Malware: Analysis, Anti-analysis, and Anti-anti-analysis

1

Abstract. *Malware are persistant threats to computer systems, and their damages extends from image issues to financial losses. Malware analysis is a set of techniques which aim to provide information used on forensic procedures and countermeasures development. In this scenario, criminals employ anti-analysis techniques in order to prevent their samples from being analyzed. These techniques rely on instructions' side effects and system's structures checks to be inspection-aware. By detecting an analysis procedure, a sample can evade inspection and remain itself stealth. Thus, detecting these evasion attempts is an important step for an investigative procedure. In this work, we present a study of anti-analysis techniques as well as their detection counterparts. We evaluated NUMBER samples in order to build an evasion-aware malware panorama.*

1. Introduction

Malicious software, also known as malware, are pieces of software with malicious purposes. Their actions can vary from data exfiltration to persistent monitoring, causing damages to both private and public institution, either on image or financial aspects. According to CITATION statistics, malware incidents answer by ....

Given this scenario, analysts are required to analyze malicious samples in order to provide both defensive mechanisms as well as forensic procedures, on already compromised systems. The set of techniques used for such kind of inspection is known as malware analysis. Analysis procedures can be classified into static, where no code is ran, and dynamic, where code is run on controlled environment [Sikorski and Honig 2012]. The scope of this work is limited to static procedures. Dynamic approaches will be left as future work.

Considering the malware analysis capabilites, criminals began to protect their artifacts from being analyzed. This way, their infection could last longer since they could make their samples stealth. Recent studies REFS shows that DATA% of samples employs anti-analysis techniques, and this number has grown...

In order to keep protected from such new armored threats, we need to understand how these anti-analysis techniques work and develop ways to detect evasive samples, what is called anti-anti-analysis. This paper presents the operation of such kind of techniques as well as detection methods. We evaluated the developed solution agains NUMBER samples, benign and malicious, allowing us to build an evasive scenario panorama. We also compared evasive technique used on distinct countries, which can help them to be ahead of coming threats.

Figure 1, 2, and 3 show real evasion we have detected...

This work is organized as follows: Section 2 presents the basic concepts related to anti-analysis techniques; section 3 presents related work and tools aimed to detect anti-analysis techniques; section 4 presents an study of how distinct evasion techniques work;

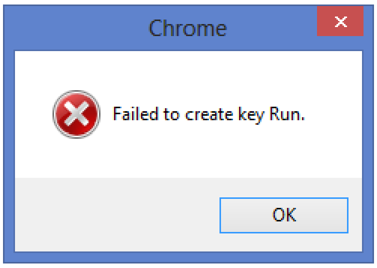

Figure 1. 1.

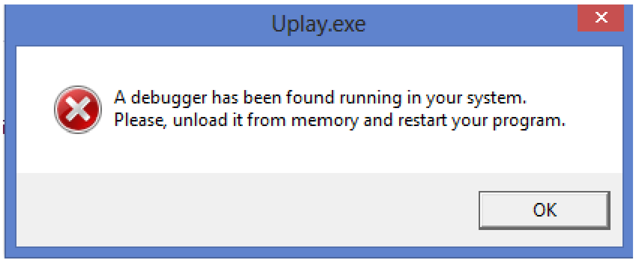

Figure 2. 2.

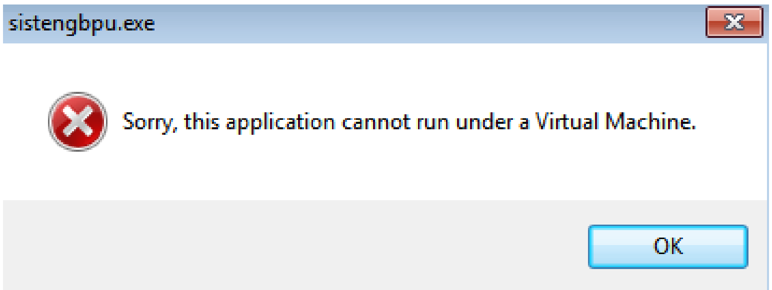

Figure 3. 3.

section 5 presents our detection framework; section 6 presents the results obtained from applying our solution to distinct datasets; finally, section 10 presents concluding remarks and future works.

2. Background

The main idea of anti-analysis techniques is to raise the bar of counteraction methods. It can be done in many ways, from analysis evasion to leveraging theorethical hard-to-compute constructions. This section gives an overview of such techniques.

One approach is to fingerprint the analysis environment. Known analysis solutions present regular pattern such as fixed IP addresses, host name and serial numbers. Evasive samples can detect those patterns and suspend its execution [Yokoyama et al. 2016]. This approach was used against Cuckoo [Ferrand 2015] and Ether [Pék et al. 2011] sandboxes.

Another approach is to evade analysis by detecting execution side effects of virtual machines and emulators, the most used solutions for malware analysis. Those systems presents a diverting behavior than their bare-metal counterparts, such as instructions not being atomic [Willems et al. 2012]. Currently, there are automated ways of detecting such side effects [Paleari et al. 2009].

Virtual Machines can also be detect by the changes it perform on system internals, such as table relocations. Many tables, such as Interrupt Descriptor Table (IDT), have their addresses changed on VMs when campared to bare-metal systems. Those address can be used as an indicator of a virtualized environment [Ferrie b].

There also approaches based on not evading the analysis itself, but on hardening the post-infection reverse engineering procedure. One notable technique is the anti-disassembly, a way of coding where junk is inserted among legitimate code in order to fool the disassembler tool. Another variation of anti-disassembly technique is to used opaque constants [Kruegel et al. 2007], construction which cannot be solved without runtime information. Static attempts to guess resulting values of such expressions tend to lead to the path explosion problem [Xiao et al. 2010].

Finally, there are samples that leverage time measurement for analysis detection, since any monitoring technique imposes significant overhead [Lindorfer et al. 2011]. Although some solutions try to mitigate this problem by faking time measures, either on system APIs [Singh] or on hardware timestamp counter [Hexacorn], the problem is unsolvable in practice since an advanced attacker can make use of an externel NTP server over encripted connections.

The anti-anti-analysis techniques, in turn, are also classified as static and dynamic approaches. The static approaches can be used as pattern matching detectors of known anti-analysis constructions, such as address verifications and locations. However, due to its known limitations, some constructions can only be solved on runtime, performed on dynamic environments.

Dynamic solutions, in a general way, are based on faking answers for known anti-analysis checks, such as in COBRA [Vasudevan and Yerraballi 2006]. These approach, however, turns into an arms-race, since new anti-analysis techniques are often released and these systems need to be updated. Recently, transparent analysis systems have been proposed, such as Ether [Dinaburg et al. 2008] and MAVMM [Nguyen et al. 2009]. These systems, however, impose high overhead and development costs.

In the following sections, we study the anti-analysis techniques for the above presented classes and present static detectors for these techniques. We also discuss the limitations of each one, how they can be overcomed, and how other solutions implement or not these detectors. Dynamic detectors implementations are left as future work.

3. State-of-the-art and Related Work

In this section, we present current tools and implementations of anti-analysis and anti-anti-analyiss techniques. A notable anti-analysis tool is pafish [Pafish], a serie of modules which implement many of mentioned detection techniques, such as virtual machine detection and environment fingerprint. The intent of the tool is to be used as a transparent solution validator and to allow malware evasion techniques more understandable. In this work, we take a look on many of pafish techniques implementation.

Other solutions, such as pyew [Pyew] and peframe [Peframe], are intended to detect the evasive techniques. They statically look for known shellcodes and library imports which are related to analysis evasion. In this work, we look to their implementation and compare them against our proposals.

