UNVEILING THE KERNEL: ROOTKIT DISCOVERY USING SELECTIVE AUTOMATED KERNEL MEMORY DIFFERENCING

Ahmed Zaki & Benjamin Humphrey
Sophos, UK

Email {ahmed.zaki, benjamin.humphrey}@sophos.com

ABSTRACT

As an increasing number of automated malware analysis systems become mainstream, the emphasis on the relevance of the data extracted from the analysis task increases. Conceptually, automated malware analysis systems provide information about a sample and also identify modifications to a computer system induced by the sample. Traditionally, the focus of such analysis systems has primarily been on monitoring process, disk and network-level behaviour with varying levels of granularity. While providing a varied set of information, these systems offer limited ability to identify and classify rootkits. The very nature of rootkits makes them hard to classify (and in some cases even detect) using these scanning techniques. Kernel memory modifications can indicate that samples are trying to conceal information or hijack execution paths, thus exhibiting malicious behaviour. In an environment with a large throughput of analysis jobs, the need arises for an efficient and accurate way to identify such complex threats that could otherwise be misclassified or pass unnoticed. We present a system for identifying rootkit samples that is based on automated analysis. In this system we recognize the performance and memory constraints of a high-throughput environment; instead of monitoring modifications to the whole memory, we capture changes to data structures and memory regions that, on a *Microsoft Windows* operating system, are known to have been targeted by rootkits in the past. We explain the reasons behind the design decisions and how they have reflected on identifying different classes of rootkits. In our research, we also explore the effectiveness of using this model as a standalone component to identify malicious behaviour. In order to do this, we run a large set of known clean versus malicious files to identify traits that could be indicative of malicious activity.

1. INTRODUCTION

The volume of malware is increasing exponentially. The number of samples seen by anti-virus vendors each day now exceeds the personnel resources available to analyse and classify them. In order to reduce the workload, automated analysis systems are used to identify, classify and cluster samples based on their malicious behaviour. Using static (examining file attributes) and dynamic (behavioural) techniques, these systems examine each sample, and usually make a reasoned assumption about its maliciousness.

Kernel-mode rootkits pose a particular problem when it comes to automated analysis. A rootkit is 'malicious software that allows an unauthorized user to maintain access to a computer by concealing programs and processes, files, or data from the operating system' [1]. Kernel-mode rootkits are a sub-class of rootkits that introduce changes in the kernel memory of an operating system to conceal or manipulate information provided by the kernel itself. Examples of kernel-mode rootkits include Necurs [2], TDL [3] and Turla [4]. A rootkit's ability to hide in the operating system increases the likelihood that automated analysis systems will fail to produce any intelligible data. Tools such as *GMER* [5] can be used to detect rootkit activity, but scan times can be long and the tools can be cumbersome to automate. In an environment where the throughput of samples is in the tens of thousands, speed of analysis is certainly an issue.

To create an automated analysis system that can detect kernel mode rootkits, it is necessary to create a reliable method of identifying rootkit-like behaviour that is efficient and fast to keep analysis times to a minimum. We propose a system that fulfils these criteria, based on the *Cuckoo Sandbox* [6] modified to use the *Sophos* AV engine. Using the engine, selected areas of the kernel are extracted and saved to create 'Before' and 'After' snapshots to identify the changes made to the kernel. By examining these changes it is possible to identify rootkit activity.

2. SYSTEM SPECIFICATIONS

We use a *Windows 7* SP1 x86 virtual machine in a *VirtualBox* virtualization environment, version 4.3.10. The kernel memory differencing is implemented as part of a *Cuckoo Sandbox* installed on *Ubuntu OS* version 12.04.

The guest virtual machine has the *Sophos* anti-virus engine installed by default. Detections are reported as part of the analysis information and sent to the host component. Once a sample executes, a kernel memory scan is started by the analyser inside the guest machine. In addition to scanning for known malware traits in kernel memory, the scan dumps specific kernel structure information to disk in JSON format. (More on this data is explained in Section 2.1.) This data is saved to the server module with other standard behavioural analysis information as part of each analysis.

