J Comput Virol (2007) 3:113–123
DOI 10.1007/s11416-007-0045-1

EICAR 2007 BEST ACADEMIC PAPERS

Rootkit detection from outside the Matrix

Sébastien Josse

Received: 12 January 2007 / Revised: 3 March 2007 / Accepted: 17 March 2007 / Published online: 15 May 2007
© Springer-Verlag France 2007

Abstract The main purpose of this article is to present a secure engine which is specifically designed for a security analyst when studying rootkits and all kinds of programs which interact at a deep level with the operating system, including Anti-Virus, Personal Firewall and HIPS programs. State-of-the-Art algorithms for rootkit detection are presented in this paper. Forensic techniques to monitor the system's critical components and advanced heuristics are also used. This survey is based on a proof-of-concept human analysis framework which puts forward a reliable system for automatically gaining information about a rootkit and its interaction with the OS executive, but focuses on human decision as a detection process without the same limitations or constraints as product-oriented anti-rootkit programs. We use the new point of view provided by this framework to take a fresh look at heuristics and forensics which are currently used by rootkit detectors.

1 Introduction

In [35], we can find a formal definition of a stealth virus by using recursive functions of the complexity theory. Using this formalism, a stealth virus is defined as a virus that modifies its execution environment in such a way that a program controlling the system through system calls will not detect the presence of the virus. As far as stealth virus detection is concerned, they prove that the general problem of stealth virus detection is Σ_3 complete. Therefore, we can see that the underlying complexity problem is too high to expect to find a generic algorithm that matches this type of Malware.

A rootkit can be defined as a program which implements a set of stealth techniques [7]. Therefore, we can understand a rootkit as a technology. A rootkit is also commonly associated with a specific class of program, kernel modules, which execute at the most privileged security level of the CPU.

There are many locations where a program can install itself and hide within a standard operating system like Windows NT, Mac OS or Linux. With the apparition of rootkits, the host-based sword against shield battle has a new theatre of operation including: the kernel land, the CPU registers, the BIOS (Basic I/O System). By examining the execution flow, starting from an I/O request through until the effective hardware operation, we can note that there are many slots for corruption and hiding. We can also observe that many ingenious (or undocumented) ways to drop a kernel module into the system have always been found by rootkits designers.

Thus the main problem in rootkit detection is the wide perimeter of this large battlefield: in fact the whole operating system, including BIOS, Registry, file systems, boot sectors, CPU registers, etc. Moreover, Rootkit technology is evolving very fast, with rootkits that exploit the growing structural complexity of operating systems and new functional features of hardware components to hide. For example, Blue Pill [31] and Vitriol [36] are a proof-of-concept rootkits which implants a thin VMM beneath the host OS by utilizing AMD's Pacifica [34] or Intel VT [21] virtualization extensions of the CPU instruction set. In both cases, the rootkit VMM is installed while running in Ring 0 and then the running OS is migrated into a VM.

Sébastien Josse is an I.T. consultant at Silicomp-AQL Security Evaluation Lab and also a Ph.D student EDX Polytechnique Doctoral School within the ESAT Virology and Cryptology Lab in Rennes sebastien.josse@esat.terre.defense.gouv.fr.

S. Josse (✉)
Silicomp-AQL - Security Evaluation Lab,
1 rue de la Châtaigneraie, 51766, Cesson-Sévigné, France
e-mail: Sebastien.Josse@aql.fr

The second problem is that a rootkit detection engine has to make a diagnosis on a system which may already be corrupted, or which may be corrupted during analysis, thus subverting the security diagnosis. An operating system which bases its security on the frank separation between two privilege levels, is an open space when both Malware and Detector are executing at the same privilege domain.

Anti-Virus, Personal Firewall and HIPS (Host Based Protection Systems) designers have to face several difficulties when dealing with rootkits. An anti-rootkit program has to face the same constraints as a rootkit program while executing on an operating system: it has to be installed as deeply as possible within the operating system, and its installation has to be robust against low-level viral attacks. In particular, it has to be stealthy. In order to be able to (proactively) gain information, it has to corrupt several of the operating system components. This constraint induces problems of stability. In this Sword against Shield battle, the two parts use the same weapons.

Human analysts do not have the same constraints while analysing the behaviour of such a program. They can run the program in an emulated environment and gather the required information in order to check the security aspects of the targeted program. They can drive transformations on the targeted program, such as unpacking, in order to obtain an unprotected version of the program, which can then be analysed statically. The static analysis of the targeted program can involve powerful tools and algorithms, with no limitation on the use of resources, in order to obtain a de-obfuscated variant/version of a program. Specialized debuggers can be used in order to carry out a dynamic analysis (and to examine the internal objects of the system). Several anti-rootkit products and forensics tools can be used successively to refine the diagnosis. The whole analysis can be conducted within a virtual environment which can be very difficult to distinguish from a real one.

These levels of freedom are far less real for a proactive security product: for performance reasons, emulation engines are necessarily much poorer. Thus, Rootkits can implement detection routines which forbid any efficient emulation process.

Analysis is not driven by a human and is thus faced with difficulties that can be avoided during a human interactively-driven analysis process while disassembling and dynamically analysing a program. The goal of this paper is to present the general specifications of a secure and advanced analysis framework for rootkit analysis based on a virtual CPU. This anti-rootkit function does not suffer the same limitations as those described above. The design of this analysis tool is isolation and stealth oriented.

