

**REcon 2017**

**Catching a Tricky Fox:  
Tactics and Strategies of Sandbox Evasion in Modern Malware**

Maksim Shudrak  
PhD, Security Researcher  
IBM Research Israel

**Montreal, 16-18 June 2017**

## **Introduction**

Nowadays, a malware analysis sandbox is a standard component of every research lab. A tempting idea to understand a goal and actual malicious intent of malware authors without reverse engineering attracts with its simplicity many researchers. The reverse cite of this simplicity is a lack of transparency of such sandbox towards analyzed software. There are numerous ways to detect a sandbox based on artifacts introduced by virtualization, emulation or some other components as shown [2] [4] [5]. Of course, it would be foolish not to assume that this will be not used by malware authors. Our observation of incoming threats demonstrates that evasive malware is becoming standard-de-facto even in a simple malware not to mention advanced threats. This trend is confirmed by other researchers [1] [3] as well.

The main goal of my talk would be to introduce for an auditory a general problem of a sandbox transparency, a set of various techniques that are used by malware to avoid automatic analysis and detection based on our detailed reverse engineering experience of a numerous recent threats including several unpublished ones as well as to discuss possible solutions to solve this problem.

The positive impact for the audience would be a useful and practical information how to make the process of malware analysis and detection stronger against detection and evasion.

## **2. Problem statement**

Most of the modern sandboxes offer dynamic analysis of potentially malicious executables. There are three possible solutions to achieve that:

- 1) Emulation
- 2) Virtualization
- 3) Bare-metal execution

Before we start describing evasion mechanisms in malware, it would be great to understand what stay behind these three approaches.

### **2.1. Emulation**

The general idea of an emulation-based sandbox is to correctly reproduce original function of a target CPU and emulated environment to be able to correctly execute a malicious executable on top of that. Thus, emulation gives us an ability to “run” code written for one architecture on another (e.g. x86->arm, arm->x64 etc). Moreover, besides the processor, emulator needs to provide access to RAM and low-level hardware features.

In terms of malware analysis, we can run the whole operating system with target application inside and have the full control and visibility, transparently handling each instruction executed by guest operating system. This is called “full-system” emulation and is supported by several solutions such as Lastline [6], Panda [7]. While such approach allows us to break semantic gap, we have a serious challenge related to performance (for example the overhead of QEMU running in emulation mode may easily exceed x20) as well as each artifact or inconsistency in the emulation of instructions or hardware may be used to detect our emulator.

There is another approach that helps us to deal with performance issues called “user-mode” emulation. In this case, instead of emulate RAM and low-level hardware features, we perform “reduced” emulation of OS’s API or system calls along with CPU instructions. In this case, we still have the same visibility under analyzed executable, however, we don’t need to emulate the rest of the OS. While it seems reasonable to use such approach to fix performance issues, we have a lot of other problems. Correct emulation of API calls is an extremely hard problem due to large amount of various functions provided by OS including undocumented ones. Of course, there is a possibility to emulate only system calls, however they are much less documented and may vary from version to version.

Despite these, a lot of AVs use such approach to perform automatic unpacking of suspicious executables by emulating only important (in terms of malware execution) subset of Windows API. This fact is often used by malware writers to evade emulation. Moreover, vulnerability in emulator may be even used to get higher privileges in the system [8].

## **2.2. Virtualization**

When it comes about malware analysis sandbox it usually means a VM with additional components for malware behavior analysis. In VM, we can run guest only on the host processor with the same architecture. Modern processors provide hardware-assistant virtualization where we can run VM almost that fast as a real machine in a restricted environment. However, instructions that are executed in a VM are not visible for the hypervisor because VM occupies the actual physical CPU and cannot be executed in parallel. To break semantic gap, we need to add additional components to trace malware in the hypervisor (to trace system calls) or OS (API hooking). We also may externally analyze the VM’s memory (for example by using Volatility framework).

Unfortunately, these components as well as virtual machine itself introduce numerous artifacts that are possible to detect without significant efforts. While theoretically there is a possibility to create absolutely “transparent” virtual machine, most researchers agree that in practice it is not a realistic task and requires huge amount of work [9]. Moreover, such virtual machine will significantly lose in performance.

## **2.3. Bare metal execution**

If our malware can detect a VM, we can dedicate a physical machine to perform dynamic analysis. While we still need to deal with semantic gap which requires some additional instruments to be installed in the system, we solve a problem of VM detection. Moreover, we can use special hardware-assistant approaches such as SMM-mode of processor [10] or even install a special hardware device [11] to log each executed instruction. It solves most of the problems described above but bare-metal execution is less scalable and requires significantly more effort to provide isolated environment for malware. We need to ensure that our system has been recovered to the state before malware execution and we need to do it efficiently to be able to perform fast analysis of incoming threats.

It is sufficient to say that all the sandboxes described above is also possible to detect by checking for configuration details related to specific sandbox (e.g. user or computer name, specific MAC-address etc).

### 3. Techniques

A lot of malicious samples use anti-research techniques to bypass automatic detection in sandboxes. However, we see a significant difference in strategies to evade dynamic analysis across various families of malware.

#### 3.1. Anti-research/anti-VM

Gootkit's authors [12] introduce dozens of anti-research tricks in their dropper to avoid running in a VM by checking for:

- *VideoBiosVersion* at HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System\ looking for substring "VirtualBox".
- *SystemBiosVersion* at HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System looking for substrings *AMI*, *BOCHS*, *VBOX*, *QEMU*, *SMCI*, *INTEL-6040000* and *FTNT-1*.
- *DigitalProductId* at HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT\CurrentVersion looking for 55274-640-2673064-23950 (*JoeBox*) and 76487-644-3177037-23510 (*CWSandbox*) product ID keys.
- Current Windows user name comparing with "CurrentUser" and "Sandbox".
- Workstation name comparing with "SANDBOX" and "7SILVIA" (Figure 1).
- Sandbox-related dlls such as *sbiedll.dll* (*Sandboxie*), *pstorec.dll* (*SunBelt Sandbox*) or *dbghelp.dll* by using *GetModuleHandle* API call.