In addition to the aforementioned tools, our work is also closest related to the [Branco et al. 2012] one, which implemented many anti-anti-analysis detector and analyzed many of evasive samples. In this work, we have implemented both the anti-analysis as well as the detectors presented in it, applying them against our distinct dataset, and enriching their analysis with formal discussion of techniques working flow. We also proceed the same way regarding the work [Ferrie a]. By the time we were writing this article, we have noticed a related work implementing similar techniques [Oleg].

Other related approaches, although more complex, are those which relies on using intermediate representations (IR) [Smith et al. 2014] or interleaving instructions [Saleh et al. 2014].

This work does not cover obfuscation techniques based on encryption. This issue was addressed by other works, such as [Calvet et al. 2012].

4. Techniques

In this section, we discuss the anti-analysis techniques, their operation, and how they can be detected. The techniques were classifyied according their purpose: anti-disassembly, anti-debugging, and virtual machine detection.

4.1. Anti-disassembly

In this section, we present techniques aimed to makes anti-disassembly harder.

4.1.1. A View on Disassembling

In a general way, diassemblers can be classified into linear sweep and recursive traversal approaches [Schwarz et al. 2002]. On the linear sweep approach, used on tools such as objdump, WinDbg, and softICE, the disassembly process starts at first byte of .text section and proceeds sequentially. The major limitation of this approach is that any data embedded in the code is interpreted as instruction, leading to a wrong final disassembled code.

The recursive approach takes into account the control flow of the program being disassembled, following the possible paths from the entry point, which solves part of problems presented by the linear approach, such as identifying jmp-preceeded data as code. The major assumption of this approach is that it is possible to identify all sucessors of a given branch, which is not always true, since any fail on identifying instruction sizes can lead to incorrect paths and instructions.

On the following subsections, we study techniques able to explore the drawbacks of both approaches and lead to incorrect disassembly, hardening the analysis procedures.

Table 1 shows the disassembling approach used by distinct tools.

Table 1. Based on [Eliam 2005]

Tool	Technique
OllyDbg	Recursive traversal
NuMega	SoftICE Linear sweep
Microsoft WinDbg	Linear sweep
IDA Pro	Recursive traversal
PEBrowse	Recursive traversal

4.1.2. PUSH POP MATH

In order to obfuscate a value, samples can make use of indirect values manipulation, such as using PUSH and POP instruction. On this technique, a known immediate is pushed into the stack and then popped to an register. This register is also used on further computation, as shown on Listing 1.

Listing 1. PUSH POP MATH

```
        push    0x1234
        pop     rax
        xor     rax, 0xFFFF
```

In order to detect this technique, we can look for the sequence of PUSH, pop to an register, and a computation over such register, as shown on Listing 2.

Listing 2. PUSH POP MATH detection

```
if 'push' in instruction and not op1 in ['ax','bx','cx','dx']:
            self.found_push=True
        elif 'pop' in instruction and self.found_pop==False:
            self.found_pop=True
            self.found_op=op1
        elif self.found_pop==True and instruction in ['and', 'or','xor']:
            if self.found_op in op1 or self.found_op in op2:
                self.found_comp=True

        if self.found_comp==True:
            self.clear()
            print "\"PushPopMath\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.1.3. PUSH-RET Instruction Replacement

Instruction replacement is a common evasive technique since it makes harder to follow the execution control flow. One of many variations of this technique is to replace CALLs and ordinary returns by a sequence of PUSH and RET, where an address is inserted into the stack and thus the function return to it. This sequence is shown on Listing 3.

Listing 3. PUSH RET

```
        push    0x12345678
        ret
```

The sequence of PUSH and RET can be used to detect the usage of such technique, as shown on Listing 4.

Listing 4. PUSH RET detection

```
        def check(self, section, address, instruction, op1, op2):

        if 'push' in instruction:
            self.found_push=True
        elif self.found_push==True and 'ret' in instruction:
```

```
                self.found_ret=True
        else:
                self.found_push=False

        if self.found_ret:
                self.clear()
                print "\"PushRet\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.2. LDR Address resolving

LDR is a PEB internal structure which contains information about loaded modules [Microsoft d], as shown on Listing 5. When looking to the PEB structure, the LDR information can be found at the offset `0x0c`, as shown on Listing 6.

Listing 5. LDR struct

```
typedef struct _PEB_LDR_DATA {
        BYTE       Reserved1[8];
        PVOID      Reserved2[3];
        LIST_ENTRY InMemoryOrderModuleList;
    } PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Listing 6. Peb ldr

```
typedef struct _PEB {
        BYTE                              Reserved1[2];
        BYTE                              BeingDebugged;
        BYTE                              Reserved2[1];
        PVOID                             Reserved3[2];
        PPEB_LDR_DATA                     Ldr;
```

This way, some samples could try to access this information directly, by loading the respective addresses, as shown on Listing 7.

Listing 7. ldr direct

```
        mov     eax, [fs:0x30]
        mov     eax, [eax+0x0c]
```

In order to detect this usage, we can check for usages of the PEB and LDR offsets, respectively, as shown on Listing 8.

Listing 8. ldr direct detect

```
def check(self, section, address, instruction, op1, op2):

        if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction in ['cmp', 'cmpxchg', 'mov', 'movsx',
                'movzx']:
                if '[' + self.found_op1 + '+0xc]' in op1 or '['
                   + self.found_op1 + '+0xc]' in op2:
                    self.clear()
                    print "\"LDR\" Detected! Section: <%s>
                        Address: 0x%s" % (section, address)
```

4.2.1. Stealth Windows API Import

Windows API is the basic toolchain for development...However, its usage can provide many information about a program behavior. This way, stealthly using system API increases sample stealthness.

One way to implement such stealth function imports is to rely on `ntdll` and `kernel32` libraries which are automatically mapped into process, wven without any explicit import.Those libraries are accessible through a walk over the process memory [Lyashko].

We can start retrieving handlers by getting the SEH chain offset (`0x0`), as shown on Listing 9, and thus iterating over it until the end, one of these libraries, to retrieve the handler at the `0x4` offset, as shown on Listing 9.

Listing 9. SEH base address

```
        mov eax,[fs:0]
        .search_default_handler:
        cmp dword[eax],0xFFFFFFFF
        jz .found_default_handler
        ;go to the previous handler
        mov eax,[eax]
        jmp .search_default_handler
        mov eax,[eax+4]
```

```
        and eax,0xFFFF0000
```

So, we can check for the MZ signature over the pages, as shown on Listing 10.

Listing 10. finding pe

```
.look_for_mz:
    cmp word [eax],'MZ'
    jz .got_mz
    sub eax,0x10000
    jmp .look_for_mz
```

At this point, we have a handle but we don't know for which library. A solution for that starts with a check on the `0x3c` offset, which contains an offset for a PE string signature followed by COFF data, as shown on Listing 11.

Listing 11. PE

```
    mov bx,[eax+0x3C]
    movzx ebx,bx
    add eax,ebx
    mov bx,'PE'
    movzx ebx,bx
    cmp [eax],ebx
    jz .found_pe
```

Once we got the COFF header, we can find the exports of the `IMAGE_DATA_DIRECTORY` at offset `0x78` and thus read the RVA and add it to the base address, allowing the use, as shown on Listing 12.

Listing 12. RVA

```
    add eax,0x78
    mov eax,[eax]
    add eax,[image_base_address]
```

By accessing the `0xC` offset we get the NAME RVA, a string containing the library address, and so discover if we are in kernel32 or ntdll, as shown on Listing 13.

Listing 13. name rva