The main idea and design principle of this analysis framework is to contribute to forensics from outside the emulated PC. We call the virtual PC and its operating system the Matrix, in which a rootkit will evolve (Fig. 1). Like an acupuncturist, we apply precise and fine probes in order to take stealthy measurements and arrive at diagnoses.

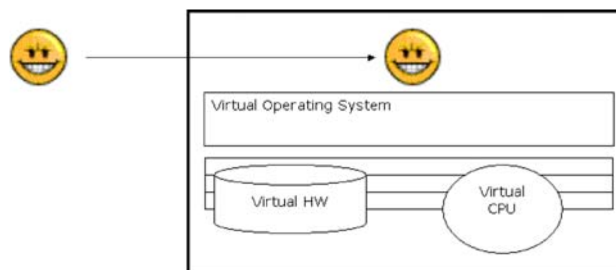

Fig. 1 Rootkit from outside the Matrix

The remainder of this paper is structured as follow:

- Section 2 presents related work in the field of rootkit detection.
- Section 3 discusses the design and implementation details of our proposed solution. This section provides also an experimental evaluation of its effectiveness, through examples of utilization.
- Section 4 discusses the contribution of this paper, in regard to existing solutions and gives the limitations and usage conditions.
- Section 5 concludes and outlines future work.

2 Related work

Many algorithms and heuristics have been proposed very recently by anti-virus researchers. Among these methods, we find exact identification detection methods based on form analysis, statistical generic methods (applied from inside the Matrix), and several algorithms which intend to reveal information about the way a rootkit installs and maintain hooks in the target system.

Integrity checking methods (comparison between two snapshots of a system's critical components). A huge number of works exists; among them - for the latest - are WinPE GhostBuster [17], SVV (System Virginity Verifier) [30], CoPilot [1] and Microsoft KPP [23], also known as Patch Guard.

Model checking methods consist of making a comparison between two values of a parameter which is representative of the underlying system statistical model. The distribution of a clean system and the distribution of a corrupted system are discriminated by using such an estimator. Model checking methods show promising results. A statistical approach inspired by steganography from an information theoretic viewpoint has been proposed in [14]. Such an approach is currently implemented in PatchFinder [29]. The principle

of this rootkit detection tool is to detect API hooks by comparing the number of instructions executed during invocation of several API functions with a clean reference (under the hypothesis that hooks cause extra instructions to be executed that would not be called by unhooked functions).

Consistency checking methods consist of checking the consistency of several objects, given information of the intrinsic rules they have to obey. Several tools exist among them SIG∧2 KprocCheck [24] and VICE [8] both look for any ServiceTable entries containing function pointers that point outside the memory image of `ntoskrnl.exe` (for SSDT) or `win32k.sys` (for Shadow SSDT).

Cross-view based detection methods consist of making a comparison between two points of view of several system objects: for example, a high-level view of these objects and a low-level view of the same objects. Microsoft SysInternals RootkitRevealer [9] detects hidden registry entries and hidden files by parsing the registry hives and filesystem at a very low level and without the help of Win32 API, and comparing the result with what is obtained through standard Win32 API calls. F-Secure BlackLight [4] and IceSword [20] both use cross-view based methods to detect process and files that are hidden from the user and security software.

Signature based detection methods consist of comparing a pattern to a signature base, in order to identify a certain rootkit. In the case of stealth Malware, this approach is limited. We have to expect that a rootkit leaves sufficient trace to allow its identification. RAIDE [6] uses a memory signature scanning method in order to find `EPROCESS` blocks hidden by rootkits. SIG∧2 APIHookCheck detects the presence of system-wide API hooks that are implemented based on insertion of jmp instructions at the start of the real API.

So it can be seen that many methods are available for detecting use of rootkit technology. Cross-view based, form or behavioural analysis methods are regularly proposed by anti-virus researchers. Their efficiency is however a real challenge, especially when a rootkit has already taken place in the host. As a matter of fact, inside-the-Matrix rootkit detection tools are easy to detect and possibly evade. Execution within a controlled environment like a virtual machine could solve the problems of isolation and stealth.

3 Design of our solution

We now go on in this section to present an approach based on emulation in order to analyse rootkits securely and reliably. Our tool is designed for security analysts. It covers at least the following range of use:

- Rootkit analysis
- Anti-Virus, Personal Firewall and HIPS products testing

We use the new standpoint provided by our framework (i.e. we gain information about the rootkit by observing the whole corrupted host system from outside) to take a fresh look at heuristics and algorithms which are currently used by rootkit detectors.

3.1 Core emulation engine

We have adapted the QEMU [3] emulator in order to implement the core emulation engine of our framework. The general software design is very similar to TTAnalyze, a tool for analysing Malware [2].

The virtual machine embeds a kernel service which communicates through a virtual network interface with the virtual machine monitor. This communication channel is used to upload the target binaries into the virtual machine, to start the execution of the main target program and to get information from the kernel which makes it possible to drive the execution of the guest process from the host system. It should be noted that this information, which is located in the guest NT executive's tables and structures, can be obtained from kernel mode (the way of obtaining it from kernel mode is well documented in [26,27]) or by forensic (and stealth) methods.

The human analysis framework receives information about:

- Win32 and native API calls of the target program
- Sequentially disassembled code of the target program
- Structure of the executable in guest memory (and dynamic comparison with the raw file)
- Several internal structures/objects of the operating system: interrupt descriptors, `SYSENTER_EIP` register, (shadow) (system) service descriptors, process double-linked list, driver objects, I/O request packets tables, etc.