000AA99A	50	PUSH EAX	
000AA99B	56	PUSH ESI	
000AA99C	FF15 CC00B00	CALL DWORD PTR [B00CC]	kernel32.GetComputerNameA
000AAA6A	3D 18E83A00	CMP EAX,3AE818	
000AAA6F	74 12	JE SHORT 000AAA83	
000AAA71	8BC7	MOV EAX,EDI	
000AAA73	8D0C37	LEA ECX,DWORD PTR [EDI+ESI]	
000AAA76	99	CDQ	
000AAA77	F77D B0	IDIV DWORD PTR [EBP-50]	
000AAA7A	8A4415 F8	MOV AL,BYTE PTR [EBP+EDX-8]	
000AAA7E	320419	XOR AL,BYTE PTR [ECX+EBX]	
000AAA81	8B01	MOV BYTE PTR [ECX],AL	
000AAA83	47	INC EDI	
000AAA84	83FF 08	CMP EDI,8	
000AAA87	7C DB	JL SHORT 000AAA64	
000AAA89	56	PUSH ESI	ASCII "7SILVIA"
000AAA8A	68 F0CD0B00	PUSH 0BCDF0	ASCII "WIN7_X86_SP1"
000AAA8F	FF15 E000B00	CALL DWORD PTR [B00E0]	kernel32.lstrcmpA
000AAA95	8B1D 34010B00	MOV EBX,DWORD PTR [B0134]	kernel32.GetProcessHeap
000AAA9B	85C0	TEST EAX,EAX	
000AAA9D	75 05	JNZ SHORT 000AAAA4	
000AAA9F	E8 84FCFFFF	CALL 000AA728	self-desctruction routine
000AAAA4	6A 1D	PUSH 1D	

Figure 1. Gootkit<sup>1</sup> checks workstation name comparing with "SANDBOX" and "7SILVIA"

While names "Sandbox" or "CurrentUser" looks like a general check, the string "7SILVIA" give us food for thought. We may guess that Gootkit's authors owns information about specific configuration details of a specific sandbox that they try to bypass.

To avoid execution on servers, the sample looks for "Xeon" substring at *ProcessorNameString*<sup>2</sup> (Figure 3). Sandbox engineers usually try to save hardware resources and allocate only one core per VM. So, Gootkit also checks for the number of available processor cores and if it equals 1, the sample initiates self-destruction ensues.

<sup>1</sup> 60e079ec28d47ef85e93039c21afd19c

<sup>2</sup> Registry key HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0

000A7407	50	PUSH EAX	pHandle
000A7408	56	PUSH ESI	Hardware\DESCRIPTION\System\CentralProcessor\0
000A7409	68 02000080	PUSH 80000002	HKEY_LOCAL_MACHINE
000A740E	FF15 60000B00	CALL DWORD PTR [B0060]	advapi32.RegOpenKeyW
000A74DC	50	PUSH EAX	pBufferSize
000A74DD	8D8424 34010000	LEA EAX,DWORD PTR [ESP+134]	
000A74E4	50	PUSH EAX	pBuffer
000A74E5	33C0	XOR EAX,EAX	
000A74E7	50	PUSH EAX	NULL
000A74E8	50	PUSH EAX	NULL
000A74E9	57	PUSH EDI	ProcessorNameString
000A74EA	FFB424 D0000000	PUSH DWORD PTR [ESP+D0]	pHandle
000A74F1	FF15 70000B00	CALL DWORD PTR [B0070]	advapi32.RegQueryValueExW
000A759A	56	PUSH ESI	Xeon
000A759B	50	PUSH EAX	Intel(R) CPU-E5-2650 v3 @ 2.30Ghz
000A759C	FFD3	CALL EBX	shlwapi.StrStrIW

Figure 2. Gootkit obtains a ProcessNameString looking for substring “Xeon”

Moreover, Gootkit checks against well-known “forbidden” processes in the OS, such as “procmon.exe” or “wireshark.exe” specific for sandbox environment. GootKit’s developers use a special self-written hash function to check against 46 hardcoded hashes in memory. Thus, we don’t see any strings nor any API calls that perform string comparisons, even though if one out of 46 forbidden processes are running in the OS, the sample will stop execution (Figure 3). The hash function itself is shown in Figure 4.

```

unsigned int hash_func(const char *str, int len) {
    unsigned int hash = 0xffffffff;
    if (len <= 0)
        return -1;
    unsigned int b = 0;
    unsigned int tmp = 0;
    for (int i = 0; i < len; i++) {
        b = str[i];
        for (int j = 0; j < 8; j++) {
            tmp = b ^ hash;
            hash = hash >> 1;
            if ((tmp & 1) == 1)
                hash ^= 0xedb88320;
            b = b >> 1;
        }
    }
    return hash;
}

```

Figure 3: C-code of the hash function used to hide plain-text strings in the memory (reversed)

```

unsigned int hash_list[] = {0x278cdf58, 0x62b621c4, 0x7ec953ab, 0xb4c2ed27, 0xe1e54873,
                           0x7b8b2670, 0x7deed7db, 0x2d386ece, 0x581419ac, 0xa93a5da5,
                           0x4055c0a5, 0xbf550eed, 0x86bd8b3a, 0x68b0f30d, 0xd35c5e5c,
                           0x6fadb57b, 0x47e5605f, 0x70f400cf, 0x9f5462ed, 0x0afb3480,
                           0x5d5421cf, 0x1e84d9c6, 0x6751a7a7, 0x896773d7, 0x0f0fc4f7,
                           0xf9b64044, 0x11e91917, 0x1cb3f267, 0x6de558e4, 0x487c3558,
                           0x72c7bd89, 0x01b54824, 0x6e4851f8, 0xbc541011, 0x4a6b6ebc,
                           0x1e24d477, 0x52feb192, 0x7e11e4cf, 0xbffde1f0, 0xaba416e3,
                           0x0e2f42d3, 0x9fe09b81, 0xe1e54873, 0x334b7fa5, 0xd8367b99,
                           0xb15afa72};

bool check_vm_processes() {
    unsigned int hash;
    bool result;
    char process_name[MAX_PROC_NAME_LEN];
    int hash_count = sizeof(hash_list)/sizeof(unsigned int);

    for (int i = 0; i < hash_count; i++) {
        /* get first process name */
        result = get_process_name(process_name/*out*/);
        while (result == true) {
            /* convert string to uppercase and calculate hash */
            hash = hash_func(to_upper_case(process_name), strlen(process_name));
            if (hash == hash_list[i])
                return true; /* if hash matches, sandbox has been detected */
            /* get next process name */
            result = get_process_name(process_name/*out*/);
        }
    }
    return false; /* sandbox has not been detected */
}

```

Figure 4. A function that checks for “forbidden” processes running in the OS (reversed)