```
        mov eax,[eax+0x0C]
        add eax,[image_base_address]
```

A detector can be implemented by checking of all these steps appears in sequence on a given code. Our detector looks like the one presented on Listing 14.

Listing 14. detector

```
    def check(self, section, address, instruction, op1, op2):

        if self.found_seh==False and instruction in ['mov', '
            movsx', 'movzx']:
            if 'fs:0x0' in op2:
```

```
                self.found_op1 = op1
                self.found_seh = True

        elif self.found_seh==True and self.found_handler==False and instruction in ['mov', 'movsx', 'movzx']:

            if 'fs:0x30' in op2:
                self.found_seh=False
            elif self.found_op1 + '+0x4' in op2:
                self.found_handler=True
                self.found_op2 = op1

        elif self.found_handler==True and instruction in ['cmp','cmpxchg']:
            if self.found_op2 in op1:
                self.found_cmp=True

        elif self.found_cmp==True and instruction in ['mov','movsx','movzx']:
            if self.found_op2+'+0x3c' in op2:
                self.found_pe=True

        elif self.found_pe==True and instruction in ['and','or','xor','add','sub','cmp']:
            if '0x78' in op1 or '0x78' in op2:
                self.found_img=True

        if self.found_img:
            self.clear()
            print "\"StealthImport\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.2.2. NOP

Another anti-disassembly technique is to add dead-code to the binary. Dead code are construction which are unreacheable or effectless intended only to make the anti-analysis process harder and to evade pattern matching detectors. A commmon dead code construction is a NOP sequence, as shown on Listing 15.

Listing 15. NOP.

```
        mov     eax, 0
        nop
        nop
        nop
        nop
        nop
        pop     rbp
```

A detector for this technique consists on finding a N-sized window ROP sequence, as shown on Listing 16.

Listing 16. NOP.

```
def check(self, section, address, instruction, op1, op2):

        if instruction == 'nop':

            self.counter += 1

            if self.counter is 5:
                self.counter = 0
                print "\"NOPSequence\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.2.3. Fake Conditional

Many solutions try to follow control flow in order to apply their detector. Making conditional control flow harder is a powerful anti-analysis techniques, since not all paths can be followed due to the path explosion problem.

One possible implementation for this evasive techique is to rely on flags computed by a previous known instruction. The example of Listing 17 shows a known-result instruction, since xor-ing the register will result on zera, followed by a zero-conditioned jump.

Listing 17. Fake Conditional.

```
xor eax, eax
jnz main
```

Detectors for this technique rely on detecting xor instructions followed by those kind of constructions, such as jmp, stc or clc, as shown on Listing 18.

Listing 18. Fake Conditional.

```
def check(self, section, address, instruction, op1, op2):

self.cycle_count += 1

if instruction == 'xor' and op1 == op2:
        self.found_xor = True
        self.xor_cycle = self.cycle_count
        return
elif instruction =='stc':
        self.found_stc= True
        self.stc_cycle=self.cycle_count
        return
elif instruction=='clc':
        self.found_clc=True
        self.clc_cycle=self.cycle_count
```

```
        return

if(instruction=='jnz' or instruction=='jne') and self.found_xor and self.cycle_count==self.xor_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
elif (instruction=='jnc' or instruction== 'jae') and self.found_stc and self.cycle_count == self.stc_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
elif (instruction =='jc' or instruction=='jb') and self.found_clc and self.cycle_count==self.clc_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
```

4.2.4. Control Flow

An anti-analysis variation of the JMP construction is to replace the unconditional JMP by other constructions, which can fool a linear disassembler. A common replacement is to pushing a value in to stack and then launching a RET instruction. This construction can be ssen on Listing 19.

Listing 19. ProgramCF.

```
mov     eax, 0
        push 0x2
        ret
        pop     rbp
```

A detector for this technique is to match push + ret constructions, as shown on Listing 20.

Listing 20. ProgramCF.

```
    def check(self, section, address, instruction, op1, op2):

        self.cycle_counter += 1

        if instruction == 'push':
            self.found = True
            self.found_cycle = self.cycle_counter
            return

        if self.found and instruction == 'ret' and self.cycle_counter == self.found_cycle + 1:
            self.clear()
```

```
            print "\"ProgramControlFlowChange\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.2.5. Garbage Bytes

A way to hide data inside binary is to set it right after an unconditional JMP, since it will be unreachable as code, as shown on Figure 4.

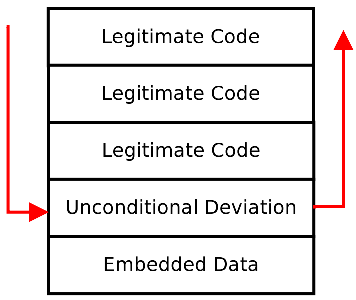

Figure 4. Dead Code.

The dead code insertion is intended to fool disassemblers which try to interpret such unreacheable bytes as code. Its usage is often associated with other control flow deviations anti analysis techniques, such as indirect jump. An implementation of this technique can be seen on Listing 21.

Listing 21. Garbage Bytes.

```
mov     eax, 0
push  0x3
ret
.data
```

The detector for this technique is based od the presented Control Flow and Fake Jump detectors, followed by additional bytes, as shown on Listing 22.

Listing 22. Garbage Bytes.

```
def check(self, section, address, instruction, op1, op2):

        self.cycle_counter += 1

        if instruction == 'push':
            self.found_push = True
            self.found_push_cycle = self.cycle_counter
        elif instruction == 'xor' and op1 == op2:
            self.found_xor = True
            self.found_xor_cycle = self.cycle_counter
        elif instruction == 'stc':
            self.found_stc = True
            self.found_stc_cycle = self.cycle_counter
        elif instruction == 'clc':
```

```
            self.found_clc = True
            self.found_clc_cycle = self.cycle_counter

        if self.found_push and instruction == 'ret' and self.cycle_counter == self.found_push_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_xor and instruction == 'jnz' and self.cycle_counter == self.found_xor_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_stc and (instruction == 'jnc' or instruction == 'jae') and self.cycle_counter == \
                        self.found_stc_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_clc and (instruction == 'jc' or instruction == 'jb') and self.cycle_counter == \
                        self.found_clc_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

NOP encoded data NOP + immediato - expandir detector

4.3. Anti-debug

In this section, we present techniques aimed to detect if a sample is being debugged.

4.3.1. Known Debug APIs Usage

The most direct way to detect debugger usage is to check if a given file imports any debug related function from system APIs. On Windows O.S., for instance, many debug related APIs, such as `IsDebuggerPresent` and `OTHER`, are available on its default libraries.

The natural countermeasure is to check the presence of such function on binary imports section. This approach is implemented in tools like PEframe. Listing 23 shows an excerpt of PEFrame's json file used for pattern matching.

Listing 23. PEframe antidbg.

```
"antidbg": [
            "CheckRemoteDebugger",
            "DebugActiveProcess",
            "FindWindow",
            "GetLastError",
```

```
                    "GetWindowThreadProcessId",
                    "IsDebugged",
                    "IsDebuggerPresent",
                    "IsProcessorFeaturePresent",
                    "NtCreateThreadEx",
                    "NtGlobalFlags",
                    "NtSetInformationThread",
                    "OutputDebugString",
                    "pbIsPresent",
                    "Process32First",
                    "Process32Next",
                    "RaiseException",
                    "TerminateProcess",
                    "ThreadHideFromDebugger",
                    "UnhandledExceptionFilter",
                    "ZwQueryInformation"
        ],

        for lib in pe.DIRECTORY_ENTRY_IMPORT:
                for imp in lib.imports:
                        for antidbg in antidbgs:
                                if antidbg:
```

4.3.2. Debugger Fingerprint

As well as API function imports, we can also check sample's strings in order to find known debugger symbols, which indicates the sample may use such values to check system properties. Tools such as JAMA [Liu] implements such kind of pattern matching checking, as shown on Listing 24.

Listing 24. JAMA antidbg.