3.2 Analysis modes

The main program can be used in two console modes:

- default mode automatically uploads the target executable into the VM, possibly unpacking it and getting required information about its functional interactions with the guest operating system;
- interactive mode makes it possible for the security analyst to dynamically drive the execution of the target executable and interact with the VM, by controlling its states.

The main program can also be used in graphics mode. This mode is useful when the analysis process requires interactions between the user and the target program through a graphical interface. The same options as in console mode

(default or interactive) are available when using the graphical mode.

3.3 API calls tracing

Among the information that can be extracted from an executable, the most important is the nature of its interaction with the operating system. Thus, we trace both Win32 and native[1] API calls. This first information is enough to understand how any stealth Malware drops its driver into the Kernel.

Let us give an example. We upload and start the migbot rootkit (inside the emulator):

```
P> upload migbot.exe
[INFO] download target migbot.exe 27136
[INFO] status: OK
P> execsuspend
P> resumeexec
P> disconnect
```

Then we examine the log file:

```
[APICALL] 7C80C7B1: call function 7C80C7B1
                    C:\WINDOWS\system32\kernel32.dll::FindResourceA
[HOOK]    FindResourceA[IN]  lpType=MIGBOT
[HOOK]    FindResourceA[IN]  lpName=BINARY
[HOOK]    FindResourceA[OUT] HRSRC=00407048
                             ...
[APICALL] 7C91D682: call function 7C91D682
                        C:\WINDOWS\system32\ntdll.dll::ZwCreateFile
[HOOK]    ZwCreateFile[IN]  ObjectName=\??\C:\MIGBOT.SYS
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000000 |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
                             ...
[APICALL] 7C80AC28: call function 7C80AC28
                   C:\WINDOWS\system32\kernel32.dll::GetProcAddress
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=ZwSetSystemInformation
[HOOK]    GetProcAddress[OUT] ProcAddress=7C91E729
                             ...
[APICALL] 7C91E729: call function 7C91E729
              C:\WINDOWS\system32\ntdll.dll::ZwSetSystemInformation
[HOOK]    ZwSetSystemInformation[IN]
                  SystemInformationClass=SystemLoadAndCallImage(38)
[HOOK]    ZwSetSystemInformation[OUT] NTSTATUS=000000F0
```

We obtain information about the way this rootkit drops its kernel module, firstly by determining the location of its body (a binary resource), then writing this binary to disk and then installing and starting the module within the kernel using the `ZwSetSystemInformation` system call.

[1] Windows NT native API is the set of system services provided by the Windows NT executive to both user mode and kernel mode programs.

By using this trace function, we are able to diagnose the method used by a rootkit to drop its kernel module whatever the method used (via SCM with `OpenSCManager`, `CreateService`, `OpenService` or via NtDLL with `ZwLoadDriver`, `ZwSystemDebugControl`, `ZwSetSystemInformation`, or by any other undocumented method).

If the dropper is packed, we first have to unpack it using the procedure described in [22].

3.4 Objects under monitoring

The next set of information that is crucial when dealing with rootkit is a view of all locations within the host platform where a hook can be installed. A first stage in our analysis method is thus to walk through the executive structures of the operating system in order to identify the potential targets of a rootkit attack.

We also need to identify and monitor all the hardware components that could be corrupted by a rootkit: BIOS flash memory, CPU registers, boot sectors, etc.

Once all this information is collected, we can put forward algorithms to help the security analyst make his diagnosis, starting with the most commonly used heuristics.

We apply forensic integrity controls at different stages of a target rootkit's life cycle. We make a snapshot of crucial structures and objects and monitor each modification. The

principle is to enumerate the target objects using forensics methods, and then to use a hash function or a field-to-field comparison of the structures.

The following objects, tables and structures are currently monitored:

- BIOS, MBR (Master Boot Record), IVT (Interrupt Vector Table)
- Process memory code sections, IAT/EAT tables
- Process and thread objects, SDT (Service Dispatch Table)
- Kernel code, kernel SSDT (System Service Dispatch Table) and IDT (Interrupt Descriptors Table)
- Driver objects, IRP_MJ tables
- CPU registers (SYSENTER_EIP, LDT, GDT, CR0, CR3, EFLAGS)

Our tool is designed to run its checks at every cycle of the virtual CPU. Thus, it is immune to timing attacks. Such an attack is feasible using a hardware solution like Copilot because it is designed to run its checks periodically (every 30s in the present prototype).

Our tool is immune to race conditions. We access physical and virtual memory directly from outside, without any interaction with the virtual hardware. Such an attack is feasible using a hardware solution like Copilot because the Copilot monitor accesses host memory only via the PCI bus.

Our tool is immune to CPU cache attacks. At each virtual CPU cycle, we can access the processor cache. Since it is possible to maintain a consistent view of Copilot's monitored memory while hiding malicious code elsewhere such as in the processor Cache, this code would remain undetected by Copilot.

3.5 Modules integrity check

The integrity of all loaded modules (including NtOsKrnl and Hal) is verified on demand or at each virtual CPU cycle. If corruption is detected, the code section, the IAT and EAT[2] integrity of the target module are checked. Let us give an example: We install an inline hook in a target executable (inside the emulator):

```
C:\>withdll /d:traceapi.dll parano.exe
```

Now examine the log file:

```
[DETECT] inline hook detected
```

We notice that an alert has been raised because one of the functions imported by the target executable has been modified. We monitor the code section, the EAT and IAT of all modules in memory in order to detect any attempt made by a rootkit to install a hook.