We checked this function against well-known names of reverse-engineering tools and sandboxes, as well as implemented a brute-force attack and successfully found the following 41 names:

procexp.exe, ollydbg.exe, vmwareuser.exe, vboxservice.exe, vboxtray.exe, apispy.exe, hips32.exe, python.exe, regmon.exe, tshark.exe, vmsrvc.exe, perl.exe, idag.exe, idaq.exe, imul.exe, peid.exe, hookanaapp.exe, procmon.exe, fortitracer.exe, emul.exe, autoruns.exe, autorunsc.exe, dumpcap.exe, hookexplorer.exe, importrec.exe, joeboxcontrol.exe, joeboxserver.exe, multi\_pot.exe, petools.exe, proc\_analyzer.exe, scktool.exe, sniff\_hit.exe, sysanalyzer.exe, vmusrvc.exe, vmwaretray.exe, wireshark.exe, xenservice.exe, autoit.exe, pythonw.exe, php.exe, pos\_trigger.exe.

While most of the processes look reasonable we have several strings that are not usual for malware. Strings imul.exe and emul.exe might be the result of a hash collision and not represent the process that the hash is designed to detect.

Along with checking for well-known research tools, a lot of samples employs detection of AV by asking for a handle of specific dlls that might be loaded in the address space of malware (Figure 5).

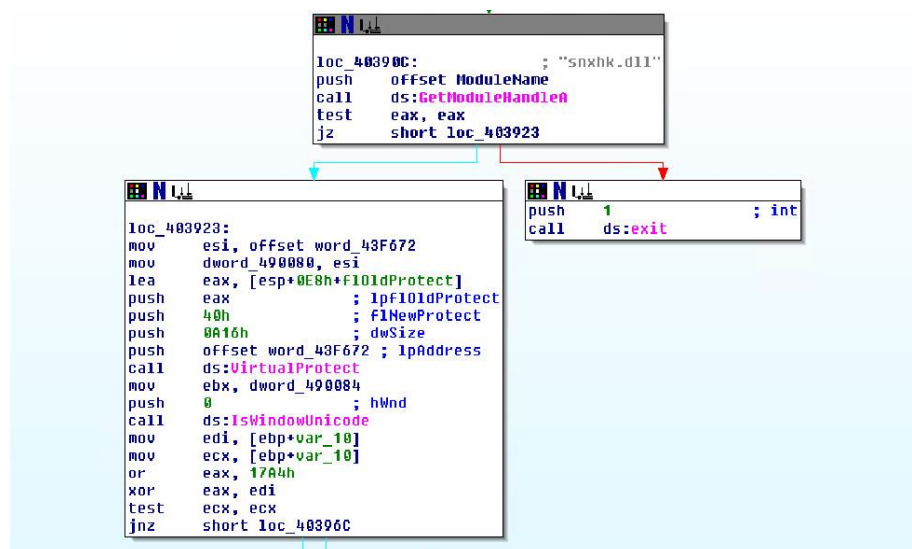


Figure 5. Check for Avast's snxhk.dll module in Nymaim<sup>3</sup>

To make evasion even more stronger, malware authors inspired by famous Red-Pill and No-Pill techniques [13][14] keep looking for new methods. According to [15], there a huge set of instructions that behaves differently in VMs. For example, instruction *xgetbv* which is used in Locky malware causes a crash in the unpatched version of QEMU (Figure 6).

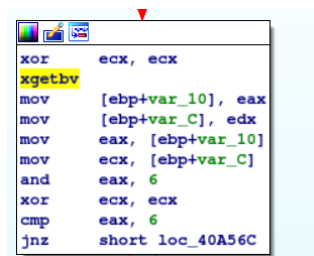


Figure 6. Locky<sup>4</sup>'s xgetbv instruction which cause a crash in QEMU

In some cases, malware even doesn't need to use anti-research techniques since a malicious activity is triggered only in a special case related to user activity such as a special event or mouse move. Figure 7 is an example of such technique. The malware will start infection only

<sup>3</sup> 8c9cb6d55539b28d8782eed64a75b75ad6f8fb76d651bc66c7641a591aa4163d

<sup>4</sup> 99513ee1c55ac5c5bab87e29c6291b50

if WM\_QUIT message exists in a message queue (a variable at [ebp+var4] is a flag that is used to trigger an infection routine).

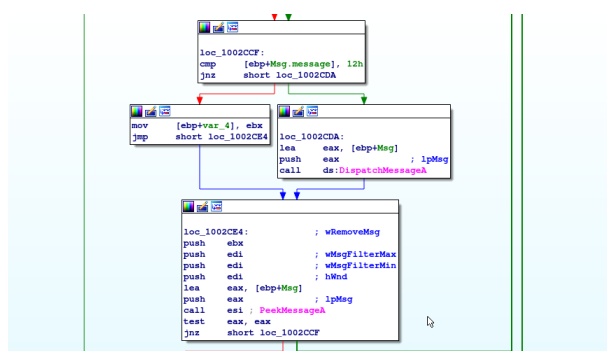


Figure 7. The Trojan.Win32.Yakes.oqml is looking for WM\_QUIT message

Another example of event-based check is presented in Figure 8. The sample checks for mouse cursor coordinates and launches infection only if cursor position was changed.

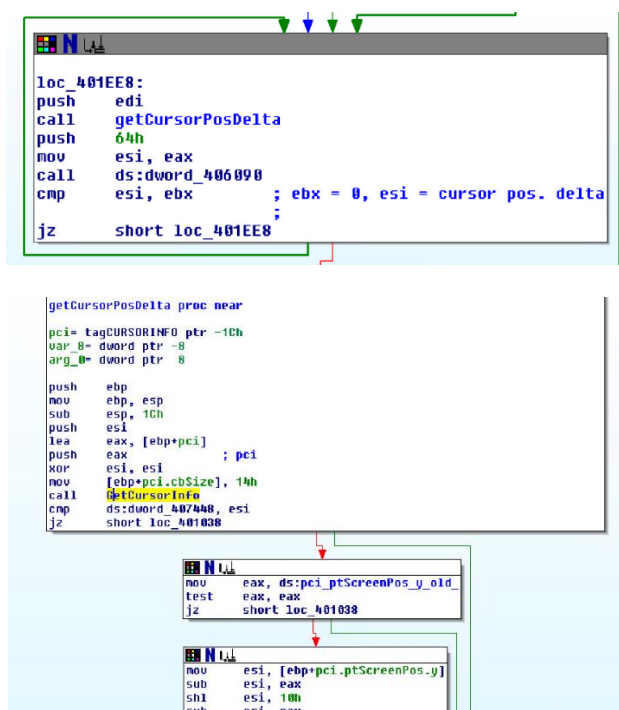


Figure 8. An example of mouse move check in Trojan.Win32.Scar.lwzh<sup>5</sup>

Some samples that are target only specific victims even doesn't need to prevent against automatic analysis in sandboxes because the malicious behavior itself is triggered only in special cases.