```
DEBUGGING_TRICKS = {
        "SICE":"SoftIce detection",
        "REGSYS":"Regmon detection",
        "FILEVXG":"Filemon detection",
        "TWX":"TRW detection",
        "NTFIRE.S":"'DemoVDD By elicz' technique",
        "OLLYDBG":"OllyDbg detection",
```

4.3.3. NtGlobalFlag

The process environment block [Microsoft c] is a system internal structure related to process management. Among its internal data, there is the `NtGlobalFlag`, which holds data related to process heap. When a process is being debugged, specific flags of this field are enabled, which can be used to identify debugger's presence.

Specifically, the flags shown on Table 2 are set. When a process is not being debugged, the typical value of the field is 0 whereas the value changes when a debugger is attached.

Table 2. NtGlobalFlag

Flag	Value
FLG_HEAP_ENABLE_TAIL_CHECK	0x10
FLG_HEAP_ENABLE_FREE_CHECK	0x20
FLG_HEAP_VALIDATE_PARAMETERS	0x40
Total	0x70

The NtGlobalFlags can be accessed directly by using the undocumented function `RtlGetNtGlobalFlags` from `ntdll.dll`, as shown on Listing 25.

Listing 25. NtGlobalFlag API Read.

```
    _RtlGetNtGlobalFlags GetNtGlobalFlags =
    (_RtlGetNtGlobalFlags)(GetProcAddress(GetModuleHandle(_T("
        ntdll.dll")),
    "RtlGetNtGlobalFlags"));
```

This function makes a straightforward PEB reading implementation, as shown on Listing 26[1].

Listing 26. NtGlobalFlag API Read Implementation.

```
ULONG WINAPI RtlGetNtGlobalFlags(void)
{
    return NtCurrentTeb()->Peb->NtGlobalFlag;
}
```

Using an API function, however, can ease the sample's detection, since API imports are shown on PE structure and can also be monitored in runtime. Some authorsm instead, prefer directly accessing PE structure in memory. The PEBs base address is located at the `fs:0x30` offset, and NtGlobalFlags at `0x68`, being directly accessible. Listing 27 shows a possible implementation of this evasion technique.

Listing 27. NtGlobalFlag Evasion.

```
        call    puts
        mov     eax, [fs:0x30]
        mov eax, [eax+0x68]
        mov     eax, 0
        pop     rbp
```

A possible detector for this technique is to check for the PEBs base address loading followed by a comparison on the NtglobalFlags offset. Listing 28 shows the implemented detector.

[1] Wine implementation

Listing 28. NtGlobalFlag Evasion Detector.

```
    def check(self, section, address, instruction, op1, op2):

        if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction in ['cmp', 'cmpxchg', 'mov', 'movsx', 'movzx']:
                if '[' + self.found_op1 + '+0x68]' in op1 or '[' + self.found_op1 + '+0x68]' in op2:
                    self.clear()
                    print "\"PEB NtGlobalFlag\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.3.4. IsDebuggerPresent

Besides the global flags, the PEB struct has also an specific flag which indicates if a process is being debbuged, as can be seen on Listing 29.

Listing 29. PEB structure.

```
        typedef struct _PEB {
                BYTE                            Reserved1[2];
                BYTE                            BeingDebugged;
```

The BeingDebugged flag is set when a debugged is attached to the process or an debug-related API call is made on the process. This way, malware can check the presence of a debugger an thus evade the analysis. This verification can be performed by using native API calls, such as `IsDebuggerPresent` [Microsoft 2016], as shown on Listing 30.

Listing 30. Identificação de *debug*.

```
                if(IsDebuggerPresent())
                        printf("debugged\n");
                else
                        printf("NO DBG\n");
```

This API implementation is a direct read from the PEB data, as can be seen on Listing 31.

Listing 31. Isdbg implementation.

```
                BOOL WINAPI IsDebuggerPresent(void)
                {
                        req->handle = GetCurrentProcess();
```

```
                    ret = req->debugged;
                    return ret;
            }
```

Likewise Ntglobalflag's case, authors avoid using API calls since they can be traced, and opt to implement their own PEBs checker. One way of performing such check is loading the PEB base address at `fs:0x30` offset and thus reading the BeingDebugged flag at `0x2` offset. This verification was implemented and can be seen on Listing 32.

Listing 32. IsDebuggerPresent.

```
mov eax, [fs:0x30]
        mov eax, [eax+0x2]
        mov     eax, 0
        pop     rbp
```

The detector for this technique should check if these two access are performed, as shown on Listing 33.

Listing 33. IsDebuggerPresent Detector.

```
    def check(self, section, address, instruction, op1, op2):

        if instruction in ['mov', 'movsx', 'movzx'] and 'fs:0x30' in op2:
            self.found_op1 = op1
            self.found_keyword = True
            return

        if self.found_keyword:
            if instruction in ['mov', 'movsx', 'movzx']:

                substring = '[' + self.found_op1 + '+0x2]'

                if substring in op1 or substring in op2:
                    self.clear()
                    print "\"IsDebbugerPresent\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.3.5. Hook Detection

Hooking is a technique where execution control flow is deviated to a trampoline function with arbitrary code. This deviation can be used to log action or system subversion, for example. This modification can be done by using system facilities, such as API calls, in the case of `SetWindowsHook` [Microsoft e]. or binary changes, in the case of `detours` [Microsoft a].

A variation of this technique, called inline hooking, consists of patching abinary with a JMP instruction. An evasive sample can check if its binary was patched on memory by checking if a given snippert of code starts with a JMP instruction. Due to this check, a comparison to the`E9` opcode (JMP) can be seen on the generated code, as shown on Listing 34.

Listing 34. Hook Detection.

```
cmp [eax+0xe9], eax
        pop     rbp
```

The `CMP` instruction with this operand (E9) can be used to build an anti-analysis detector, as shown on Listing 35.

Listing 35. Hook Detection.

```
    def check(self, section, address, instruction, op1, op2):

        if instruction == 'cmp' and ('0xe9' in op1.lower() or '0xe9' in op2.lower()):
            print "\"HookDetection\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.3.6. Heap Flags

Likewise global flags, PEB's heaps have also their own flags, as shown on table 3, which can be used as a analysis indicators by evasive malware.

Table 3. heap

Flag	Value
HEAP_GROWABLE	2
HEAP_TAIL_CHECKING_ENABLED	0x20
HEAP_FREE_CHECKING_ENABLED	0x40
HEAP_SKIP_VALIDATION_CHECKS	0x10000000
HEAP_VALIDATE_PARAMETERS_ENABLED	0x40000000

Heap checks can be performed by using the API call to `GetProcessHeap` [Microsoft b] or by implementing their own checks, retrieving the PEB base address at `fs:0x30` and them referencing the default heap at `0x18`, as shown on Listing 36.

Listing 36. Heap Flags.

```
        mov     eax, [fs:0x30]
        mov eax, [eax+0x18]
        mov     eax, 0
        pop     rbp
```

As a detector, we can save the resulting address associated to the PEB query and them search for this values plus the heap offset on the following instructions, as shonw on Listing 37.