[2] The IAT (import Address Table) is used when an application uses an API function to import its address. The EAT (Export Address Table) is used by a DLL to export its API functions.

3.6 IDT and Sysenter register

The IDT (Interrupt Descriptor Table) is used to find interrupt handlers. We use the virtual CPU register to locate this table in virtual memory (SegmentCache IDTInfo= virtual_cpu->idt). We then parse the IDT table.

Let us give an example: we start the strace rootkit (inside the emulator):

```
C:\>pslist paranoiac
    Process        PID
    Paranoiac.exe  1436
C:\>strace 1436 10000
```

Now examine the log file:

```
[INFO]   IDT Base: 8003F400 Limit:000007FF
+--------+--------------------+---+---+
| IDT    |  Selector:Handler  |DPL| P |
+--------+--------------------+---+---+
|  00    |   0008:804dfbff    | 0 | 1 |
|  01    |   0008:804dfd7c    | 0 | 1 |
|  02    |   0058:0000112e    | 0 | 1 |
   ..           ...             .   .
|  2e    |   0008:804deea6    | 3 | 1 |
   ..           ...             .   .
|  fd    |   0008:804ded32    | 0 | 1 |
|  fe    |   0008:804ded39    | 0 | 1 |
|  ff    |   0008:804ded40    | 0 | 1 |
+--------+--------------------+---+---+

[INFO]   IDT Base: 8003F400 Limit:000007FF
+--------+--------------------+---+---+
| IDT    |  Selector:Handler  |DPL| P |
+--------+--------------------+---+---+
|  00    |   0008:804dfbff    | 0 | 1 |
|  01    |   0008:804dfd7c    | 0 | 1 |
|  02    |   0058:0000112e    | 0 | 1 |
   ..           ...             .   .
|  2e    |   0008:fa00f2a0    | 3 | 1 |
   ..           ...             .   .
|  fd    |   0008:804ded32    | 0 | 1 |
|  fe    |   0008:804ded39    | 0 | 1 |
|  ff    |   0008:804ded40    | 0 | 1 |
+--------+--------------------+---+---+
[DETECT] IDT hook detected
```

We notice that an alert has been raised because the INT 2E entry in IDT has been modified. Any program which modifies the IDT will be detected. Moreover, we obtain information about the location of the rootkit routine in memory.

INT 2E and SYSENTER instructions are issued by NtDLL.dll to trap the Kernel. In order to detect IA32_ SYSENTER_EIP rootkit attack, we need only to monitor the corresponding virtual CPU register virtual_cpu-> sysenter_eip.

Let us give an example: we install and start the rootkit sysenter (inside the emulator):

```
C:\>instdriver -Install sysenter sysenter.sys
C:\>instdriver -Start sysenter
```

Now examine the log file:

```
[INFO]   sysenter_eip=804def6f
[INFO]   sysenter_eip=fa02e49a
[DETECT] sysenter hook detected
```

We notice that an alert has been raised because the SYSENTER_EIP has been modified. Any program which modifies the SYSENTER_EIP will be detected. Moreover, we obtain information about the location of the rootkit routine in memory.

3.7 Process and driver objects

The double-linked list of process structures is parsed from virtual memory, starting from the initial system process location. The whole structure is monitored in order to detect any attempt to suppress one of its elements or to modify its critical values (security token for privilege elevation purpose, etc.).

Let us give an example: the Fu rootkit [16] can hide processes and device drivers. It can also elevate process privilege and groups. All this is done by DKOM (Direct Kernel Object Manipulation). Fu is a play on words from the UNIX program su used to elevate privilege.

We start the fu rootkit (inside the emulator):

```
C:\>pslist paranoiac
Process        PID
Paranoiac.exe  1436
C:\>fu -prs 1436 SeLoadDriverPrivilege
```

Now, look at the log file:

```
[DETECT] EPROCESS corruption detected
         (privilege elevation)
```

We notice that an alert has been raised because the EPROCESS structure has been modified. Any program which modifies the EPROCESS structure will be detected.

Let us suppress one element of the EPROCESS double-linked list:

```
C:\>fu -ph 1436
```

Now examine the log file:

```
[INFO]   activeLink=815b3718, PID=0000059c,
         Name=paranoiac.exe
[DETECT] EPROCESS DKOM detected
```

We notice that an alert has been raised because the EPROCESS double-linked list has been modified. Any program which modifies the EPROCESS double-linked list will be detected.

The double-linked list of module entry structures is parsed from virtual memory, starting from the PsLoadedModuleList symbol value which is exported by NtOsKrnl. The whole structure is monitored in order to detect any attempt to suppress one of its elements (kernel module hiding) or to modify any of its critical values.

Let us suppress one element of the DRIVER_OBJECT double-linked list:

```
C:\>fu -phd paranoiac.sys
```

Now examine the log file:

```
[INFO]   activeLink=815b2f98, imageBase=f8c95000, imageSize=00053000,
                          drvPath=\SystemRoot\system32\DRIVERS\srv.sys
         activeLink=816f1c50, imageBase=f8d95000, imageSize=0000f000,
  drvPath=\??\C:\Documents and Settings\Sébastien Josse\paranoiac.sys
         activeLink=8055ab20, imageBase=f8aec000, imageSize=00041000,
                         drvPath=\SystemRoot\System32\Drivers\HTTP.sys
[INFO]   activeLink=816f1c50, imageBase=f8c95000, imageSize=00053000,
                          drvPath=\SystemRoot\system32\DRIVERS\srv.sys
         activeLink=815716c0, imageBase=f8aec000, imageSize=00041000,
                         drvPath=\SystemRoot\System32\Drivers\HTTP.sys
[DETECT] DRIVER_OBJECT DKOM detected
```

We notice that an alert has been raised because the DRIVER_OBJECT double-linked list has been modified. Any program which modifies this double-linked list will be detected.

