Appendix to the "Analysis, Anti-Analysis, Anti-Anti-Analysis: An Overview of the Evasive Malware Scenario" paper, submited for publication to XVII SBSEG

A. Appendix

We added to this appendix additional information to support our findings on anti-analysis-armored malware. This text is organized as follows: section A.1 presents examples of evasion found on real samples; section...

A.1. Real Malware Evasion

Some samples stealthly quit their operations when an anti-analysis trick is successful on identifying an analysis environment. However, some of them opt to display some information. We present below (Figure 10, 11, and 12) some examples we found when running real samples we have collected on the wild.

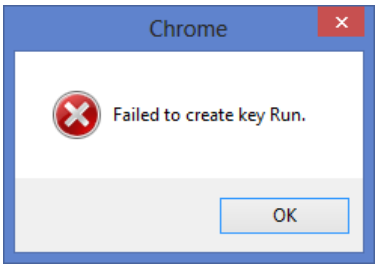

Figure 10. Real malware claiming a registry problem when an anti-analysis trick succeded.

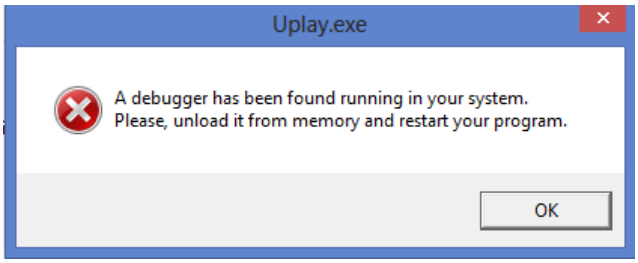

Figure 11. Commercial solution armored with anti-debug technique.

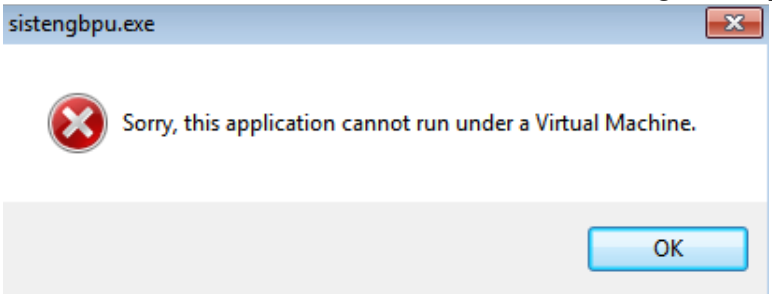

Figure 12. Real malware impersonating a secure solution which cannon run under an hypervisor.

B. Techniques

This section details how the techniques and their detectors are implemented.

B.1. Anti-disassembly

How disassembly, anti-disassembly, and anti-disassembly detectors are implemented.

B.1.1. Disassembly techniques used by distinct solutions

As presented on Section B.1.1, disassembly can be performed on two ways: `Linear Sweep` or `Recursive Traversal`. The disassembler operation mode affects the obtained results. In order to make the scenario clearer, we summarized, in the Table 7, the disassembly approach used by distinct tools.

Table 7. Disassembly technique used by distinct solutions (Based on [Eliam 2005]).

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

B.1.2. Tricks

This section show how the anti-disassembly tricks and their detectors are implemented.

B.1.3. 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 pop-ed to a register. This register is then used on further computations, as shown on Listing 1.

Listing 1. PUSH POP MATH trick.

```
1        push    0x1234
2        pop     rax
3        xor     rax, 0xFFFF
```

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

Listing 2. PUSH POP MATH trick detection.

```
1  if 'push' in instruction and not op1 in ['ax','bx','cx','dx']:
2              self.found_push=True
3          elif 'pop' in instruction and self.found_pop==False:
4              self.found_pop=True
5              self.found_op=op1
6          elif self.found_pop==True and instruction in ['and', 'or
               ','xor']:
7              if self.found_op in op1 or self.found_op in op2:
```

```
8                   self.found_comp=True
9
10          if self.found_comp==True:
11              self.clear()
12              print "\"PushPopMath\" Detected! Section: <%s>
                    Address: 0x%s" % (section, address)
```

B.1.4. 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 the ordinary `CALL` instruction by a sequence of `PUSH` and `RET`, on which an address is inserted into the stack and thus the function returns to it. This sequence is shown on Listing 3.