In order to capture kernel memory changes made by the sample, we need a baseline snapshot of the kernel data before execution of the sample (the baseline data). The baseline data is associated with a virtual machine snapshot. Each snapshot has a unique identifier and baseline data. The baseline data is generated by performing a dummy analysis with no files submitted. The results of the analysis are tagged as baseline data, and comparisons are made with data obtained from subsequent analysis tasks.

Cuckoo

We utilize the *Cuckoo Sandbox* as a base framework for automating sample analysis. *Cuckoo Sandbox* is an open-source malware analysis system [6]. It supports running different types of files using different virtualization technologies.

Cuckoo Sandbox's modularity makes it ideal for our purposes. It can be split into a host-side management component and a guest-side component running in a virtual machine (VM). The host-side component is responsible for the core execution of the analysis task: starting a VM, preparing an analysis task for execution (packaging the guest-side components that run inside the VM to collect analysis information), collecting the raw data resulting from the analysis, processing it, and finally running a set of signatures and reports to present the results. The guest-side component runs selected modules packaged by the host-side component and returns the results to it [6].

Cluster requirements

In building an efficient automated analysis system capable of processing high sample volumes, both time and stability are key requirements. We did not want the time taken to complete a job to extend beyond four minutes. This time interval captures the total time from the sample submission to the report generation. By adhering to this requirement we could be confident that our system would work efficiently in a cluster environment.

We can split the overall task into three main time frames:

1. Preparation and initialization – starting the VM and sending the sample to the VM for execution and analysis.
2. Analysis – running the sample in the VM, returning the raw data for processing, and powering off the VM.
3. Processing and reporting – kernel memory data differencing and executing the signatures against the processed results.

Using generic anti-rootkit tools inside the VM was not feasible due to the time it takes to complete a scan (minutes with some tools), the inconsistent formats of the log files produced by the various scanners, and the footprint some of them have on the kernel memory, which taints the collected information. A tool like *GMER* [5], for example, could take more than two minutes to perform an exhaustive scan.

We measured the time taken for the kernel memory data to be produced with our additions, and it added 10–15 seconds to the standard analysis time. The default time taken from the point the VM stops until the report is produced can be between 20 and 80 seconds. The time is dependent on the amount of processing performed on the raw results retrieved from the job, the number and performance of the signatures that are run on the results, and the type of report to be produced. On evaluating our signatures and the differencing mechanism, we found that the added time is between one and two seconds.

Given the minimal amount of time added, we decided to maximize utility. We used a default analysis timeout of 120 seconds and set our kernel memory scan to start after 105 seconds. By using those intervals we allowed more time for the sample to run and make changes to kernel memory and disk so that the changes would be picked up during the scan. We are aware that this setting could be costly in a cluster environment, since many samples do not execute in automated systems and thus we would be wasting the 120 seconds. The timeout and scan start settings can be altered to reduce the delay before the scan. The benefit for us was that the feature did not add a significant time delay in producing the necessary information.

The overall analysis task time from sample submission to report generation using *VirtualBox* on a standalone machine took between three and four minutes. This included the initialization, analysis and reporting time. This length of time is acceptable in a cluster environment.

2.1 Usage of the Sophos AV Engine

The requirements for the kernel examination component of this system are:

- A presence in the kernel.
- The ability to dump areas of kernel memory selected at runtime.
- The ability to write dumped memory to a JSON file.

The decision to use the *Sophos* AV engine to gather the required data was based on it meeting the above requirements. The *Sophos* AV engine is already capable of dumping and scanning kernel memory, negating the need to design and implement new software to do so. Since we were already using the *Sophos* AV engine to scan for malicious traits during analysis, we decided to leverage its presence to gather the desired information.