Listing 37. Heap Flags.

```
def check(self, section, address, instruction, op1, op2):

        if op1 is not None and 'fs:0x30' in op1:
            self.found_op = op1
            self.found_first = True
            return
        elif op2 is not None and 'fs:0x30' in op2:
            self.found_op = op2
            self.found_first = True
            return

        if self.found_op is not None and ((op1 is not None and
           '[' + self.found_op + '+0x18]' in op1) or (op2 is not
                                                                                None
          and
          '['
          +
          self.found_op
          +
          '+0x18]'
          in
          op2
          )
          )
          :
            self.clear()
            print "\"HeapFlags\" Detected! Section: <%s> Address
                : 0x%s" % (section, address)
```

4.3.7. Hardware Breakpoint Detection

Modern processor provide hardware-assisted debugging facilities, such as hardware breakpoints, which allow instruction addresses being stored on special register and stop the execution when one of such address is fetched by the processor.

Listing 38 shows an excerpt of code used to manipulate debugging data presented when a hardware debugger is attached. The `0xc` offset represents the debugger context struct whereas the `0x4` an access to the debug register number 0.

Listing 38. Hardware dbg detect.

```
        mov     [fs:0x0], rsp
        mov     rax, [rsp+0xc]
        cmp     rbx, [rax+0x4]
```

An detector can be implemented by checking if such kind of manipulation is performed on a given function. On the implementation provided on Listing 39, we check for `0x4,0x8,0xc,0x10` offsets, representing debug registers 0 to 3.

Listing 39. Hardware dbg antidetect.

```
if instruction in ['mov','movsx','movzx'] and 'fs:0x0' in op1 and 'rsp' in op2:
            self.seh = True

        elif self.seh==True and instruction in ['mov','movsx','movzx'] and 'rsp+0xc' in op2:
            self.found_first=True
            self.found_op=op1

        elif self.found_first==True and instruction in ['mov','movsx','movzx','cmp','cmpxchg'] and self.found_op in op2 and ('0x4' in op2 or '0x8' in op2 or '0xc' in op2 or '0x10' in op2):
            self.found_second=True

        if self.found_second==True:
            self.clear()
            print "\"HardwareBreakpoint\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.3.8. SS register

When running on a debugger, it replaces the first byte of a given instruction by a trap flag. In order to be more transparent, many debugger solutions try to clean this trap flag. However, when the SS register is loaded through a `POP` instruction, the interruption is disabled until the end of the next instruction, avoiding invalid stack issues. This way, an evasive sample could insert a check right after popping the SS.

The code presented on Listing 40 illustrates an implementation for this technique.

Listing 40. ss register

```
        pop     ss
        pushf
```

The detection technique implemented is to check the usage of the SS register immediately after popping. The implementation of this detector is shown on Listing 41.

Listing 41. ss register detection

```
         if instruction in ['mov', 'movsx', 'movzx']:
            if 'ss' in op1:
                self.found_ss = True
        elif instruction== 'pop':
            if 'ss' in op1:
                self.found_ss = True
        elif 'pushf' in instruction:
            self.found_flag=True
```

4.3.9. Software breakpoint

Unlike hardware breakpoints, which are register based and thus limited in numer, Software breakpoints are software constructions and thus unlimited in practice. In order to identify the distinct points where the execution will be stopped, the debugger changes the first byte of the instruction to the `0xCC` byte, which represents the `INT3` instruction.

An evasive sample can scan its own memory and check for the 0xCC byte, detecting the debugger, as shown on Listing 42.

Listing 42. soft debugger

```
        cmp     rax, 0xCC
```

As a detector for this technique, we can check code for comparisons to the 0xCC byte, as shown on Listing 43.

Listing 43. soft debugger detection

```
        if 'cmp' in instruction:
            if '0xcc' in op1 or '0xcc' in op2:
                self.found_cmp = True

        if self.found_cmp==True:
            self.clear()
            print "\"Software Breakpoint\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

4.3.10. SizeOfImage

A trick able to defeat debugger consists on changing the image size field, so a debugger becomes unable to parse its content. This technique can be seen on Listing 44.

Listing 44. sizeof

```
        mov     eax, [fs:0x30]
        mov     eax, [eax+0xc]
```

```
        mov      eax , [ eax+0xc ]
        addw     [ eax +20] ,0 x1000
```

As a detector for this technique, we can look for signals of value changing on this field, as shown on Listing 45.

Listing 45. size detect

```
if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction in ['mov', 'movsx', 'movzx']:
                if '[' + self.found_op1 + '+0xc]' in op2:
                    self.found_op2 = op1
                    self.found_keyword2 = True

        if self.found_keyword2:
            if instruction in ['mov', 'movsx', 'movzx']:
                if '[' + self.found_op2 + '+0xc]' in op2:
                    self.found_op3 = op1
                    self.found_keyword3 = True

        if self.found_keyword3:
            if instruction in ['addw','add','sub']:
                if '['+self.found_op3 + '20]':
                    print "\"SizeOfImage\" Detected! Section: <%
                        s> Address: 0x%s" % (section, address)
                    self.clear()
```

A way to defeat such trick is to recompute the image size. This calculation can be performed using the `VirtualQuery` [Microsoft f] api.

4.4. Anti-VM

In this section, we present techniques related to virtual machine detection.

4.4.1. VM Fingerprint

Some solutions left presence indicators in the system, such as known strings. A straightforward way to detect VM is to check the presence of such strings on system properties. VMware, for example, presents a code, shown on Listing 46 which detects the VMware solution by the presence of its strings on BIOS code.

Listing 46. VMware finger.

```
int dmi_check(void)
        char string[10];
```

```
        GET_BIOS_SERIAL(string);

        if (!memcmp(string, ”VMware-”, 7) || !memcmp(string, ”VMW”, 3))
                return 1;                       // DMI contains VMware specific string.
        else
                return 0;
```

The same way, a straightforward evasion detection technique is to verify the presence of such verifiable strings on the suspicious binary. PEframe implements such kind of verification. Listing 47 shows an excerpt of PEframes implementation performing such checks.

Listing 47. PEFrame fingerprint.

```
VM_Str  = {
                ”Virtual Box”:”VBox”,
                ”VMware”:”WMvare”
        }

for string in VM_Str:
        match = re.findall(VM_Str[string], buf, re.IGNORECASE | re.MULTILINE)
        if match:
```

4.4.2. CPUID check

Another fingerprint approach is to make use of the `CPUID` instruction, which fills CPU register with the vendor string. On VM cases, the hypervisor name is supplie. This way, a traditional evasive approach is to compare CPUID results to known VM strings, such as Xen or QEMU. The same way, the natural check for evasive samples is to locate such strings on the binaries, such as implemented by PEFRAME, as shown on Listing 48.

Listing 48. PEFrame CPUID.

```
        VM_Sign = {
                ”Xen”:”XenVMM”,
```

Another detection possibility is to check the 31th returned bit from CPUID instruction, which should return if processor has hypervisor capabilities or not [hexacorn].

As presented by Vmware [VMWare], this verification could be implemented as shown on Listing 49.