3.8 IRP Major functions

IRPs (I/O Request Packet) are handled by NT executive and drivers to communicate buffered data or control code to a user-mode program. We parse the IRP Major functions table of each driver object within the guest operating system executive virtual memory in order to detect any hook in these crucial tables. In order to get at these driver objects using only forensic methods, we have to walk the whole device tree and explore each node of this structure in virtual memory.

Let us give an example: we install and start the rootkit irphook (inside the emulator):

```
C:\>instdriver -Install irphook irphook.sys
C:\>instdriver -Start irphook
```

Now examine the log file:

```
[DETECT] IRP_MJ hook detected
```

We notice that an alert has been raised because the IRP Major Functions Table within a `DRIVER_OBJECT` has been modified. Any program which modifies this table within any `DRIVER_OBJECT` will be detected. Moreover, we obtain information about the location of the rootkit handler in memory.

3.9 Service dispatch tables

The SSDT (System Service Dispatch Table) is used by NT executive for handling system calls. By parsing NtOSKrnl PDB symbol file, we can locate the service descriptor table in virtual memory. We parse the whole SSDT and monitor any modification that is made to this table. The same monitoring operation is done with the SHADOW SSDT (Shadow SSDT).

Let us give an example: we install and start the rootkit hideprocess (inside the emulator):

```
C:\>instdriver -Install hideprocess
                hideprocess.sys
C:\>instdriver -Start hideprocess
```

Now examine the log file:

```
[INFO]   _service_descriptor[0]=804e2d20
         _service_descriptor[1]=00000000
         _service_descriptor[2]=0000011c
         _service_descriptor[3]=804d8f48
         service_table[000]=80586691, argument_table=18
         service_table[001]=805706ef, argument_table=20
                                        ...
         service_table[173]=8057cc27, argument_table=10

         service_table[282]=80648e3c, argument_table=10
         service_table[283]=8062c033, argument_table=00
[INFO]
         service_table[000]=80586691, argument_table=18
         service_table[001]=805706ef, argument_table=20
                                        ...
         service_table[173]=fa044486, argument_table=10

         service_table[282]=80648e3c, argument_table=10
         service_table[283]=8062c033, argument_table=00
[DETECT] SSDT hook detected
```

We notice that an alert has been raised because an entry in the SSDT has been modified. Any program which modifies the SSDT (or SHADOW SSDT) will be detected. Moreover, we obtain information about the location of the rootkit handler in memory.

3.10 Boot process monitoring

During the booting up process, BIOS transfers execution to code from some other medium such as disk drive, CD-ROM or via network boot (BIOS PXE[3] agent requests and downloads a boot file from the BOOTP/TFTP server).

- In the case of a booting up process from a disk drive, the BIOS loads the first sector of the disk drive (MBR[4]) which locates a bootable partition in the partition table and executes the first sector of this boot partition. The partition boot sector loads and executes the next boot stage of the OS.
- In the case of a booting up process from a CD-ROM, the BIOS loads boot code from the CD-ROM which executes the next boot stage of the OS.
- In case of a network boot, BIOS executes PXE agent, which downloads and executes the network boot file. The network boot file loads and executes the next boot stage of the OS.

Windows boot loader loads and executes NtLdr, which creates GDT and IDT, and maps and executes OsLoader. OsLoader loads the OS (NtDetect, Hal, NtOsKrnl, BootVid, etc.).

Some rootkits exploit the reliance of Windows start-up upon the BIOS. Windows start-up uses BIOS interrupts, so IVT is mostly preserved. Moreover, it has to respect memory ranges reserved by the BIOS. BIOS rootkit like eEye BootRoot or BootKit exploit this trust to function like BIOS hooks.

[3] Preboot eXecution Environment.

[4] Master Boot Record.

We monitor several locations (BIOS and VGA BIOS reserved range of memory, MBR, real-mode IVT, etc.) in order to detect any attempt to take control before Windows boot loader.

Let us give an example: eEye's BootRoot [25] is a proof-of-concept rootkit presented as an exploration of technology that custom boot sector code can use to subvert the Windows kernel as it loads. It is a boot sector-based NDIS backdoor that demonstrates the implementation of this technology. It executes after the BIOS but before the operating system, enabling complete control over the operating system.