Listing 3. PUSH RET trick.

```
1           push    0x12345678
2           ret
```

The presence of a sequence of `PUSH` and `RET` can be used to detect this trick usage, as shown on Listing 4.

Listing 4. PUSH RET trick detection.

```
1           def check(self, section, address, instruction, op1, op2)
                :
2
3           if 'push' in instruction:
4               self.found_push=True
5           elif self.found_push==True and 'ret' in instruction:
6               self.found_ret=True
7           else:
8               self.found_push=False
9
10          if self.found_ret:
11              self.clear()
12              print "\"PushRet\" Detected! Section: <%s> Address:
                    0x%s" % (section, address)
```

B.2. LDR Address resolving

The LDR is a PEB internal structure which contains information about loaded modules [Microsoft 2017d], as shown on Listing 5.

Listing 5. LDR struct definition.

```
1   typedef struct _PEB_LDR_DATA {
2           BYTE        Reserved1[8];
3           PVOID       Reserved2[3];
4           LIST_ENTRY  InMemoryOrderModuleList;
5       } PEB_LDR_DATA, *PPEB_LDR_DATA;
6
7   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;
```

When looking to the PEB structure, the LDR information can be found at the `0x0c` offset, as shown on Listing 6.

Listing 6. PEB's LDR entries.

```
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. Direct LDR handling.

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

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

Listing 8. Detecting LDR direct access.

```
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:
```

```
12                          self.clear()
13                          print "\"LDR\" Detected! Section: <%s>
                               Address: 0x%s" % (section, address)
```

B.2.1. Stealth Windows API Import

The Windows API is the basic toolchain for development tasks on this system, being provided by system libraries (DLLs) to be imported by the programs. However, such imports can reveal many information about a program behavior. Thus, directly using system API increases samples' stealthness.

One way to implement such stealth function imports is to rely on `ntdll` and `kernel32` libraries which are automatically mapped into processes, wven without any explicit import. Those libraries are accessible through a walk over the process memory [Lyashko 2011].

This trick starts by retrieving handlers through the SEH chain offset (`0x0`), as shown on Listing 9. Secondly, by iterating over it, until the end, to retrieve the handler at the `0x4` offset, as also shown on Listing 9.

Listing 9. SEH base address and library handler retrievals.

```
1           mov eax,[fs:0]
2           .search_default_handler:
3           cmp dword[eax],0xFFFFFFFF
4           jz .found_default_handler
5           ;go to the previous handler
6           mov eax,[eax]
7           jmp .search_default_handler
8           mov eax,[eax+4]
9           and eax,0xFFFF0000
```

Given such memory location, it is required to look for the MZ signature over the pages, as shown on Listing 10.

Listing 10. Looking for the PE on memory pages.

```
1  .look_for_mz:
2     cmp word [eax],'MZ'
3     jz .got_mz
4     sub eax,0x10000
5     jmp .look_for_mz
```

At this point, a handle for an unknown library was retrieved. In order to identify it, the trick starts with a check of the `0x3c` offset, which contains an offset for a PE string signature followed by `COFF` data, as shown on Listing 11.

Listing 11. Finding PE signature and COFF data.

```
1     mov bx,[eax+0x3C]
2     movzx ebx,bx
3     add eax,ebx
4     mov bx,'PE'
5     movzx ebx,bx
6     cmp [eax],ebx
7     jz .found_pe
```

Once the register holds the COFF header, the trick 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. Getting image data and handling its RVA.

```
1    add eax,0x78
2    mov eax,[eax]
3    add eax,[image_base_address]
```

By accessing the 0xC offset, it gets the NAME RVA, a string containing the library address, and so discovers whether it is kernel32 or ntdll, as shown on Listing 13.

Listing 13. Accesing name rva.

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

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

Listing 14. Stealth import trick detector.