Listing 49. Vmware check.

```
int cpuid_check()
{
        unsigned int eax, ebx, ecx, edx;
        char hyper_vendor_id[13];
```

```

        cpuid(0x1, &eax, &ebx, &ecx, &edx);
        if (bit 31 of ecx is set) {
                cpuid(0x40000000, &eax, &ebx, &ecx, &edx);
                memcpy(hyper_vendor_id + 0, &ebx, 4);
                memcpy(hyper_vendor_id + 4, &ecx, 4);
                memcpy(hyper_vendor_id + 8, &edx, 4);
                hyper_vendor_id[12] = '\0';
                if (!strcmp(hyper_vendor_id, "VMwareVMware"))
                        return 1;            // Success - running under VMware
        }
        return 0;
}
```

4.4.3. Invalid Opcodes

Hypervisors used to support special opcodes and parameter values not accepted on physical machines. An evasive sample can try to execute such special instructions in order to verify if the environments answer properly or not. The code from [Bachaalany], reproduced on Listing 50, shows how this technique can be used to detect the VirtualPC hypervisor.

Listing 50. Virtual PC detection.

```
__try
  {
        _asm __emit 0Fh
        _asm __emit 3Fh
        _asm __emit 07h
        _asm __emit 0Bh
}catch{
        //real machine
```

Those bytes can be used as pattern for anti-anti-analysis techniques, as in PE-FRAME, shown on Listing 51.

Listing 51. PEFrame vpc.

```
VM_Sign = {
                "VirtualPc trick":"\x0f\x3f\x07\x0b",
```

4.4.4. System Table Checks

As previously mentioned, virtual machines change tables addresses, such as IDT and GDT, Thus, table relocations can be interpreted as virtual machine identifiers by malware samples. We can detect these kind of checks by verifying the presence of instructions related to table addresses on binaries. The instructions of interest are those which store the table addresses on given memory locations. The `store` meaning is due to the fact that a system address is stored on memory. On the other side, when a new table address is defined, this address is `load`-ed into the system. Listing 52 shows the detector for this technique.

Listing 52. Detecting instructions related to tables addresses checking.

```
if instruction.lower() in ['sidt', 'sldt', 'sgdt', 'str']:
print "\"CPUInstructionsResultsComparison\" Detected! Section:
    <%s> Address: 0x%s" % (section, address)
```

On our environment, table...

This kind of IDT checkage can be found in practice on many samples. It first appear is credited to Joanna Rutkowska, on non-academical literature. Listing 53 shows how this kind of check is usually implemented [Securiteam].

Listing 53. Check.

```
int swallow_redpill () {
        unsigned char m[2+4], rpill[] =
        "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
            *((unsigned*)&rpill[3]) = (unsigned)m;
            ((void(*)())&rpill)();
            return (m[5]>0xd0) ? 1 : 0;
 }
```

The hex-encoded data launches the IDT check instruction. Many detector use these bytes as signatures, such as on PEframe implementation, shown on Listing 54.

Listing 54. PEFRAME sign.

```
VM_Sign = {
        "Red Pill":"\x0f\x01\x0d\x00\x00\x00\x00
            \xc3",
```

4.4.5. VMware Hypercall Detection

Similarly to the O.S syscalls, hypervisors have their own ways to be invoked by the running systems. This way is usually named hypercall. On Vmware-based systems, one hypercall is made by generating an I/O operation on a specific port (Vx), present only on these virtualized systems. An evasive sample can try to write on this port and, if sucessfull, identify it is a VMware-powered system. A detector for this technique consists on detecting the `IN` instruction on the VX port. Listing 55 presents the detector implementation.

Listing 55. Vmware.

```
def check(self, section, address, instruction, op1, op2):

    if instruction == 'in' and ('vx' in op1.lower() or 'vx'
        in op2.lower()):
```

```
            print "\"VMWareINInstruction\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

In practice [Laboratory], the code to check the VX port looks like the one presented on Listing 56

Listing 56. VMx check.

```
        __asm
        {
                mov eax, 0x564D5868 ; ascii: VMXh
                mov edx, 0x5658 ; ascii: VX (port)
                in eax, dx ; input from Port
                cmp ebx, 0x564D5868 ; ascii: VMXh
                setz ecx ; if successful -> flag = 0
                mov vm_flag, ecx
        }
```

Detectors like PEFrame used to consider the VMx instruction as signatures for anti-vm techniques, as shown on Listing 57.

Listing 57. PEFRAME vmx.

```
        VM_Sign = {
                "VMware trick":"VMXh",
```

A Similar approach is taken on torpig detection [MNIN.org]. Its IDT check is used as signature on PEFRAME(58) and even SNORT (59).

Listing 58. PEFRAME torpig.

```
VM_Sign = {
                "Torpig VMM Trick": "\xE8\xED\xFF\xFF\xFF\x25\x00\x00\x00\xFF\x33\xC9\x3D\x00\x00\x00\x80\x0F\x95\xC1\x8B\xC1\xC3",
                "Torpig (UPX) VMM Trick": "\x51\x51\x0F\x01\x27\x00\xC1\xFB\xB5\xD5\x35\x02\xE2\xC3\xD1\x66\x25\x32\xBD\x83\x7F\xB7\x4E\x3D\x06\x80\x0F\x95\xC1\x8B\xC1\xC3"
                }
```

Listing 59. snort torpig.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"BLEEDING-EDGE VMM
Detecting Torpig/Anserin/Sinowal Trojan"; flow:to_client, established;
content:"|51 51 0F 01 4C 24 00 8B 44 24 02 59 59 C3 E8 ED FF FF FF 25
00 00 00 FF 33 C9 3D 00 00 00 80 0F 95 C1 8B C1 C3|"; classtype:trojan-
activity; sid:20060810; rev:1;)
```

5. Evaluation Proposal

Given the presented detectors, we have implemented them by using a python script , available at URL. It is based on objdump disassembly, being able to provide information about the presence of each technique in a given binary, the number of occurrences per binary, and section.

Unlike Branco et al. approach, which considered `RET` instruction as a code delimiter, we have implemented a variable-size window in order to evaluated if techniques make use of multiple-block constructions.

6. Case Study

In this section, we present the results of applying the developed detector against multiple datasets. Initially, we observed the general detection rate on a diverse dataset. We can observe, on it, if samples are making use of distinct binary section and inter-block constructions. Secondly, We can compare the incidence of evasive techniques against distinct datasets. In addition, we can verify the incidence of these techniques on goodware binaries.

6.1. General Detection

Total of 71038 samples from malshare database....

peframe results

Table 4. peframe inter

Vmcheck	45726	64,33%
Bochs	25563	35,94%
QEMU	25563	35,94%
VirtualBox	1350	1,9%
VMWare	700	0,98%
VirtualPC	40	0,05%

our solution

Table 5. tricks inter

PushPopMath	67889	95,57%
PushRet	67704	95,31%
GarbageBytes	67432	94,92%
ProgramControlFlowChange	67411	94,89%
Software Breakpoint	60732	85,49%
CPUInstructionsResultsComparison	59937	84,37%
HookDetection	57654	81,16%
FakeConditionalJumps	47956	67,51%
SS Register	31326	44,10%
NOPSequence	19070	26,84%
LDR	534	0,75%
PEB NtGlobalFlag	492	0,69%
IsDebbugerPresent	474	0,67%
SizeOfImage	321	0,45%

comparar pyew, etc

PEframe fez TOTAL 5247 7,39% nosso fez total 69091 97,26%

ver se há antivm em seções que nao .text

um total de 253258 entradas

Table 6. section dist

.rsrc	60833	24,02%
.rdata	51684	20,41%
.reloc	43727	17,27%
.data	37431	14,78%
.text	33897	13,38%
.None	15777	6,23%
.idata	2190	0,86%
.itext	1466	0,58%
.complua	752	0,30%
.aspack	425	0,17%
.bindat	424	0,17%