We install and start the eEye BootRoot from a (virtual) floppy drive. Now examine the log file:

```
[INFO] IVT
+-----+--------------------+
| INT |    ADDR            |
+-----+--------------------+
| 00  |  f000ff53          |
| 01  |  f000ff53          |
  ..       ...
| 13  |  f000e3fe          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[INFO] IVT
+-----+--------------------+
| INT |    ADDR            |
+-----+--------------------+
| 00  |  f000ff53          |
| 01  |  f000ff53          |
  ..       ...
| 13  |  9f80005c          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[DETECT] IVT corruption detected
```

BootKit [5] is a project related to custom boot sector code subverting Windows NT Security Model. The sample presented currently keeps escalating cmd.exe to system privileges every 30 s. It patches the kernel at runtime. BootKit is PXE-compatible. Quite a similar result is obtained while booting on the virtual BootKit CD-ROM:

```
[INFO] IVT
+-----+--------------------+
| INT |    ADDR            |
+-----+--------------------+
| 00  |  f000ff53          |
| 01  |  f000ff53          |
  ..       ...
| 13  |  9e000068          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[DETECT] IVT corruption detected
```

Both rootkits hook INT 13h to patch OS files (OsLoader, etc.) as they load. The boot process monitoring function is under development. By tracing every CPU instructions (starting from time t0: first BIOS instruction is executed) and by comparing this trace with a clean boot process, we should be able to detect any attempt to execute additional code after the BIOS but before the operating system. By monitoring a wider range of virtual hardware locations, we should be able to detect next generation BIOS viruses [12] such as BootRoot [25], ACPI[5] BIOS rootkits [18] and PCI[6] rootkits [19].

3.11 Model checking

We propose a generic approach to specify detection algorithms based on statistical tests (and thus suffering errors rates) as introduced in [15]. This detection method tries to decide if a given system, observed from outside (using forensics methods) is corrupted by a stealth Malware. No information about the identity of the rootkit or about the way it corrupts the target system is provided by this generic method.

PatchFinder intends to detect API hooks by comparing the number of instructions executed during invocation of several API functions with a clean reference. The estimator chosen by this statistical detection test is thus the average m of instructions executed during invocation of a given API call. This estimator describes D_1.

Let $x_s(t)$ denote the number of CPU instructions executed during the system call s on the clean system and $x'_s(t)$ denote the number of CPU instructions executed during the same system call s on an unknown system.

The principle of PatchFinder is to compare those two values. If the difference is too high, the system is considered as compromised by a rootkit. Let's see how this approach can be transposed in an outside-the-Matrix context (passive analysis): We define the null and alternative hypothesis:

$$\begin{cases} H_0: & \text{the system call } s \text{ is not hooked} \\ H_1: & \text{the system call } s \text{ is hooked} \end{cases}$$

For t from 1 to n, we execute a code in virtual memory that calls s. Next, we calculate the average CPU instructions number m.

We denote m_0 the average number instructions executed during invocation of the API call s on the clean system and m the same calculation on the unknown system.

By choosing the m estimator we can define a bilateral test as follows:

$$\begin{cases} H_0: m = m_0 \\ H_1: m \neq m_0 \end{cases}$$

[5] Advanced Configuration and Power Interface.

[6] Peripheral Component Interconnected.

Indeed, if a hook is installed on the system for the s function, the average number of CPU instructions m is either larger than m_0 (if the hook instrumentalizes the function s by adding functionalities) or smaller than m_0 (if the hook inhibits the function s).

Given an accepted false positive rate, if $|m - m_0|$ is greater than a given decision threshold, we consider that the function s is hooked (and thus that the target system is corrupted).

By analysing the system from the outside, we are completely stealthy. The detection function is thus more difficult to circumvent. Our sampling can be made by executing code that calls s from a random memory area. By taking n measurements, we are able to estimate the probabilistic distribution law under H_0 more precisely and thus detect (with a controlled false positive rate) any suspicious variation when compared to this law.

4 Evaluation

4.1 So what is new?

We propose a reliable, stealthy and secure analysis system for automatically (or interactively, if needed) gaining accurate information about a rootkit. We get accurate information about its interactions with the OS executive, particularly about the way it installs and maintains hooks in the target system.

The design of our tool is isolation and stealth oriented. By providing a completely controlled environment, we can analyse a rootkit without any risk for the real system.

The whole analysis is conducted within a virtual environment (the Matrix), which can be very difficult to distinguish from a real one, thus increasing the technical skills required by rootkit designers, in order to forge a rootkit that could evade or spread.

Our approach enables a stealthy and reliable snapshot of the critical Hardware/Kernel objects which are usually patched by Malwares implementing rootkit technology. This monitoring function is as stealthy as possible because we use an accurate and complete CSIM (including virtual CPU and hardware components), on which an unmodified version of the operating system executes.

These controls can occur at any time during the life cycle of the monitored rootkit: load-time, run-time or boot-time. As a matter of fact, we could apply a check at every CPU cycle. We thus have a great degree of freedom and a smooth granularity.

By observing the whole of the corrupted host system from the outside, we offer a new standpoint which allows for the study of the current rootkit detection algorithms (inside-the-Matrix) and to validate their relevance in this new context (outside-the-Matrix).

When compared to hardware-based rootkit detection solutions, and because the whole machine can be inspected at any time during the analysis process, our solution does not suffer the same limitations and does not have their vulnerabilities (timing attacks, CPU cache attacks, race conditions).

In comparison with the current inside-the-Matrix rootkit detection tools, the same detection algorithms are more difficult to circumvent:

- integrity checking functions do not have to face the constraints and rules imposed by the operating system or by the hardware. Stealth measurements can be made without disturbing the operating system and without being locatable. Since rootkits do not execute at the same level as the detection function, they can no longer disturb their diagnosis.
- for the same reasons model checking algorithms are more robust and difficult to evade. Moreover, because we can sample and make finer-grained random measurements at higher frequencies (at every CPU cycle, if needed), it is possible to specify more efficient implementations of these types of data mining algorithms.