Selecting the areas of the kernel to examine

To select which areas of the kernel to examine, it is necessary to understand what effect kernel-mode rootkits have on the kernel. For a kernel-mode rootkit to have a presence in the kernel it must load a driver. The driver will then modify various functions or pointers to hide malicious activity from the user, other programs, and sometimes from itself.

Drivers

A driver object represents an individual driver in the system [7]. The DRIVER_OBJECT structure contains information about a driver including its name, initialization routine, unload routine, a pointer to the loaded driver image, a pointer to its device object, and its MajorFunction dispatch table.

When a rootkit loads its driver, a DRIVER_OBJECT structure is created for it. The DriverStart field points to the loaded image of the rootkit driver. When the initialization routine is executed, as well as creating its own driver object, a rootkit driver can modify the driver objects of other drivers.

To track changes, driver objects and device objects are enumerated, and data from each is extracted and dumped. For speed and memory efficiency, only specific fields from the structures are dumped.

Modules

A kernel module is a loaded image file that resides in kernel address space. The address space typically comprises drivers and the *Windows* kernel. Currently, the information dumped to the JSON file is minimal, consisting of the module name, image base and module size. By comparing before and after snapshots of the module list, it is possible to tell which modules have been added (indicating the possible presence of a new driver), and which modules have been changed. Changes to the image base or size of a module suggest that the module has been replaced, thus indicating the presence of a rootkit.

System Service Dispatch Table (SSDT)

The System Service Dispatch Table (SSDT) is an array of function pointers that resides in kernel address space. The entries in the array reference core functions exported by the *Windows* kernel (e.g. ntoskrnl). These functions provide an interface for user-mode programs to perform tasks such as opening a file or querying a directory. By modifying the address of a function pointer or the code that an entry points to, rootkits can hook a function and control the data that is passed to and from it.

To detect changes to the SSDT, the entire table is dumped. The addresses in the table are checked (during processing) against the ones saved in the baseline snapshot to detect any hooks. Typically, when the code of a function is hooked, an assembly language 'jmp' instruction is written to the beginning of the function which jumps to hook the code. To detect such code hooks, the first eight bytes of each function are also dumped, to enable the processing module to compare them against the baseline snapshot.

Interrupt Descriptor Table (IDT)

The IDT is a processor-specific array that has 256 entries. Each entry (an interrupt vector) in the array contains information about its corresponding interrupt handler. Interrupts are operating system conditions that divert the processor to code from the normal flow of execution of a program [8]. These interrupts are called (using the INT x assembly language instruction) by both hardware and software. As defined by *Intel*, the first 32 entries are reserved for exceptions [8] (i.e. INT 0 - Division by 0), and the rest are 'user' defined. The *Windows* operating system and various core system drivers also install interrupts, but there remain spare entries that rootkit developers can use to their advantage.

The entire IDT for each processor is dumped to the JSON log to enable us to find new IDT additions and hooks.

Callbacks

A callback is a function that is executed on completion of a specific event. The *Windows* operating system provides routines that allow driver writers to register callbacks on a number of events that rootkit writers can exploit. The routines that are of most interest are:

- CreateProcessNotify – triggered when a new process is created
- LoadImageNotify – triggered when a new image is loaded
- CreateThreadNotify – triggered when a new thread is created
- CmRegister – triggered when an operation is performed on the registry.

A rootkit registering any of these callbacks will be able to hide processes, images, threads or registry entries upon creation.

The callbacks of each type are enumerated and added to the JSON log.

Disk information

Rootkits reside in kernel address space, but bootkits begin outside the operating system. Their execution begins in either a modified Master Boot Record (MBR) or a modified Volume Boot Record (VBR). A common task of a bootkit is to load a driver, typically from hidden storage at the end of the disk, which may proceed to hide the modified MBR/VBR. As the bootkit executes before the operating system is fully started, it can bypass detection by security software.