For example, our recent sample caught by [IBM Trusteer](#) shown in Figures 9-11 is targeted only specific users of Brazilian banks. The infection starts only if a web-browser appears at foreground and a window title match some of the hardcoded and encrypted strings such as *Santander* or *Internet Banking BNB*.

<sup>5</sup> e86023bf959fc851e14447439fded274

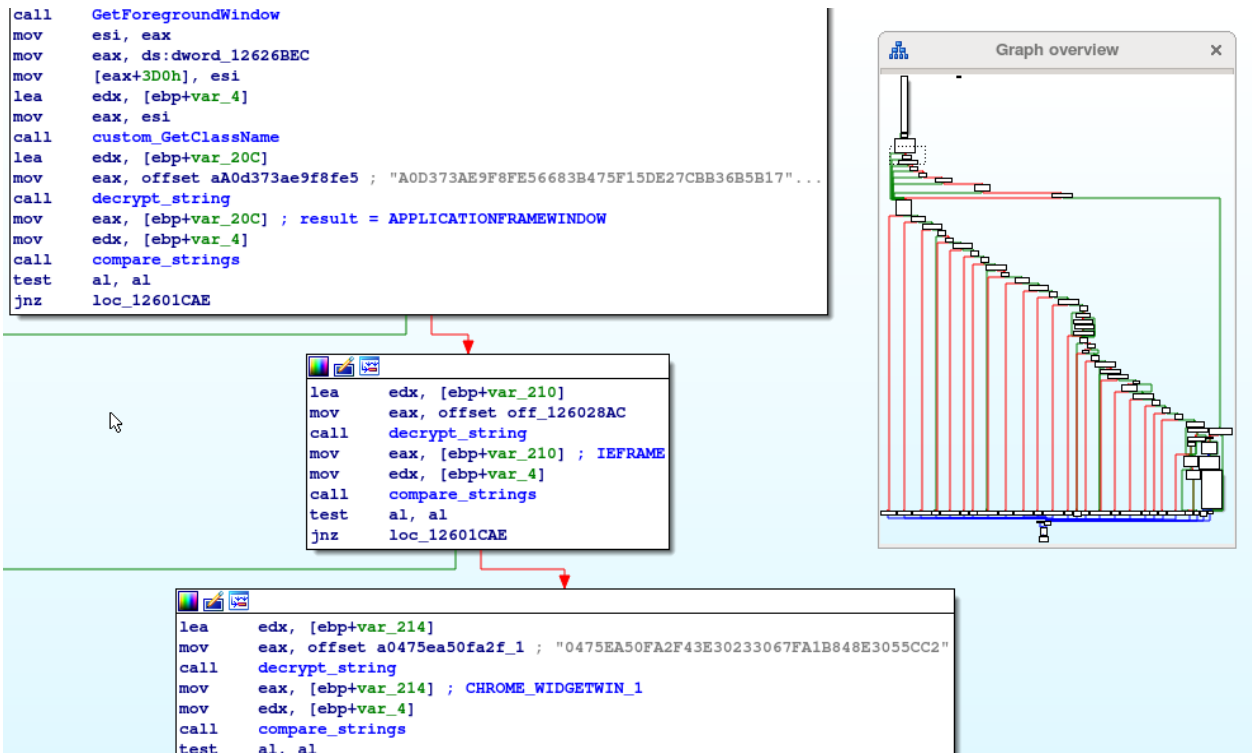


Figure 9. The sample is checking a class of window at foreground looking for specific application class name such as Internet Explorer or Google Chrome.

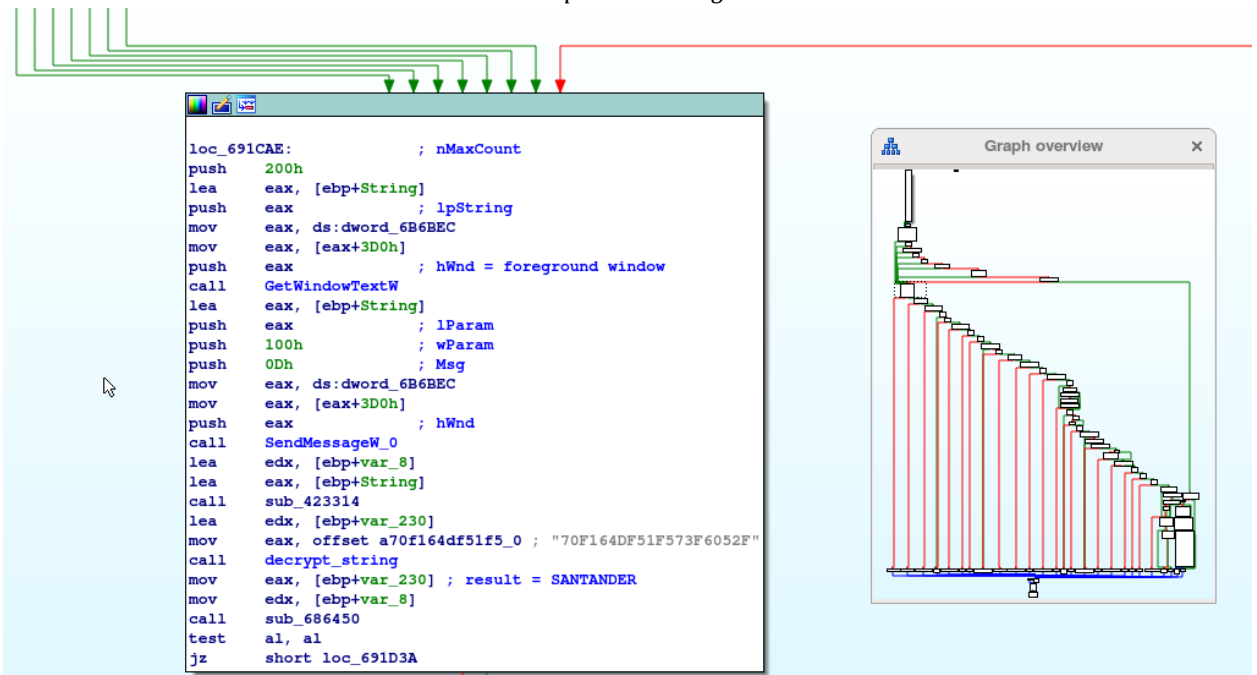


Figure 10. The sample is checking a title of foreground window looking for bank's name SANTANDER



