |
| Memory | Creating new entry in interrupt descriptor table (IDT) |
| Memory | Modifying interrupt descriptor table (IDT) |
| Memory | Modifying image in kernel address space |
| Rootkit | Disabling driver signature verification |
| Rootkit | Disabling kernel patch protection (PatchGuard) |
| Rootkit | Hiding running processes |
| Rootkit | Intercepting/monitoring filesystem activity |
| Rootkit | Intercepting/monitoring network activity |
| Rootkit | Intercepting/monitoring process creation |
| Rootkit | Intercepting/monitoring system registry activity |

# Addressing Evasion



# What can we do about evasion?

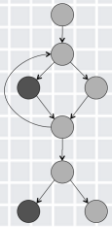
- Visibility is key
  - when the sandbox can see more things, it can react to more threats

# Visibility Matters

| Type | Family | Driver | lastline | Traditional Sandbox |
|---|---|---|---|---|
| Traditional Rootkit | XCP | 32-bit | Detected | Failed |
| Traditional Rootkit | Zhelatin | 32-bit | Detected | Failed |
| Traditional Rootkit | Srizbi | 32-bit | Detected | Failed |
| Traditional Rootkit | Blakken | 32-bit | Detected | Failed |
| Traditional Rootkit | Agent | 32-bit | Detected | Failed |
| Traditional Rootkit | TDSS | 32-bit | Detected | Failed |
| APT | Dark Hotel | 32-bit | Detected | Failed |
| APT | Mask | 32-bit | Detected | Failed |
| APT | Turla | 32-bit | Detected | Failed |
| APT | Turla | 64-bit | Detected | Failed |

# 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 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
  - restoring works by overwriting current address space with stored image
- Explore alternative branch
  - restore process memory image
  - set the tainted operand (register or memory) to a value that reverts branch condition
  - let the process continue to run

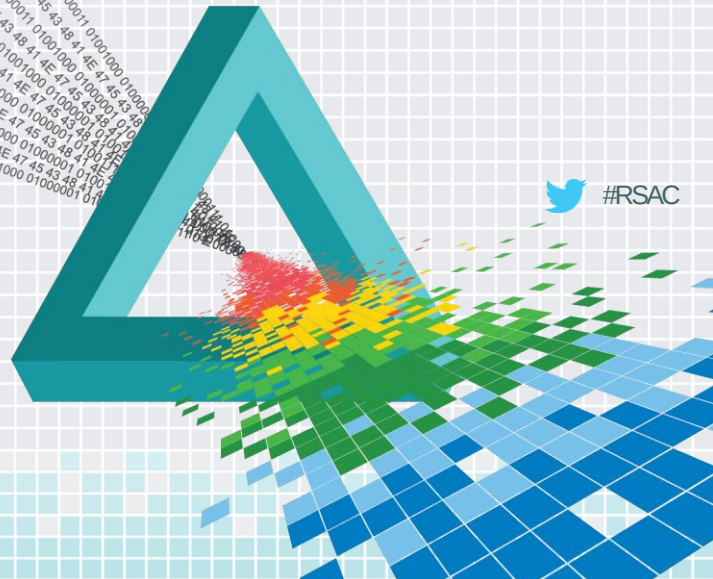
# What can we do about evasion?

- Sometimes, it is difficult to get to interesting behaviors
  - however, evasion is a strong signal for malicious intent
  - when you can see evasion, you can use this against malware

RSAConference2015

San Francisco | April 20-24 | Moscone Center

# Wrapping Up



# Apply

- Dynamic analysis is a powerful tool
  - consider integrating sandbox capabilities into your defenses
- Dynamic analysis capabilities vary significantly
  - understand limitations and evasive threat
  - ask your vendor questions about their sandbox, dig deeper
    - what file types can the sandbox analyze? what activities can it see?
    - how does it handle evasion? how does it deal with malicious kernel code?
- Think about what you want to get out of a sandbox
  - detection (black/white) and/or support for forensics (detailed behaviors)?

# 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
- Many evasion tricks are possible
  - detecting environment
  - timing-based attacks
  - avoid analysis system by moving into the kernel

# THANK YOU!

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