To detect the presence of a bootkit, checksums of the MBR and VBRs for all partitions are written to the JSON log. In addition to the initial boot code, the MBR also contains the disk's partition table [9]. The entire partition table is also dumped and written to the JSON log so that changes in partition location/size (also indicative of a hidden file system) can be detected.

2.2 A processing module

Cuckoo's processing modules are Python scripts that analyse the raw data from the guest-side component of the sandbox and append the result to a global data structure [6]. To identify changes in the kernel data we had to add a custom processing module that would compare the data received from the analysis to the baseline data associated with the guest machine. Since new analysis tasks are started from the same snapshot, we do not encounter noise caused by ASLR in kernel memory space.

After differencing both data structures and identifying any new changes, the processing module produces two dictionary objects. One object encompasses all the changes identified (the diff object), while the second object includes all the information retrieved from the analysis task (the task object). Both objects are then appended to the global container which holds information produced by other processing modules. The task object will include information about driver objects, modules, kernel callbacks, the system service dispatch table, the interrupt descriptor table and hashing of the master boot record, along with information about each partition defined in the partition table.

The diff object will show whether information has been changed, added or deleted. The comparison mechanism is agnostic to the number or name of the data; it assumes both objects are JSON formatted. Post comparison, noise is cancelled from the diff object. We identified noise heuristically and used a list approach to identify driver objects and modules that we do not include in the diff output. We decided to exclude noise coming from the following modules, since it was almost always due to paging in:

- \windows\system32\autochk.exe
- \systemroot\system32\drivers\kmixer.sys
- \systemroot\system32\drivers\spsys.sys
- \systemroot\system32\drivers\asyncmac.sys
- \windows\system32\devobj.dll
- \windows\system32\setupapi.dll

Some drivers will experience the creation of devices once the execution starts, while others will change DEVICE_OBJECT addresses. The device object attached to \Driver\usbhub named '\Device_HID00000000' gets a different address each time the job starts. The same problem occurs for \Driver\mouclass, which is the class driver for all mouse devices. Namely, a

'\Device\PointerClass' DEVICE_OBJECT gets a different address. This different DEVICE_OBJECT gets attached to the list of devices of \Driver\mouhid and \Driver\hidusb.

We also found less common cases of \Driver\swenum and \Driver\asyncmac having devices attached to them, so we also filtered noise coming from them.

The drivers from which noise was filtered were:

- \Driver\mouhid
- \Driver\usbhub
- \Driver\mouclass
- \Driver\hidusb
- \Driver\umbus
- \Driver\asyncmac

There was noise introduced in the SSDT information returned due to paging of certain portions of the ntoskrnl.exe. We excluded all changes introduced due to paging out, and tagged data that had been paged in.

2.3 A driver analysis package

Cuckoo uses separate packages to instruct the analyser inside the guest machine to conduct the analysis procedure [6]. By default, *Cuckoo* provides packages to handle running dynamic link libraries (DLL), PDF, VBS and DOC files, among other packages. *Cuckoo Sandbox* is unable to process driver files for analysis by default. Since kernel-level malicious alteration is normally introduced by a malicious kernel module, we wanted the ability to run driver files alone to capture their pure behaviour.

The driver analysis package (sys package) gets invoked if the analyser identifies that the file is of native type. Since our guest machine is 32-bit, we restricted the analysis package to 32-bit driver files.

The sys package will invoke Sc.exe, which is provided in the *Windows* Resource Kit to register the service. This is done by running the command:

```
sc create newservicebinPath=<C:\path_to_file.sys>
type=kernel start=auto error=normal
```

The default parameters used to create the service are:

- Servicename=newservice
- Type=kernel
- Start=auto
- Error=normal

Each parameter can be changed by the user when a sample is submitted for analysis.

'Sc create' is the equivalent of calling the CreateService Win32 API.

The sys package supports two ways to execute the driver file. The default method uses the NtLoadDriver *Windows* API. The other method for execution is using sc.exe by running the command:

```
sc start newservice
```

This is equivalent to calling the StartService Win32 API.