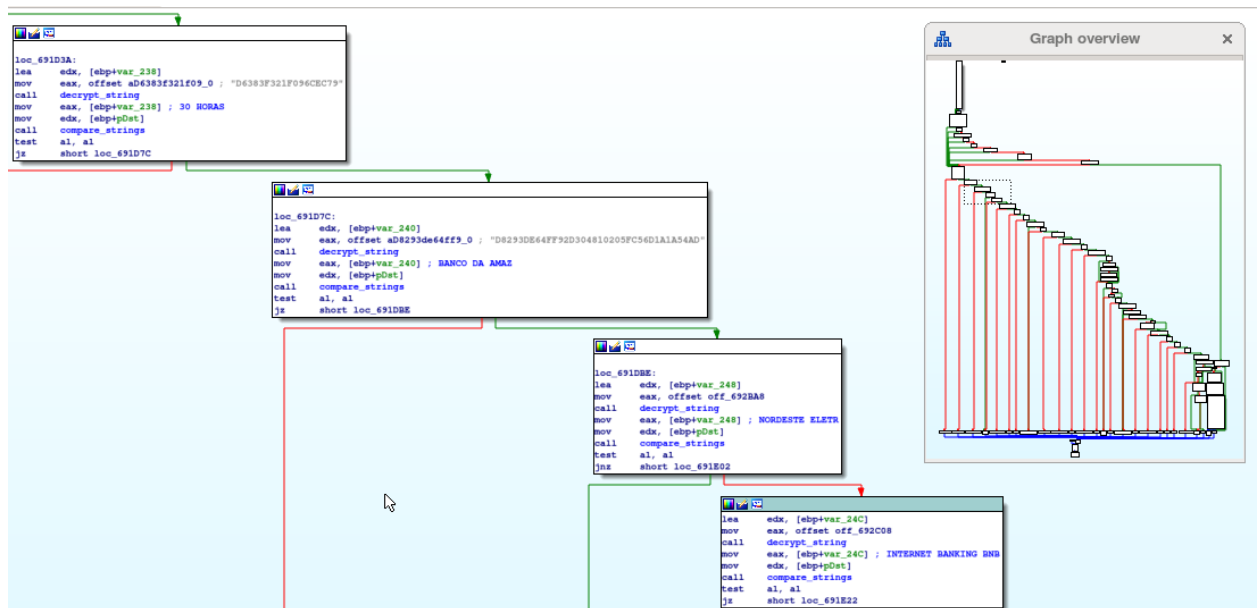


Figure 11. The sample is checking a title of foreground window looking for various Brazilian banks

The sample is looking for window classes of all popular web-browsers (*Internet Explorer*, *Google Chrome*, *Mozilla Firefox*) to be appeared at user's screen foreground. They are also checking for *Windows 10 Store Container* (APPLICATIONFRAMEWINDOW class name) window and *Java* (SUNAWTFRAME). When one of the required window exists on foreground, the sample moves to the next step.

So, the whole algorithm of this part of malware may be represented as following:

1. Get handle of a foreground window.
2. Get class name of a foreground window.
3. Sequentially compare class name with each decrypted string: IEFAME, CHROME\_WIDGETWIN\_1, MOZILLAWINDOWCLASS, BUTTONCLASS, MAKROBROWSER, SUNAWTFRAME.
4. If the strings are equal, it jumps to step 5, if they don't, it returns to step 1 after a short delay.
5. Get text title of a foreground window.
6. Sequentially compare the title with each decrypted string (e.g. BANCO BRADESCO).
7. If the strings are equal, the malware begin an actual malicious activity; if they are not equal, it returns to step 1 after a short delay.

Thus, the automatic-detection of such event-based or environmental-based samples is possible only if we can simulate an actual behavior of a user in the OS. Moreover, authors of the last demonstrated sample employ a scheme where benign executable is used to load a malicious dll only from a specific folder. In this case, a sandbox should decide where to put the malicious dll, otherwise the sample will not begin execution of malicious code at all.

### 3.2. Anti-emulation

The second class of anti-research techniques that I want to describe is anti-emulation. To avoid automatic unpacking in AV's emulator, a lot of samples try to bypass it by using a huge set of various libraries calls and/or pass in wrong parameters to them (Figure 12). So, if an unusual library call such as *GetIpForwardTable* is used, an emulator will not know how to correctly handle it thereby give a chance for malware to detect or stop emulator. Of course,

the second important reason to use a various API calls from different libraries is to make an import table look benign thereby preventing against static detection (but this is completely different story).

```
push    eax                ; hbr
lea     edx, [esp+4084h+rcDst]
push    edx                ; lprc
push    0                  ; hDC
call    edi ; FillRect
```

Example A

```
push    0                  ; bOrder
lea     eax, [esp+4084h+pdwSize]
push    eax                ; pdwSize
push    0                  ; pIpForwardTable
mov     [esp+408Ch+pdwSize], 0
call    GetIpForwardTable
```

Example B

```
push    5                  ; uCmd
push    0                  ; hWnd
mov     [esp+4088h+hWndParent], eax
call    ds:GetWindow
mov     edx, nIDDlgItem
mov     eax, hWnd
push    edx                ; nIndex
push    eax                ; hWnd
mov     [esp+4088h+wParam], eax
call    ds:GetWindowWord
```

Example C

```
push    ecx
push    0
push    offset aUtf16      ; "utf-16"
push    0
push    0
mov     [esp+4098h+hbr], 0
mov     dword ptr [esp+4098h+pszAgentW], edx
call    CreateXmlReaderInputWithEncodingName
```

Example D

Figure 12. Nymaim<sup>6</sup> uses various unusual WinAPI library calls with incorrect arguments to bypass emulation and static analysis

Some authors improve this technique and check for very specific details related to some library calls e.g. a state of registries (Figure 13) or values in a previous stack frame after specific library call (Figure 14). Even if an emulator correctly reproduces WinAPI calls, it also needs to correctly setup a CPU context after a call and sometimes even reproduce stack values which is very hard to emulate correctly for all possible cases.