Classic algorithms apply but without the same operational constraints. As a consequence, it is much easier to apply most of them jointly and possibly increase the scope of the analysis.

During the analysis process, we can drive complex transformations on the targeted program, such as unpacking, and thus analyse possibly armoured rootkit.

Most rootkit currently do not implement software protection such as packing. Our unpacking engine, which is a plug-in of our analysis framework, provides a generic unpacking functionality which could be useful in the future while analysing rootkits.

4.2 Limitations

In the context of rootkit analysis, two security requirements must be covered: isolation (propagation containment)é1and stealth (if emulation is detected, the target rootkit will no longer provide information). Several additional security functions are mandated in order to increase the security of our proposed solution. In order to be isolated and secure, an emulator must implement robust device protection, filtering and network quarantine mechanisms. At least, a Firewall must be installed on the host system in order to monitor the network communication interfaces between host and guest operating systems. We are currently exploring the possibility of inhibiting every interface between host and guest systems, providing a highly isolated VM. Natively, QEMU does not support guest-to-host or host-to-guest communication since it is intended to behave like a completely stand-alone machine. It is

possible to upload the target executable into the VM, without using any standard communication channel, by using forensics techniques: writing the target files directly on the raw virtual disk. The task is not so easy.

We will recall now the problem of detection of an emulated environment by a rootkit, which is crucial in the context of rootkit detection using emulation: since knowing that it is running on a virtual machine is the first step in a viral evasion attack attempt, a secure emulator must simulate the hardware components as precisely as possible and must be resilient against pattern matching recognition algorithms and any hardware functionality scan[7]. It is very difficult to simulate the hardware components of an emulated PC perfectly.

Other mechanisms can be implemented by a Malware to recognize an emulated environment. Several detection methods have long been documented for detecting commercial VMM (Virtual Machine Monitor) like VMware or VirtualPC [10,28,33]. New methods have recently been discovered in order to detect VMware, VirtualPC, Bochs, Hydra, QEMU and Xen [11]. According to their author, only CSIM (Complete Software Interpreter Machine) can approach complete transparency. Bochs, Hydra, and QEMU, all suffer from bugs and limitations that allow their detection, but these are problems that are possibly fixed.

Concerning QEMU emulator, among the most frequently encountered problems, due to incorrect emulation, we find:

- problem in emulating self modifying code (a self overwriting `rep` sequence for example), since using dynamic translation,
- wrong returns value of some CPU instruction (wrong returns value of `cpuid` instruction for processor name or Easter egg on AMD CPU, for example),
- unexpected behaviour of the result of some CPU instructions (`CMPXCHG8B` instruction does not always write to memory, for example),
- limitations in the exception handling code (Double Fault exception is not supported, for example).

In order to be as stealth as possible, these bugs and limitations of the core emulation engine must be corrected. Thus some work has to be done to increase the security level of a rootkit detector based on a core emulation engine.

5 Conclusion and future work

As we have seen in Sect. 2 of this paper, some interesting detection methods, currently implemented by specialized forensics tools, should be transposed to our rootkit analysis module:

- Cross-view based methods could be implemented by injecting code into the virtual machine, in order to provide more or less low level system calls.
- Consistency checks could be implemented on several kernel structures by providing intrinsic bounds characterizing a healthy state.
- Signature based methods could be implemented by controlling several locations of the system (for example the kernel module part of the rootkit, its dropper, the memory area where it loads, etc.) at crucial stages during the execution of the targeted program or during the life cycle of the whole virtual system (boot time, for example).
- Model checking methods could be also implemented, as specified in Sect. 3 of this paper.

We mentioned in the abstract of this paper that our analysis tool applies to rootkit but also to any security product which interacts deeply with the operating system in order to ensure its robustness or to hide information by installing hooks.

Indeed, we can use this framework to check the robustness of security mechanisms and to recover information on their internal working procedures: as we have seen in Sect. 3 of this paper, we are able to collect many data which reveal possible hiding locations.

We could implement some more specialized diagnosis modules in order to assess the robustness of security functions and mechanisms through a black-box or grey box approach:

- for example, by using a black box approach, we could extract the non-detection Boolean function under its Disjunctive Normal Form [13]. Thus we could measure the quality of an anti-rootkit product as far as its knowledge base and its detection algorithms are concerned. This approach is valid if the testing method is based on form analysis: pattern matching algorithms, API calls sequences analysis, etc. A black-box approach and combinatorial modelling are enough to underwrite the effectiveness of a detector.
- the ability to apply code patching transformations at instruction or basic block level during the target security product's execution (possibly using a reliable unpacking functionality) and to modify any location on the whole platform is a critical component of any information extraction or fault injection system.

We could also validate learning detection algorithms and the calculation of similarity indices in several statistical detection models. Many statistical models, such as Markov models, have been used in DNA sequence analysis, and can be

[7] A Hardware functionality scan is an authentication method proposed by Microsoft. The driver exercises complex inner workings of a chip and checks for correct responses. In this way, the driver can authenticate that a chip really is valid hardware.

used in metamorphic virus family analysis and recognition. The implementation of such algorithms on viral sets is technically difficult using a static approach. A dynamic approach, such as the one provided by emulation, is more suited than a static approach.

Putting all this potential use of emulation together (operating system internals integrity check, dynamic binary code analysis, statistical detection and learning algorithms validation on an intra-procedural level), it is easy to imagine the benefits that could be gained from using an outside-the-Matrix framework to help the security analyst do his job.