Using sc.exe, which is a command-line interface to the service control manager, takes care of the prep work necessary to register a service in the service control manager database.

'sc create' creates the service registry key under HKLM\System\CurrentControlSet\services, which both NtLoadDriver and 'sc start' use to start the service. The sys package also provides an option to try both methods (sc.exe and NtLoadDriver) to check which one succeeds.

2.4 The signatures

Cuckoo provides an easy way to interpret the results of an analysis by matching the results via signatures. The signatures will look for predefined patterns in the results and flag them if they are present [6]. Each signature has an attribute called 'severity' that is used to define how malicious a signature is. This allows a user to write purely informative signatures in addition to ones that indicate malicious behaviour. (We explain more about the values we assigned to each signature to build a malicious index in section 3.2.)

We have written a set of signatures in order to identify certain modifications to the kernel data retrieved from an analysis task. Table 1 shows the signatures and their definitions.

2.5 The report

We built a template in the Django framework used by *Cuckoo* to show all the information we gathered from an analysis task. The report is split into seven tabs, with each tab showing the information specific to a certain category in the results data container.

Kernel memory changes

This category shows the changes to the kernel introduced during the analysis. For example, in Figure 1 we can see that a new device was attached to \Filesystem\FltMgr and \Filesystem\RAW. Also, the MBR SHA1 has been changed.

Drivers

This section shows a breakdown of all the driver objects found in kernel memory after running the sample. The breakdown also shows the I/O request packet handlers for each object.

Modules

This section shows all modules found on the system by enumerating the linked list of modules starting with ntoskrnl.exe. For each module we get the name, address and size.

SSDT

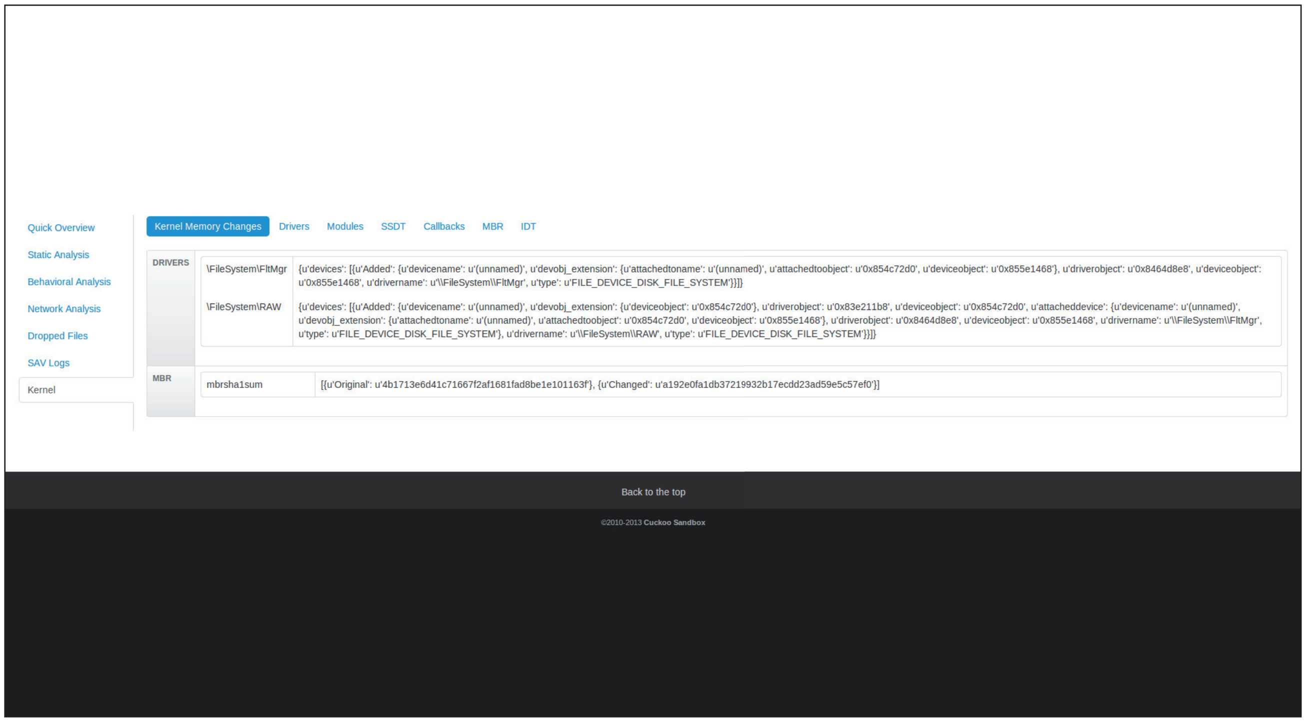
This section shows the entries in the system service dispatch table. For each system call we get its address and starting bytes. This shows if any inline hooks are added to the system calls.