```
1    def check(self, section, address, instruction, op1, op2):
2
3        if self.found_seh==False and instruction in ['mov', '
            movsx', 'movzx']:
4            if 'fs:0x0' in op2:
5                self.found_op1 = op1
6                self.found_seh = True
7
8        elif self.found_seh==True and self.found_handler==False
            and instruction in ['mov', 'movsx', 'movzx']:
9
10           if 'fs:0x30' in op2:
11               self.found_seh=False
12           elif self.found_op1 + '+0x4' in op2:
13               self.found_handler=True
14               self.found_op2 = op1
15
16       elif self.found_handler==True and instruction in ['cmp
            ','cmpxchg']:
17           if self.found_op2 in op1:
18               self.found_cmp=True
19
20       elif self.found_cmp==True and instruction in ['mov','
            movsx','movzx']:
21           if self.found_op2+'+0x3c' in op2:
22               self.found_pe=True
23
24       elif self.found_pe==True and instruction in ['and','or
            ','xor','add','sub','cmp']:
25           if '0x78' in op1 or '0x78' in op2:
```

```
26                  self.found_img=True
27
28          if self.found_img:
29              self.clear()
30              print "\"StealthImport\" Detected! Section: <%s>
                    Address: 0x%s" % (section, address)
```

B.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 analysis procedures harder and to evade pattern matching detectors. A commmon dead code construction is a NOP sequence, as shown on Listing 15.

Listing 15. NOP sequence trick.

```
1 mov     eax, 0
2 nop
3 nop
4 nop
5 nop
6 nop
7 pop     rbp
```

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

Listing 16. NOP sequence trick detection.

```
1 def check(self, section, address, instruction, op1, op2):
2         if instruction == 'nop':
3             self.counter += 1
4             if self.counter is 5:
5                 self.counter = 0
6                 print "\"NOPSequence\" Detected! Section: <%s>
                      Address: 0x%s" % (section, address)
```

B.2.3. Fake Conditional

Many solutions try to follow control flow in order to apply their detectors. Making conditional control flow harder is a powerful anti-analysis technique, 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 registers will always result on zero, thus triggering the zero-conditioned jump.

Listing 17. Fake Conditional trick.

```
1 xor eax, eax
2 jnz main
```

Detectors for this technique rely on detecting `XOR` instructions followed by these kind of construction, such as `JMP`, `STC` or `CLC`, as shown on Listing 18.

Listing 18. Fake Conditional trick detector.

```
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)
```

B.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 tool. A common replacement is to pushing a value in to stack and then launching a `RET` instruction. This construction can be seen on Listing 19.

Listing 19. Control Flow trick.

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

A detector for this technique is to match a sequence of `PUSH` and `RET`, as shown on Listing 20.

Listing 20. Control Flow trick detection.

```
    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)
```

B.2.5. Garbage Bytes

A way to hide data inside binaries is to add the data right after an unconditional `JMP`, since it will be unreachable as code, as shown on Figure 13.

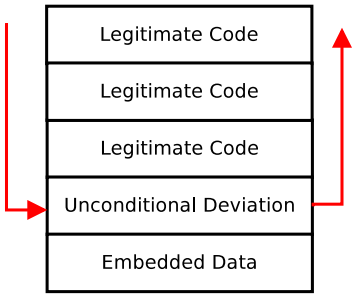

Figure 13. Binary data as a Dead Code.

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-deviation-based anti-analysis techniques, such as `indirect jump`. An implementation of this technique can be seen on Listing 21.

Listing 21. Garbage Bytes trick.

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

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

Listing 22. Garbage Bytes tricks detection.

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

        self.cycle_counter += 1
```

```
4
5           if instruction == 'push':
6               self.found_push = True
7               self.found_push_cycle = self.cycle_counter
8           elif instruction == 'xor' and op1 == op2:
9               self.found_xor = True
10              self.found_xor_cycle = self.cycle_counter
11          elif instruction == 'stc':
12              self.found_stc = True
13              self.found_stc_cycle = self.cycle_counter
14          elif instruction == 'clc':
15              self.found_clc = True
16              self.found_clc_cycle = self.cycle_counter
17
18          if self.found_push and instruction == 'ret' and self.
               cycle_counter == self.found_push_cycle + 1:
19              self.clear()
20              print "\"GarbageBytes\" Detected! Section: <%s>
                   Address: 0x%s" % (section, address)
21          elif self.found_xor and instruction == 'jnz' and self.
               cycle_counter == self.found_xor_cycle + 1:
22              self.clear()
23              print "\"GarbageBytes\" Detected! Section: <%s>
                   Address: 0x%s" % (section, address)
24          elif self.found_stc and (instruction == 'jnc' or
               instruction == 'jae') and self.cycle_counter == \
25                              self.found_stc_cycle + 1:
26              self.clear()
27              print "\"GarbageBytes\" Detected! Section: <%s>
                   Address: 0x%s" % (section, address)
28          elif self.found_clc and (instruction == 'jc' or
               instruction == 'jb') and self.cycle_counter == \
29                              self.found_clc_cycle + 1:
30              self.clear()
31              print "\"GarbageBytes\" Detected! Section: <%s>
                   Address: 0x%s" % (section, address)
```