Table 7. .text

Technique	# occurences	% total	% text
PushRet	29849	42,02%	88,06%
GarbageBytes	29481	41,50%	86,97%
ProgramControlFlowChange	29439	41,44%	86,85%
PushPopMath	19996	28,15%	58,99%
Software Breakpoint	18894	26,60%	55,74%
HookDetection	9005	12,68%	26,57%
NOPSequence	7836	11,03%	23,12%
FakeConditionalJumps	6281	8,84%	18,53%
CPUInstructionsResultsComparison	6031	8,49%	17,79%
SS Register	2260	3,18%	6,67%
LDR	136	0,19%	0,40%
SizeOfImage	85	0,12%	0,25%
PEB NtGlobalFlag	76	0,11%	0,22%
IsDebbugerPresent	63	0,09%	0,19%

fazer conta dos packer: 63064 tem packer (88,77%) seção por packer

Table 8. packer, section

Nullsoft PiMP Stub ->SFX'	<.rsrc>	7,24%
Microsoft Visual C++ 8'	<.rsrc>	6,81%
Microsoft Visual C++ 8'	<.reloc>	6,42%
Nullsoft PiMP Stub ->SFX'	<.rdata>	2.07%
Nullsoft PiMP Stub ->SFX'	<.text>	1,73%
Microsoft Visual C++ 8'	<.text>	1,67%
UPX ->www.upx.sourceforge.net'	<.reloc>	1,04%
UPX ->www.upx.sourceforge.net'	<.rsrc>	0,82%

tecnica por packer

Table 9. packer technique

Visual C++	
PushRet	5,86
PushPopMath	5,04
GarbageBytes	2,68
(u'Nullsoft PiMP Stub ->SFX'	
PushRet	5,62
PushPopMath	4,74
GarbageBytes	2,57
UPX	
PushRet	0,85
GarbageBytes	0,37

ver se variar a janela resolve (pula ret)

Skip ret, split section

Listing 60 show fakeret code and Listing 61 show splited disasm.

Listing 60. fakeret

```
xor eax, eax
jmp     fakeret
call    puts
ret
fakeret:
jnz     main
```

Listing 61. fakeret

```
  27:   31 c0                   xor    %eax,%eax
  28:   eb 26                   jmp    31 <fakeret>
  29:   e8 00 00 00 00          callq  2e <main+0x2e>
  30:   c3                      retq

0000000000000031 <fakeret>:
  31:   0f 85 00 00 00 00       jne    37 <fakeret+0x6>
  37:   b8 00 00 00 00          mov    $0x0,%eax
```

Table 10. changing window

Window	Abs	Base
10	543066	100,00%
20	543090	100,00%
30	543561	100,09%
40	544072	100,19%
50	544573	100,28%
60	545366	100,42%
70	546208	100,58%
80	546208	100,58%
90	546585	100,65%
100	546641	100,66%

contrib: show construction in practice

6.2. BR x World

primeira impressao surpreende

Table 11. peframe nacional

outro estudo(BR)			Este(Inter)	
VMCheck.dll	2,729	(10.48%)	2556	3,60%
VMware	850	(3.26%)	2556	3,60%
VirtualBox	306	(1.17%)	135	0,19%
Bochs	340	(1.31%)	4572	6,44%
VirtualPC	17	(0.07%)	70	0,10%

quando aplica o outro

BR 15541 26043 59,68%

INTER: 69061 71038 97,21%

cenario se revela com tecnicas mais avançadas

Table 12. our tricks nacional x inter

		nacional	inter
PushPopMath	15484	59,46%	95,57
PushRet	15457	59,35%	95,31
GarbageBytes	15398	59,13%	94,92
ProgramControlFlowChange	15390	59,09%	94,89
CPUInstructionsResultsComparison”	15047	57,78%	84,37
Software Breakpoint	15039	57,75%	85,49
HookDetection	14880	57,14%	81,16
FakeConditionalJumps	13266	50,94%	67,51
SS Register	12871	49,42%	44,1
NOPSequence	8158	31,33%	26,84
PEB NtGlobalFlag	634	2,43%	0,69
LDR	599	2,30%	0,75
IsDebbugerPresent	486	1,87%	0,67
SizeOfImage	203	0,78%	0,45

conclusao é que BR tem mais dos mais simples

6.3. Malware x Goodware

2 partes: divide pq nao sabe se tem adware no cnet

1. 2870 Win/System – 2422 DLL and 448 PE

161/2870 (5,6%)

Table 13. benignos win

Tecnica	% total bin	% total detec
PushPopMath	13,11%	80,00%
PushRet	12,59%	79,86%
GarbageBytes	11,71%	79,10%
ProgramControlFlowChange	11,71%	79,10%
”CPUInstructionsResultsComparison”	11,45%	80,92%
Software Breakpoint	9,70%	80,18%
HardwareBreakpoint	8,83%	86,14%
NOPSequence	7,08%	85,19%
HookDetection	6,99%	76,25%
FakeConditionalJumps	4,02%	76,09%
SS Register	2,80%	75,00%

razoavel ter mta dll, vide a arquitetura do windows

2. softonic

2239

1898/2239 tem algo

Table 14. softnoic tricks

PushPopMath	1881	84,01%
PushRet	1809	80,79%
GarbageBytes	1737	77,58%
ProgramControlFlowChange	1736	77,53%
CPUInstructionsResultsComparison”	1724	77,00%
HookDetection	1288	57,53%
Software Breakpoint	637	28,45%
FakeConditionalJumps	599	26,75%
SS Register	369	16,48%
NOPSequence	329	14,69%
IsDebbugerPresent	31	1,38%
LDR	30	1,34%
PEB NtGlobalFlag	30	1,34%
SizeOfImage	21	0,94%
HardwareBreakpoint	1	0,04%

dificil explicar, deu mais do que eu esperava

seções

280 : ¡.itext¿ 752 : ¡.text¿

6.4. Compiler-based evasion

ROP itself malware [Poulios et al.].

Ropinjector [Poulios]

SSexy [Bremer]

Movfuscator [domas]

MMU [Barngert]

Table 15. compilation

ShellCode	Unarmored	ROPinjector	Xor	SSEXY
1[2]	4/57	0/57		
2[3]	15/58	0/57		
3[4]	9/57	0/54		
4[5]	7/58	0/54		
5[6]	9/53	0/53		

6.5. Unaligned Evasion Techniques

opcode pattern matching - unaligned - using YARA rules – predefined rules

Listing 62.

Listing 62. YARA

```
rule CPU_Detector : CPU
{
        meta:
        description = "CPU Instruction Detector"

        strings:
                $str={0F 00}
                $sidt={0F 01}

        condition:
                $str or $sidt
}
```

Listing 63.

Listing 63. YARA

```
rule FakeJump_Detector : FakeJump
{
        meta:
        description = "FakeJump Detector"

        strings:
                $seq={31 ?? 0F}

        condition:
                $seq
}
```

wildcards ignore instr. immediate

subset of 287 samples, 182 CPU tricks (295 occorences) and 63 FakeJMP (685 occorences)

Yara detected all 287 as having tricks, 287 CPU tricks (2693 possible ocurences) and 203 FakeJMP (5578 possible ocurrences)

not sure, but a tip!