For example, trojan Upatre sets `ecx` register to zero, performs WinAPI call to `CreateFile` and then checks `ecx` expecting it to be non-zero. Authors of this trojan knew that `CreateFileA` anyway changes `ecx` and the result after return from the `kernel32.dll` shouldn't be zero.

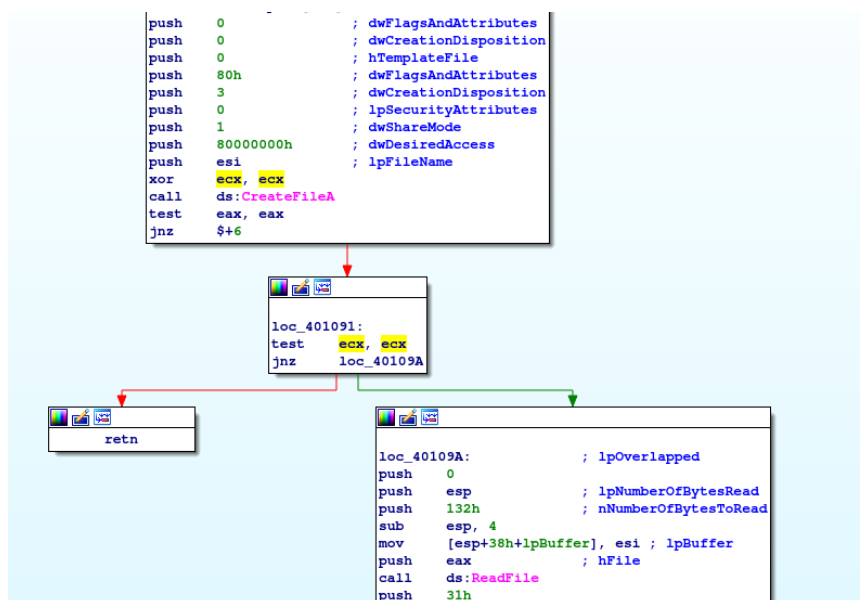
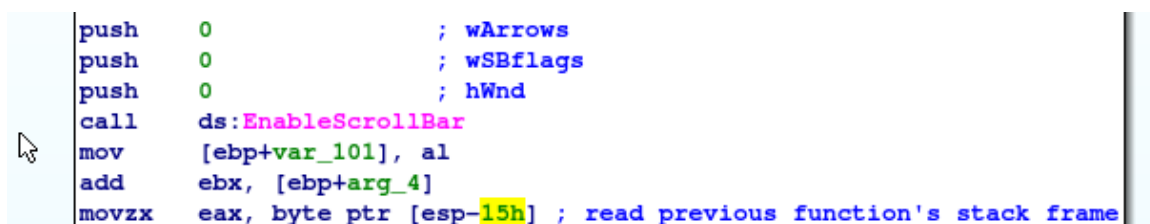


Figure 13. Trojan-Downloader.Win32.Upatre.ergs<sup>7</sup> checks for `ecx` to be non-zero after `CreateFileA`

<sup>6</sup> 48d7e9385ba7708019e1817c6fe76f706bb36dd1433b017b9b6330436efe0606

<sup>7</sup> 7598575b607dd61444911614640fccd4619aaf8ec2fe2102af0c40dc31b8f43f

The same approach is used by ransomware TeslaCrypt, the malware calls EnableScrollBar, reads previous function's stack frame and then later checks the result to be 0xFF. In its turn, EnableScrollBar has \_\_seh\_prolog4 function that always installs 0xFFFFFFFF (-2) at this stack address. So, authors were confident that the value at this address would be always equal 0xFF (at least in this version of Windows) as shown in Figure 14-16.



```

push    0                ; wArrows
push    0                ; wSBflags
push    0                ; hWnd
call     ds:EnableScrollBar
mov     [ebp+var_101], al
add     ebx, [ebp+arg_4]
movzx   eax, byte ptr [esp-15h] ; read previous function's stack frame

```

Figure 14. The Ransom.TeslaCrypt!g2<sup>8</sup> reads the specific value from the EnableScrollBar stack frame

```

; BOOL __stdcall EnableScrollBar(HWND hWnd, UINT wSBflags, UINT wArrows)
public EnableScrollBar
EnableScrollBar proc near                ; DATA XREF: .text:off_7DC60570fo

var_20      = dword ptr -20h
var_1C      = dword ptr -1Ch
ms_exc      = CPPEH_RECORD ptr -18h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h

; FUNCTION CHUNK AT .text:7DC78DB9 SIZE 0000000C BYTES
; FUNCTION CHUNK AT .text:7DC9FC3A SIZE 00000028 BYTES

push    10h
push    offset stru_7DC78D98
call    __SEH_prolog4

```

Figure 15. EnableScrollBar from user32.dll calls \_\_SEH\_prolog4 at the beginning of execution

```

__SEH_prolog4 proc near                ; CODE XREF: sub_7DC66A95+7↓p
                                        ; sub_7DC66C83+7↓p ...

arg_4      = dword ptr  8

push    offset sub_7DCCA61E
push    large dword ptr fs:0
mov     eax, [esp+8+arg_4]
mov     [esp+8+arg_4], ebp
lea     ebp, [esp+8+arg_4]
sub     esp, eax
push    ebx
push    esi
push    edi
mov     eax, __security_cookie
xor     [ebp-4], eax
xor     eax, ebp
push    eax
mov     [ebp-18h], esp
push    dword ptr [ebp-8]
mov     eax, [ebp-4]
mov     dword ptr [ebp-4], 0FFFFFFFh ; the value to check
mov     [ebp-8], eax
lea     eax, [ebp-10h]
mov     large fs:0, eax
retn
__SEH_prolog4 endp

```

Figure 16. The \_\_SEH\_prolog4 function installs the value -2 in the stack

While these techniques look like an effective trick to bypass emulation, a strong emulator in theory might correctly implement a required subset of WinAPI calls or perform emulation at system calls level. To handle this problem, malware writers usually use some methods to delay execution using for example WinAPI function Sleep or long loops. AV's emulator has

<sup>8</sup> e769ccbf7fe749e10ac4356a26a289506123f4e230cf2bfbcaf8cfea467ccede

full control on execution, so it can just avoid long sleeps and use some heuristics to bypass through the long loops.