References

1. Arbaugh, W.A., Fraser, J.T., Molina, J., Petroni, N.L.: Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. Available at: http://www.usenix.org/events/sec04/tech/full_papers/petroni/petroni_html/main.html, (2004)
2. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: a tool for analyzing malware. In: proceedings of the 15th EICAR Conference, Hamburg, Germany, April 29 - May 3, 2006. In Journal in computer Virology, EICAR 2006 Special Issue, V. Broucek et al. Editor (2006)
3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the 2005 USENIX Conference (2005)
4. BlackLight.: Available at: http://www.f-secure.com/blacklight/, (2006)
5. BootKit.: Available at: http://www.rootkit.com/vault/vipinkumar/, (2007)
6. Butler, J.: RAIDE: rootkit analysis identification elimination. Available at: http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Silberman-Butler.pdf, (2006)
7. Butler, J., Hoglund, G.: Rootkits: subverting the Windows kernel. Addison Wesley, ISBN 0-321-29431-9 (2006)
8. Butler, J., Hoglund, G.: VICE - Catch the hookers! (Plus new rootkit techniques). Available at http://www.rootkit.com/, (2006)
9. Cogswell, C., Russinovich, M.: RootkitRevealer. Available at: http://www.sysinternals.com/, (2006)
10. Elias: Detect if your program is running inside a Virtual Machine. 14 Mars 2005. Retrieved from: http://lgwm.org (Elias homepage), (2005)
11. Ferrie, P.: Attacks on virtual machine emulator. In: proceedings of AVAR 2006 Conference, Auckland, New Zealand, December 3–5, (2006)
12. Filiol, E.: Introduction to computer viruses: from theory to applications. IRIS International Series, Springer, Heidelberg (2005)
13. Filiol, F.: Malware pattern scanning schemes secure against black-box analysis. In: proceedings of the 15th EICAR Conference, Hamburg, Germany, April 29 - May 3, 2006, and In: Broucek, V., Turner, P. (eds.) Eicar 2006 Special Issue, J. Comput. Virol. 2(1), pp. 35–50 (2006)
14. Filiol, E.: Techniques virales avancées, IRIS Series, Springer Verlag France, January 2007. An English translation is pending (due mid 2007)
15. Filiol, F., Josse, S.: A statistical model for undecidable viral detection. In: proceedings of the 16th EICAR Conference, Budapest, Hungary, May 5 - 8, 2007. In: Broucek, V. (ed.) Eicar 2007 Special Issue, J Comput Virol 3(2), (2007)
16. Fu.: Fu rootkit. Available at: https://www.rootkit.com/vault/fuzen_op/, (2006)
17. GhostBuster.: the Strider GhostBuster Project. Avalaible at: http://research.microsoft.com/rootkit/, (2006)
18. Heasman, J.: Implementing and detecting an ACPI BIOS Rootkit, Black Hat Europe (2006)
19. Heasman, J.: Implementing and detecting a PCI rootkit, Available at: http://www.ngssoftware.com/, (2006)
20. IceSword.: IceSword, Available at: http://xfocus.net/tools/200509/1085.html, (2006)
21. IntelVT.: Intel Virtualization Technology, Available at: http://www.intel.com/technology/virtualization/, (2007)
22. Josse, S.: Secure and advanced unpacking using computer emulation. In: proceedings of the AVAR Conference, Auckland, New Zealand, December 3–5, (2006)
23. KPP.: Kernel Patch Protection: Frequently asked questions, Available at: http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx, (2006)
24. KprocCheck.: SIG∧2 KprocCheck, Available at: http://www.security.org.sg/, (2006)
25. Permeh, R., Soeder, D.: eEye BootRoot: A Basis for Bootstrap-Based Windows Kernel Code, Available at: http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf, (2006)
26. Russinovich, M.E., Solomon, D.A.: Inside Microsoft Windows 2000, 3rd edn. Microsoft Press, ISBN 0-7356-1021-5 (2000)
27. Russinovich, M.E., Solomon, D.A.: Microsoft windows internals, 4th edn: Microsoft Windows Server 2003, Windows XP, and Windows 2000, (2004)
28. Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction. Retrieved from: http://www.invisiblethings.org/papers/,(2004)
29. Rutkowska, J.: Detecting Windows Server Compromises with Patchfinder 2. Retrieved from: http://www.invisiblethings.org/papers/, (2004)
30. Rutkowska, J.: System virginity verifier, defining the roadmap for malware detection on windows system. Hack in the box security conference, September 28th -29th 2005, Kuala Lumpur, Malaysia (2005)
31. Rutkowska, J.: Subverting VistaTM kernel for fun and profit. SyScan'06 July 21st, 2006, Singapore & Black Hat Briefings 2006 August 3rd, 2006, Las Vegas (2006)
32. Szor, P.: The art of computer virus research and defense, Addison-Wesley, ISBN 0-321-30454-3 (2005)
33. Z0mbie.: Z0mbie. VMWare has you. Retrieved from: http://vx.netlux.org/, (2001)
34. Zeichick, A.: Coming soon to VMware, microsoft, and Xen: AMD virtualization technology solves virtualization challenges, Available at: http://www;devx.com/amd/Article/30186/, (2005)
35. Zhou, M., Zuo, Z.: Some further theoretical results about computer viruses, In: The computer journal, vol. 47, N°6 (2004)
36. Zovi, D.A.D.: Harware virtualization rootkits. Black Hat Federal 2006, Washington D.C., January 25th (2006)