Callbacks

These sections show any callbacks installed by any drivers. For

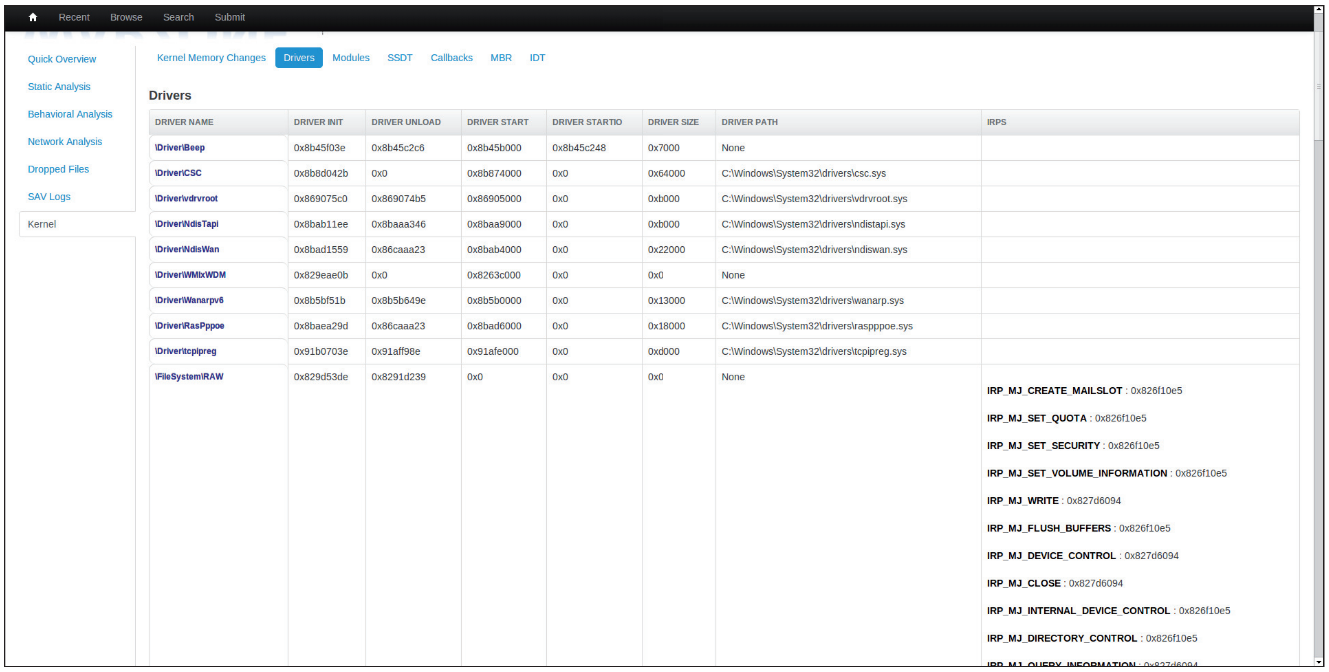
Signature name	**Definition**
generic_new_driver	Flags driver objects that have been created in the analysis job.
generic_modified_driver	Flags changes to already existing driver objects.
generic_deleted_driver	Flags driver objects that have been deleted during an analysis job.
generic_new_module	Flags new modules created during an analysis job.
generic_deleted_module	Identifies modules that were present before the analysis job took place, but which aren't present anymore.
generic_modified_module	Identifies changes to properties such as size or address of a module.
generic_ssdt_hook	Identifies changes to the system service dispatch table. Namely, changes in the address of any of the system calls (and flags them).
generic_idt_hook	Identifies changes to the interrupt descriptor table of any processor.
generic_new_callback	Identifies new driver-supplied callbacks. We check: • LoadImageNotify • CreateThreadNotify • CreateProcessNotify • CmRegister (registry callbacks)
generic_modified_callback	Identifies changes to the table holding driver-supplied callbacks.
generic_attached_device	Identifies new devices added to driver objects.
generic_modified_mbr	Identifies if the SHA1 checksum of the MBR has changed during the analysis.
generic_modified_vbr	Identifies if the SHA1 checksum of the VBR has changed during the analysis
generic_modified_eod_size	Identifies if the size of the unformatted region beyond the file system at the end of disk has changed during the analysis.