B.3. Anti-debug

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

B.3.1. Known Debug APIs Usage

The most direct way to detect a debuggers presence is to rely on debug-related function provided by the O.S. API. On Windows O.S., for instance, many debug related APIs, such as `IsDebuggerPresent`, are available on its default libraries.

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

Listing 23. PEframe's anti-debug detection.

```
"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:
```

B.3.2. Debugger Fingerprint

As well as API function imports, one can also checks 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 2012] implements such kind of pattern matching checking, as shown on Listing 24.

Listing 24. JAMA's anti-debug fingerprint.

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

B.3.3. NtGlobalFlag

The Process Environment Block (PEB) [Microsoft 2017c] is a system internal structure related to process management. Among its internal data, there is the `NtGlobalFlag`, which holds data related to the process heap. When a process is being debugged, specific flags of this field are enabled, thus allowing the debugger's presence check.

Specifically, the flags shown in the Table 8 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 8. NtGlobalFlag's heap flags.

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

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

Listing 25. Undocumented NtGlobalFlag API Usage.

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

This function makes a straightforward PEB reading implementation, as shown on Listing 26[11].

Listing 26. Wine's implementation of NtGlobalFlag API.

```
1 ULONG WINAPI RtlGetNtGlobalFlags(void)
2 {
3     return NtCurrentTeb()->Peb->NtGlobalFlag;
4 }
```

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 authors, instead, prefer to access the PE structure directly in memory. The PEB 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 trick.

```
1          call    puts
2          mov     eax, [fs:0x30]
3          mov eax, [eax+0x68]
4          mov     eax, 0
5          pop     rbp
```

A possible detector for this technique is to check for the PEB base address's load followed by a comparison on the `NtglobalFlags`'s offset. Listing 28 shows the implemented detector.

[11] Wine implementation

Listing 28. NtGlobalFlag trick 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)
```

B.3.4. IsDebuggerPresent

Besides the global flags, the PEB struct has also an specific flag which indicates whether 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 debugger is attached to the process or when an debug-related API call is made on the process. This way, malware samples can check the presence of a debugger, and 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. API-based debug detection.

```
                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. Wine's IsDebuggerPresent implementation.

```
BOOL WINAPI IsDebuggerPresent(void)
{
        req->handle = GetCurrentProcess();
        ret = req->debugged;
        return ret;
}
```

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

Listing 32. IsDebuggerPresent trick.

```
1 mov eax, [fs:0x30]
2         mov eax, [eax+0x2]
3         mov      eax, 0
4         pop      rbp
```

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

Listing 33. IsDebuggerPresent trick detector.

```
1    def check(self, section, address, instruction, op1, op2):
2
3        if instruction in ['mov', 'movsx', 'movzx'] and 'fs:0x30
             ' in op2:
4            self.found_op1 = op1
5            self.found_keyword = True
6            return
7
8        if self.found_keyword:
9            if instruction in ['mov', 'movsx', 'movzx']:
10
11               substring = '[' + self.found_op1 + '+0x2]'
12
13               if substring in op1 or substring in op2:
14                   self.clear()
15                   print "\"IsDebbugerPresent\" Detected!
                         Section: <%s> Address: 0x%s" % (section,
                         address)
```

B.3.5. Hook Detection

Hooking is a technique on which the execution control flow is deviated to a trampoline function having arbitrary code. This deviation can be used to action logging or system subversion, for example. This modification can be done by using system facilities, such as API calls, in the case of `SetWindowsHook` [Microsoft 2017e], or by performing binary changes, in the case of `detours` [Microsoft 2017a].