However, there are a lot of ways to fool the anti-loop heuristic by implementing a complex stalling code. An example of such stalling code has shown in Figure 17. ZBot has a long loop at the beginning of the execution comparing `cycle_count` with `0x2DC6C0` and calling twice function at `[ebp+var_64]` that contains a code that unpack 1 byte of malicious payload per call. The sample uses this function in many places during execution thereby preventing against fast unpacking in emulators.

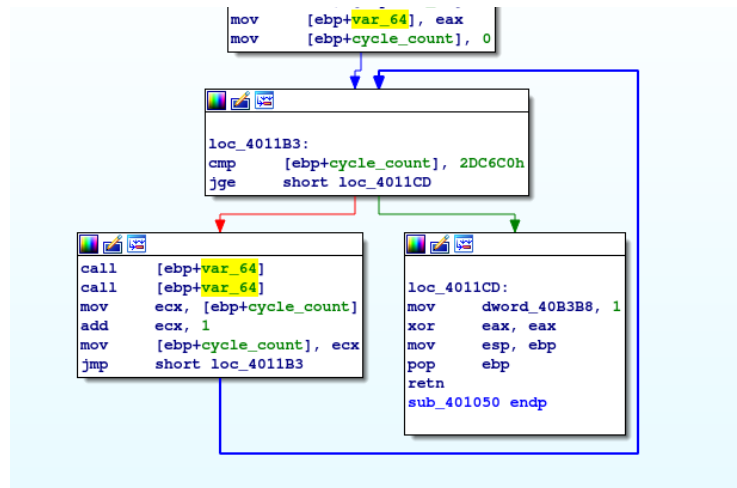


Figure 17. Stalling code in ZBot<sup>9</sup>

Nymaim also adds several API calls that do nothing to slowdown emulators even more (especially in case of system call level emulation) and fool heuristics (Figure 18).

```

loc_402AF8:                                ; CODE XREF: WinMain(x,x,x,x)+193D↓j
cmp     cchMax, 0
jnz     loc_402B8A
push    0                                ; cchSize
lea     eax, [esp+4084h+Buf]
push    eax                                ; Buf
lea     ecx, [esp+4088h+pszAgentW]
push    ecx                                ; LPCWSTR
mov     dword ptr [esp+408Ch+pszAgentW], ebx
mov     dword ptr [esp+408Ch+Buf], 0
call    ds:GetFileTitleW ; GetFileTitle("", pBuffer, buflen) - always false
lea     edx, [esp+4080h+var_14E0]
movsx   edi, ax
mov     eax, uValue
push    edx                                ; lpString
push    eax                                ; nIDDlgItem
push    ebx                                ; hDlg
call    ds:SetDlgItemTextA ; SetDlgItemTextA(NULL, WrongHandle, "") - always false
cmp     edi, ebx                            ; compare result of GetFileTitleW with ebx = 0
jge     short loc_402B77
cmp     eax, hWnd                            ; compare result of SetDlgItemTextA with hWnd = 0
jnz     short loc_402B77

loc_402BA0:                                ; CODE XREF: WinMain(x,x,x,x)+18F7↑j
; WinMain(x,x,x,x)+1905↑j ...
mov     eax, [esp+4080h+counter] ; counter
inc     eax
cmp     eax, [esp+4080h+counter_max_value] ; compare counter with 0x39f64
mov     [esp+4080h+counter], eax
jl      loc_402AF8

```

Figure 18. An example of stalling code in the Win32/TrojanDownloader.Nymaim.BA<sup>10</sup>

<sup>9</sup> 350791c6b9a7ae91869fa09d042f734dc37c864c661374dcce3161dffa6d7fc5

<sup>10</sup> 48d7e9385ba7708019e1817c6fe76f706bb36dd1433b017b9b6330436efe0606

Locky's authors made a bet on intensive usage of recursion probably to prevent against automatic stalling-loops bypassing (Figure 19) and significantly decrease a performance of emulator. This stalling function is used in many places of the malware especially during initialization before unpacking routine.

```

1 int __thiscall stalling_func(int arg1)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     tmp_arg1 = arg1;
6     if ( arg1 )
7     {
8         if ( arg1 == 1 )
9         {
10             result = 1;
11         }
12         else
13         {
14             res = stalling_func(0);
15             res2 = res + stalling_func(tmp_arg1 - 1) + 1;
16             res3 = stalling_func(1) + 1;
17             res4 = res3 - (stalling_func(0) + 1);
18             res5 = res4 + stalling_func(1) + 1;
19             result = res5 + stalling_func(tmp_arg1 - 2) - res2;
20         }
21     }
22     else
23     {
24         result = 0;
25     }
26     return result;
27 }

```

Figure 19. Locky<sup>11</sup>'s stalling recursive function

On a real processor, the functions described above will be executed in several seconds or minutes while emulator may spend dozens of minutes or even hours trying to pass through the cycles thereby preventing against automatic unpacking and detection.

#### 4. Possible Solutions

As we mentioned in second part of the white paper, a creation of a transparent towards analyzed executable sandbox is a complex, probably unsolvable task. However, in this section we will discuss possible solutions that can significantly complicate an evasion and detection.

As shown in the previous sections, many detection mechanisms are based on some specific configuration details of a sandbox (e.g. username or BIOS version). The easiest solution is to replace all standard configuration details with custom ones. However, as it was shown by Gootkit, an attacker may collect this custom configuration details and then use them against sandbox in the next versions of their malware.

To handle this problem, the concept of moving target is a good decision. Instead of using some static information, we can leverage a randomness in our configuration details, e.g. Windows username, MAC-address and BIOS version may change at each run of the VM or emulator thereby significantly increasing complexity of detection.