Table 1: Signatures and definitions.



DRIVERS	\FileSystem\FltMgr	{u'devices': [{u'Added': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x854c72d0', u'deviceobject': u'0x855e1468'}, u'driverobject': u'0x8464d8e8', u'deviceobject': u'0x855e1468', u'drivername': u'\\FileSystem\\FltMgr', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}}]}
	\FileSystem\RAW	{u'devices': [{u'Added': {u'devicename': u'(unnamed)', u'devobj_extension': {u'deviceobject': u'0x854c72d0'}, u'driverobject': u'0x83e211b8', u'deviceobject': u'0x854c72d0', u'attacheddevice': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x854c72d0', u'deviceobject': u'0x855e1468'}, u'driverobject': u'0x8464d8e8', u'deviceobject': u'0x855e1468', u'drivername': u'\\FileSystem\\FltMgr', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}, u'drivername': u'\\FileSystem\\RAW', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}}]}
MBR	mbrsha1sum	[{u'Original': u'4b1713e6d41c71667f2af1681fad8be1e101163f'}, {u'Changed': u'a192e0fa1db37219932b17ecdd23ad59e5c57ef0'}]

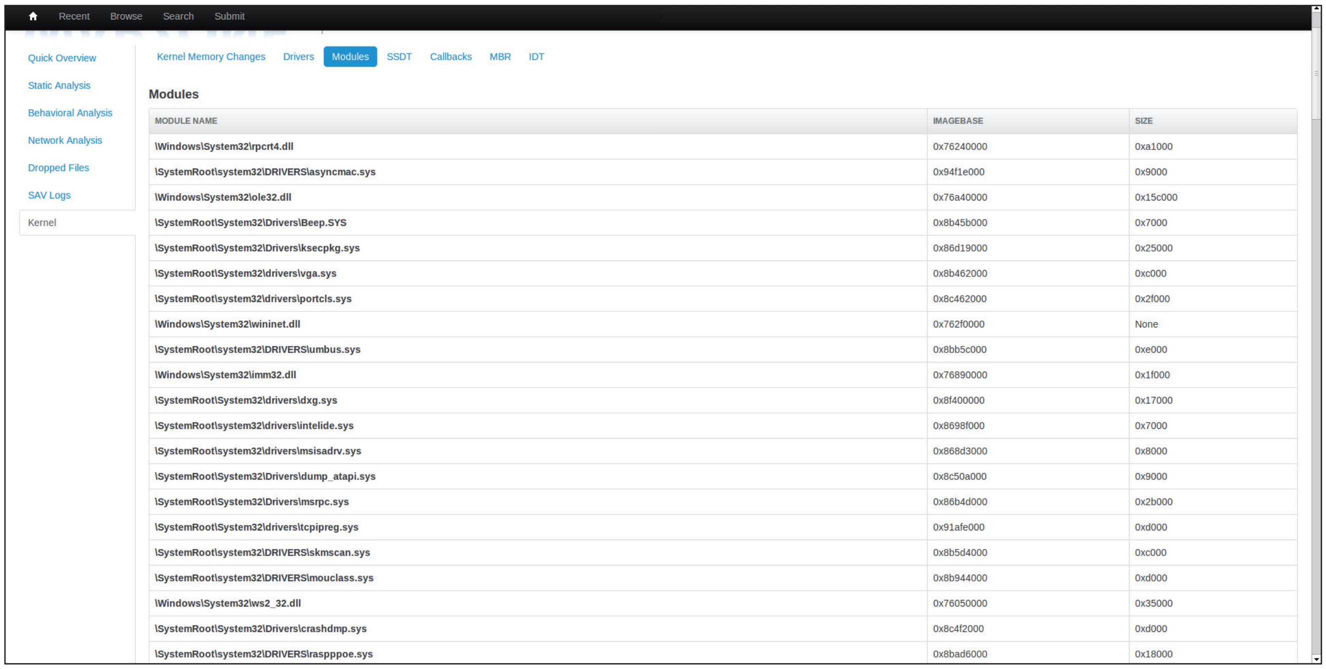
Figure 1: A new device attached to \Filesystem\FltMgr and \Filesystem\RAW.