A variation of this technique, called `inline hooking`, consists of patching a binary with a `JMP` instruction. An evasive sample can check whether its own binary was in-memory patched by checking whether a given snippet 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 trick.

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

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

Listing 35. Hook detection trick detector.

```
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)
```

B.3.6. Heap Flags

Likewise global flags, PEB's heaps have also their own flags, as shown on table 9, which can be used as a analysis indicators by evasive malware.

Table 9. PEB's heap flags

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

Samples can perform Heap checks by using the API call to GetProcessHeap [Microsoft 2017b] or by implementing their own checks, retrieving the PEB base address at fs:0x30 offset and them referencing the default heap at 0x18 offset, as shown on Listing 36.

Listing 36. Heap Flags trick.

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

As a detector, one can save the resulting address associated to the PEB query and then search for this value plus the heap offset on the proceeding instructions, as shown on Listing 37.

Listing 37. Heap Flags trick detection.

```
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
```

```
11
12          if self.found_op is not None and ((op1 is not None and
                '[' + self.found_op + '+0x18]' in op1) or (op2 is not
13                                                                  None
                                                                    and
                                                                    '['
                                                                    +
                                                                    sel
                                                                    .
                                                                    fou
                                                                    +
                                                                    '+0
                                                                    x18
                                                                    ]'
                                                                    in
                                                                    op2
                                                                    )
                                                                    )
                                                                    :
14              self.clear()
15              print "\"HeapFlags\" Detected! Section: <%s> Address
                    : 0x%s" % (section, address)
```

B.3.7. Hardware Breakpoint Detection

Modern processors provide hardware-assisted debugging facilities, such as hardware breakpoints, which allow instruction addresses to be stored on special registers, and thus 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 when a hardware debugger is attached. The `0xc` offset represents the debugger context struct whereas the `0x4` offset means an access to the debug register number 0.

Listing 38. Hardware debugger detection trick.

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

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

Listing 39. Hardware debugger trick detector.

```
1  if instruction in ['mov','movsx','movzx'] and 'fs:0x0' in op1
       and 'rsp' in op2:
2              self.seh = True
3
4          elif self.seh==True and instruction in ['mov','movsx','
               movzx'] and 'rsp+0xc' in op2:
5              self.found_first=True
6              self.found_op=op1
7
8          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):
9              self.found_second=True
10
11         if self.found_second==True:
12             self.clear()
13             print "\"HardwareBreakpoint\" Detected! Section: <%s
                   > Address: 0x%s" % (section, address)
```

B.3.8. SS register

When running on a debugger, it replaces the first byte of given instructions by a trap flag. In order to be more transparent, many debugger solutions try to hide 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 trick

```
1          pop     ss
2          pushf
```

The implemented detection technique is to check the usage of the SS register immediately after it was pop-ed. The implementation of this detector is shown on Listing 41.

Listing 41. SS register trick detection.

```
1           if instruction in ['mov', 'movsx', 'movzx']:
2              if 'ss' in op1:
3                  self.found_ss = True
4          elif instruction== 'pop':
5              if 'ss' in op1:
6                  self.found_ss = True
7          elif 'pushf' in instruction:
8              self.found_flag=True
```

B.3.9. Software breakpoint

Unlike hardware breakpoints, which are register-based, and thus limited in number, software breakpoints are 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. Software debugger detection trick.

```
1 cmp     rax, 0xCC
```

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

Listing 43. Software debugger detection trick detector.

```
1 if 'cmp' in instruction:
2     if '0xcc' in op1 or '0xcc' in op2:
3         self.found_cmp = True
4 if self.found_cmp==True:
5     self.clear()
6     print "\"Software Breakpoint\" Detected! Section: <%s>
         Address: 0x%s" % (section, address)
```

B.3.10. SizeOfImage

Another trick able to defeat some debuggers 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. SizeOfImage trick.

```
1         mov     eax, [fs:0x30]
2         mov     eax, [eax+0xc]
3         mov     eax, [eax+0xc]
4         addw    [eax+20],0x1000
```

As a detector for this technique, one can look for signals of value changes om this field, as shown on Listing 45.