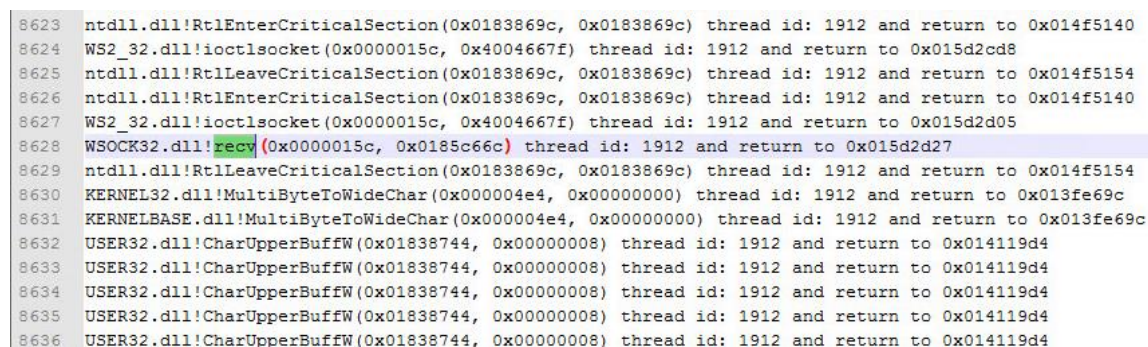
To break a semantic gap when malware is executing in VM, instead of WinAPI-hooking, we may apply dynamic binary instrumentation approach for malware behavior tracing. The author has implemented a tool for WinAPI library calls tracing called *drltrace* [16]. The tool has been implemented on top of DynamoRIO framework and is distributed under BSD-

<sup>11</sup> 99513ee1c55ac5c5bab87e29c6291b50

license. The tool is very simple to use, it is enough to specify the following command to begin instrumentation:

```
drltrace.dll -only_from_app -logdir . -- <app_name>
```

As a result, the tool will produce a log file that contains all library calls that application performs during the execution (Figure 20).



```
8623 ntdll.dll!RtlEnterCriticalSection(0x0183869c, 0x0183869c) thread id: 1912 and return to 0x014f5140
8624 WS2_32.dll!ioctlsocket(0x0000015c, 0x4004667f) thread id: 1912 and return to 0x015d2cd8
8625 ntdll.dll!RtlLeaveCriticalSection(0x0183869c, 0x0183869c) thread id: 1912 and return to 0x014f5154
8626 ntdll.dll!RtlEnterCriticalSection(0x0183869c, 0x0183869c) thread id: 1912 and return to 0x014f5140
8627 WS2_32.dll!ioctlsocket(0x0000015c, 0x4004667f) thread id: 1912 and return to 0x015d2d05
8628 WSOCK32.dll!recv(0x0000015c, 0x0185c66c) thread id: 1912 and return to 0x015d2d27
8629 ntdll.dll!RtlLeaveCriticalSection(0x0183869c, 0x0183869c) thread id: 1912 and return to 0x014f5154
8630 KERNEL32.dll!MultiByteToWideChar(0x0000004e4, 0x00000000) thread id: 1912 and return to 0x013fe69c
8631 KERNELBASE.dll!MultiByteToWideChar(0x0000004e4, 0x00000000) thread id: 1912 and return to 0x013fe69c
8632 USER32.dll!CharUpperBuffW(0x01838744, 0x00000008) thread id: 1912 and return to 0x014119d4
8633 USER32.dll!CharUpperBuffW(0x01838744, 0x00000008) thread id: 1912 and return to 0x014119d4
8634 USER32.dll!CharUpperBuffW(0x01838744, 0x00000008) thread id: 1912 and return to 0x014119d4
8635 USER32.dll!CharUpperBuffW(0x01838744, 0x00000008) thread id: 1912 and return to 0x014119d4
8636 USER32.dll!CharUpperBuffW(0x01838744, 0x00000008) thread id: 1912 and return to 0x014119d4
```

Figure 20. An example of output from drltrace. The screenshot demonstrates successful log about malware communication attempt with CnC server.

It should be noted that DBI technique in general is very easy to detect as shown in this works [17][18]. However, while this technique is not standard in malware analysis, there is a possibility to use it for most of the samples.

## Conclusion

The evolution history of malware demonstrates that their authors will keep investing in new ways to bypass modern malware analysis solutions. Equipped with advanced evasion techniques, malware owners significantly increase a lifecycle of their campaigns. Security community should keep investing in flexible and robust automatic malware detection sandboxes that may significantly help to perform early detection of new threats.

## References

- [1] Marta Janus. Going Gets Tough: A Tale of Encounters With Novel Evasive Malware. REcon 2014.
- [2] Dilshan Kiragala. Detecting Malware and Sandbox Evasion Techniques. SANS Institute InfoSec Report 2016. <https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667>
- [3] Christopher Kruegel. Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware. BlackHat USA 2014.
- [4] Egele, Manuel, et al. "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools." ACM Computing Surveys (CSUR) 44.2 (2012): 6.
- [5] Alexei Bulazel. AVLeak: Fingerprinting Antivirus Emulators for Advanced Malware Evasion. Black Hat USA 2016.
- [6] Christopher Kruegel. How to Build an Effective Malware Analysis Sandbox. <https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/>
- [7] Dolan-Gavitt, Brendan, et al. "Repeatable Reverse Engineering with PANDA." Proceedings of the 5th Program Protection and Reverse Engineering Workshop. ACM, 2015.
- [8] Comodo Antivirus Forwards Emulated API Calls to the Real API During Scans. <https://bugs.chromium.org/p/project-zero/issues/detail?id=769>



- [9] Tal Garfinkel, et al. "Compatibility Is Not Transparency: VMM Detection Myths and Realities." HotOS. 2007. [http://static.usenix.org/legacy/events/hotos07/tech/full\\_papers/garfinkel/garfinkel.html/](http://static.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel.html/)
- [10] Zhang, Fengwei, et al. "Using Hardware Features for Increased Debugging Transparency." Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, 2015.
- [11] Spensky, Chad, Hongyi Hu, and Kevin Leach. "LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis." (2016).
- [12] Limor Kessem. Unraveling Gootkit's Stealth Loader <https://securityintelligence.com/news/unraveling-gootkits-stealth-loader/>
- [13] Rutkowska, Joanna. "Red Pill... or How to Detect VMM Using (almost) One CPU Instruction." (2004). <http://www.securiteam.com/securityreviews/6Z00H20BQS.html>
- [14] Danny Quist. Detecting the Presence of Virtual Machines Using the Local Data Table. [http://charette.no-ip.com:81/programming/2009-12-30\\_Virtualization/](http://charette.no-ip.com:81/programming/2009-12-30_Virtualization/)
- [15] Martignoni, Lorenzo, et al. "Testing System Virtual Machines." Proceedings of The 19th International Symposium on Software Testing and Analysis. ACM, 2010.
- [16] <https://github.com/DynamoRIO/drmemory/tree/master/drltrace>
- [17] Francisco Falcon and Nahuel Riva. Detecting Binary Instrumentation Frameworks. REcon 2012.
- [18] Ke Sun, Xiaoning Li, Ya Ou. Break Out of the Truman Show. Active Detection and Escape of Dynamic Binary Instrumentation. BlackHat Asia 2016.