6.6. Packed x Unpacked

tirar os UPX que der

1392 UPX - extrairam em 850

1325 detectaram algo – 848 detectaram algo

Table 16. com e sem uPX

Packed				Unpacked	
PushPopMath	1310	94,11%		PushPopMath	842
PushRet	1308	93,97%		GarbageBytes	833
GarbageBytes	1305	93,75%		PushRet	827
ProgramControlFlowChange	1305	93,75%		ProgramControlFlowChange	825
Software Breakpoint	1252	89,94%		CPUInstructionsResultsComparison”	791
CPUInstructionsResultsComparison”	1241	89,15%		HookDetection	736
HookDetection	1226	88,07%		Software Breakpoint	723
FakeConditionalJumps	1144	82,18%		FakeConditionalJumps	719
SS Register	999	71,77%		SS Register	672
NOPSequence	751	53,95%		NOPSequence	207
PEB NtGlobalFlag	31	2,23%		IsDebbugerPresent	9
LDR	29	2,08%		LDR	9
IsDebbugerPresent	27	1,94%		PEB NtGlobalFlag	9
SizeOfImage	9	0,65%		SizeOfImage	9

tbm nao mudou mto, mas nao é erro, detector de cpu é simples, por exemplo. por que tanta verificação de cpu no UPX ?

7. Rule-based anti-dbg

Como no artigo [Lee et al. 2013]

8. Evading AV

Bypass AV [Nasi] usar essas tecnicas + payload metasploit

Table 17. AV Evasion

Shellcode	SC1		SC2		SC3	
Technique	Without Trick	With Trick	Without Trick	With Trick	Without Trick	With Trick
Fakejmp	10/58	6/57	20/58	17/58	15/58	10/57
PushRet		7/57		17/58		10/58
NOP		6/57		17/57		10/58

9. Discussion

9.1. Limitations

limitations - stealth import n funciona igual em win mais novo, tem que comparar string na mao - objdump suscetivel mtos desses problemas - packer, etc size of image - unpack ms article

moving para x64, as shown on Table 17.

Table 18. x32 to x64 mapping

Value	X32	x64
PEB	fs:0x30	fs:0x60
NtGlobalFlag	0x68	0xbc
_HEAP	0x40	0x70

9.2. Future Work

expansion, bla bla bla

10. Conclusion

In this work, we have studied anti-analysis techniques, their effect on malware analysis and theorethical limitation. We also developed static detectors able to identify known evasive constructions on binaries. We have tested these detectors against multiple datasets and observed that...

References

Bachaalany, E. Detect if your program is running inside a virtual machine. `https://www.codeproject.com/articles/9823/detect-if-your-program-is-running-inside-a-virtual`.

Barngert, J. Trapcc. `https://github.com/jbangert/trapcc`.

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies. `https://github.com/rrbranco/blackhat2012/blob/master/blackhat2012-paper.pdf`.

Bremer, J. Ssexy. `https://github.com/jbremer/ssexy.git`.

Calvet, J., Fernandez, J. M., and Marion, J.-Y. (2012). Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 169–182, New York, NY, USA. ACM.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA. ACM.

domas. Movfuscator. `https://github.com/xoreaxeaxeax/movfuscator`.

Eliam, E. (2005). *Reverse: Secrets of Reverse Engineering*.

Ferrand, O. (2015). How to detect the cuckoo sandbox and to strengthen it? *Journal of Computer Virology and Hacking Techniques*, 11(1):51–58.

Ferrie, P. Anti-unpacker tricks. `http://pferrie.tripod.com/papers/unpackers.pdf`.

Ferrie, P. Attacks on virtual machine emulators. `https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

hexacorn. Protecting vmware from cpuid hypervisor detection. `http://www.hexacorn.com/blog/2014/08/25/protecting-vmware-from-cpuid-hypervisor-detection/`.

Hexacorn. Rdtscp - a recooked antire trick. `http://www.hexacorn.com/blog/2014/06/15/rdtscp-a-recooked-antire-trick/`.

Kruegel, C., Kirda, E., and Moser, A. (2007). Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC) 2007*.

Laboratory, B. Detecting vmware. `https://brundlelab.wordpress.com/2012/10/21/detecting-vmware/`.

Lee, J., Kang, B., and Im, E. G. (2013). Rule-based anti-anti-debugging system. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, pages 353–354, New York, NY, USA. ACM.

Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. (2011). Detecting environment-sensitive malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 338–357, Berlin, Heidelberg. Springer-Verlag.

Liu, Q. Just another malware analyzer. `https://github.com/lqhl/just-another-malware-analyzer`.

Lyashko, A. Stealth import of windows api. `http://syprog.blogspot.com.br/2011/10/stealth-import-of-windows-api.html`.

Microsoft. Download detours express. `https://www.microsoft.com/en-us/download/details.aspx?id=52586`.

Microsoft. Getprocessheap function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366569%28v=vs.85%29.aspx`.

Microsoft. Peb structure. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813706%28v=vs.85%29.aspx`.

Microsoft. Peb_ldr_data structure. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813708%28v=vs.85%29.aspx`.

Microsoft. Setwindowshookex function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990%28v=vs.85%29.aspx`.

Microsoft. Virtualquery function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366902%28v=vs.85%29.aspx`.

Microsoft (2016). Isdebuggerpresent. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680345%28v=vs.85%29.aspx.

MNIN.org. The torpig/anserin/sinowal family of trojans detect virtual machine information and abort infection when present. https://www.mnin.org/write/2006_torpigsigs.pdf.

Nasi, E. Bypass antivirus dynamic analysis. http://packetstorm.foofus.com/papers/virus/BypassAVDynamics.pdf.

Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., and Nguyen, H. D. (2009). Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 441–450.

Oleg, K. Anti-debug protection techniques: Implementation and neutralization. https://www.codeproject.com/articles/1090943/anti-debug-protection-techniques-implementation-an.

Pafish. Pafish. https://github.com/a0rtega/pafish.

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 2–2, Berkeley, CA, USA. USENIX Association.

Peframe. Peframe. https://github.com/guelfoweb/peframe.

Pék, G., Bencsáth, B., and Buttyán, L. (2011). nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA. ACM.

Poulios, G. Ropinjector. https://github.com/gpoulios/ROPInjector.

Poulios, G., Ntantogian, C., and Xenakis, C. Ropinjector: Using return oriented programming for polymorphism and antivirus evasion. https://www.blackhat.com/docs/us-15/materials/us-15-Xenakis-ROPInjector-Using-Return-Oriented-Programming-For-Polym pdf.

Pyew. Pyew. https://github.com/joxeankoret/pyew.

Saleh, M., Ratazzi, E. P., and Xu, S. (2014). Instructions-based detection of sophisticated obfuscation and packing. In *2014 IEEE Military Communications Conference*, pages 1–6.

Schwarz, B., Debray, S., and Andrews, G. (2002). Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 45–, Washington, DC, USA. IEEE Computer Society.

Securiteam. Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.securiteam.com/securityreviews/6Z00H20BQS.html.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

Singh, A. Not so fast my friend - using inverted timing attacks to bypass dynamic analysis. http://labs.lastline.com/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic

Smith, A. J., Mills, R. F., Bryant, A. R., Peterson, G. L., and Grimaila, M. R. (2014). Redir: Automated static detection of obfuscated anti-debugging techniques. In *2014 International Conference on Collaboration Technologies and Systems (CTS)*, pages 173–180.

Vasudevan, A. and Yerraballi, R. (2006). Cobra: fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–279.

VMWare. Mechanisms to determine if software is running in a vmware virtual machine (1009458). https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458.

Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., and Vasudevan, A. (2012). Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 189–198, New York, NY, USA. ACM.

Xiao, X., s. Zhang, X., and d. Li, X. (2010). New approach to path explosion problem of symbolic execution. In *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, pages 301–304.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., and Rossow, C. (2016). *SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion*, pages 165–187. Springer International Publishing, Cham.