Listing 45. SizeOfImage trick detection.

```
1  if instruction in ['mov', 'movsx', 'movzx']:
2             if 'fs:0x30' in op2:
3                 self.found_op1 = op1
4                 self.found_keyword = True
5                 return False
6
7         if self.found_keyword:
8             if instruction in ['mov', 'movsx', 'movzx']:
9                 if '[' + self.found_op1 + '+0xc]' in op2:
10                    self.found_op2 = op1
11                    self.found_keyword2 = True
```

```
12
13          if self.found_keyword2:
14              if instruction in ['mov', 'movsx', 'movzx']:
15                  if '[' + self.found_op2 + '+0xc]' in op2:
16                      self.found_op3 = op1
17                      self.found_keyword3 = True
18
19          if self.found_keyword3:
20              if instruction in ['addw','add','sub']:
21                  if '['+self.found_op3 + '20]':
22                      print "\"SizeOfImage\" Detected! Section: <%
                            s> Address: 0x%s" % (section, address)
23                      self.clear()
```

A way to defeat such trick is to recompute the image size. This calculation can be performed using the `VirtualQuery` [Microsoft 2017f] API.

B.4. Anti-VM

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

B.4.1. VM Fingerprint

Some solutions leave presence indicators in the system, such as known strings. Thus, a straightforward way to detect an hypervisor 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 the BIOS code.

Listing 46. VMware fingerprint.

```
1  int dmi_check(void)
2          char string[10];
3          GET_BIOS_SERIAL(string);
4
5          if (!memcmp(string, "VMware-", 7) || !memcmp(string, "
               VMW", 3))
6                  return 1;                       // DMI contains
                       VMware specific string.
7          else
8                  return 0;
```

The same way, a straightforward evasion detection technique is to verify the presence of such verification strings on the suspicious binary. PEframe, for instance, implements such kind of verification. Listing 47 shows an excerpt of PEframe's implementation which performs such checks.

Listing 47. PEFrame's VM fingerprint detection.

```
1  VM_Str  = {
2                  "Virtual Box":"VBox",
3                  "VMware":"WMvare"
4          }
5
```

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

B.4.2. CPUID check

Another fingerprint approach is to make use of the `CPUID` instruction, which fills CPU registers with the vendor string. On VM cases, the hypervisor name is supplied. This way, a traditional evasive approach is to compare `CPUID` results to known VM-vendor strings, such as `Xen` or `QEMU`. The same way, a strightforward 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's CPUID trick detection.

```
1 VM_Sign = {
2         "Xen":"XenVMM",
```

Another detection possibility is to check the 31^{th} returned bit from `CPUID` instruction, which should return whether the processor has hypervisor capabilities or not [hexacorn 2014]. As presented by Vmware [VMWare 2010], this verification could be implemented as shown on Listing 49.

Listing 49. Vmware's hypervisor capabilities check.

```
1  int cpuid_check()
2  {
3          unsigned int eax, ebx, ecx, edx;
4          char hyper_vendor_id[13];
5
6          cpuid(0x1, &eax, &ebx, &ecx, &edx);
7          if  (bit 31 of ecx is set) {
8                  cpuid(0x40000000, &eax, &ebx, &ecx, &edx);
9                  memcpy(hyper_vendor_id + 0, &ebx, 4);
10                 memcpy(hyper_vendor_id + 4, &ecx, 4);
11                 memcpy(hyper_vendor_id + 8, &edx, 4);
12                 hyper_vendor_id[12] = '\0';
13                 if (!strcmp(hyper_vendor_id, "VMwareVMware"))
14                         return 1;               // Success -
                               running under VMware
15         }
16         return 0;
17 }
```

B.4.3. Invalid Opcodes

Hypervisors often 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 whether the environments answer properly or not. The code from [Bachaalany 2005], 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 PEframe, shown on Listing 51.

Listing 51. PEframe's VirtualPC detection.

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

B.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 table addresses checking.

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

According to [Radare 2008], the hypervisors' tables presents the values shown in the Table 10.

Table 10. Hypervisor's tables values.

Hypervisor	GDT	IDT
VMware 3.2	0xFFC05000	0xFFC6A370
VMware 4.0	0xFFC07000	0xFFC17800
VMware 4.5 Windows	0xFFC07C00	0xFFC18000
VMware 5.0/5.5 Windows	0xFFC07C80	0xFFC18000
VMware 5.0.0 (13124) Linux	0xFFC075A0	0xFFC18000
VMware GSX 3.1 Linux	0xFFC07000	0xFFC18000
VMware GSX 3.1 Linux	0xFFC07E00	0xFFC18000
VMware Player 1.0.1 Linux	0xFFC07880	0xFFC18000
Xen-2.0.7 (dom0 or domU)	0xFF400000	0xFC571C20
Kqemu 0.7.2	0xF903F800	0xF903F000
MS Virtual PC 2004	0xE80B6C08	0xE80B6408

This kind of IDT check can be found in practice on many samples. It first appear is credited to Joanna Rutkowska, on the non-academical literature. Listing 53 shows how this kind of check is usually implemented [Securiteam 2004].

Listing 53. IDT check implementation.

```
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 detectors use these bytes as signatures, such as on PEframe's implementation, shown on Listing 54.

Listing 54. PEframe's IDT check detection.

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

B.4.5. VMware Hypercall Detection

Similarly to the O.S syscalls, hypervisors have their own ways to be invoked by the running systems. These ways are usually named hypercalls. On Vmware-based systems, an hypercall is made by generating an I/O operation on a specific port (`VX`), present only on their virtualized systems. An evasive sample can try to write on this port and, if sucessful, 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 trick detector.

```
 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 2012], the code to check the `VX` port looks like the one presented on Listing 56

Listing 56. VMware detection trick.

```
__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 often consider the `VMx` string as a signatures of an anti-vm techniques, as shown on Listing 57.

Listing 57. PEframe's VMX detection.

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

B.4.6. A bit more about signatures

A similar approach is taken on `torpig` detection [MNIN.org 2006]. Its IDT check is used as a signature on PEframe (Listing 58) and on Snort (Listing 59).

Listing 58. PEframe's torpig detection.

```
1 VM_Sign = {
2         "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",
3         "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"
4         }
```

Listing 59. Snort's torpig detection.

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

B.5. FakeRet

We have previously discussed what happens when the detection window changes. We explain here how a inter-block trick could be implemented. In order to give a clear idea of what happens on such case, Listing 60 and Listing 61 show the `fakeret` code and the splited disassembly,respectively.

Listing 60. Fakeret implementation.

```
1 xor eax, eax
2 jmp     fakeret
3 call    puts
4 ret
5 fakeret:
6 jnz     main
```

Listing 61. Fakeret disassembly.

```
1   27:   31 c0                   xor    %eax,%eax
2   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
```

As can be noticed, the trick is placed right after the `RET`. In order to detect this trick, the detector would have to follow the `JMP` flow, not stopping at the end of the block. We can notice that the trick flows from the byte `28` to the `31` and then `37`. A linear check would proceed to the byte `29` instead.

B.6. Unaligned Evasion Techniques

We have previously mentioned YARA rules to detect unaligned tricks. These rules are presented in this section. Listing 62 shows the rule which checks for the `str` and `sidt` instructions, present on the `CPU Instruction` trick.

Listing 62. CPU Instruction detector implemented on YARA.

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

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

        condition:
                $str or $sidt
}
```

Listing 63 shows the rule corresponding to the `FakeJump` trick. The wildcards (?) are responsible for ignoring instruction immediates.

Listing 63. FakeJump detector implemented on YARA.

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

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

        condition:
                $seq
}
```

B.7. 32-bit Limitation

We have implemented the tricks for the x86 (32 bits) architecture. In order to detect the tricks on the x86-64 (64 bits) architecture, the corresponding offsets should be updated. The Table 11 shows the correspondence between known offsets.

Table 11. Mapping: x32 to x64

Value	x32	x64
PEB	fs:0x30	fs:0x60
NtGlobalFlag	0x68	0xbc
_HEAP	0x40	0x70

In addition, some techniques, such as the `Stealth Import` are only functional on 32-bits systems.