Drivers

DRIVER NAME	DRIVER INIT	DRIVER UNLOAD	DRIVER START	DRIVER STARTIO	DRIVER SIZE	DRIVER PATH	IRPS
\Driver\Beep	0x8b45f03e	0x8b45c2c6	0x8b45b000	0x8b45c248	0x7000	None	
\Driver\CSC	0x8b8d042b	0x0	0x8b874000	0x0	0x64000	C:\Windows\System32\drivers\csc.sys	
\Driver\vdrvroot	0x869075c0	0x869074b5	0x86905000	0x0	0xb000	C:\Windows\System32\drivers\vdrvroot.sys	
\Driver\NdisTapi	0x8bab11ee	0x8baaa346	0x8baa9000	0x0	0xb000	C:\Windows\System32\drivers\ndistapi.sys	
\Driver\NdisWan	0x8bad1559	0x86caaa23	0x8bab4000	0x0	0x22000	C:\Windows\System32\drivers\ndiswan.sys	
\Driver\WMIxWDM	0x829eae0b	0x0	0x8263c000	0x0	0x0	None	
\Driver\Wanarpv6	0x8b5bf51b	0x8b5b649e	0x8b5b0000	0x0	0x13000	C:\Windows\System32\drivers\wanarp.sys	
\Driver\RasPppoe	0x8baea29d	0x86caaa23	0x8bad6000	0x0	0x18000	C:\Windows\System32\drivers\raspppoe.sys	
\Driver\tcpipreg	0x91b0703e	0x91aff98e	0x91afe000	0x0	0xd000	C:\Windows\System32\drivers\tcpipreg.sys	
\FileSystem\RAW	0x829d53de	0x8291d239	0x0	0x0	0x0	None	**IRP_MJ_CREATE_MAILSLOT** : 0x826f10e5 **IRP_MJ_SET_QUOTA** : 0x826f10e5 **IRP_MJ_SET_SECURITY** : 0x826f10e5 **IRP_MJ_SET_VOLUME_INFORMATION** : 0x826f10e5 **IRP_MJ_WRITE** : 0x827d6094 **IRP_MJ_FLUSH_BUFFERS** : 0x826f10e5 **IRP_MJ_DEVICE_CONTROL** : 0x827d6094 **IRP_MJ_CLOSE** : 0x827d6094 **IRP_MJ_INTERNAL_DEVICE_CONTROL** : 0x826f10e5 **IRP_MJ_DIRECTORY_CONTROL** : 0x826f10e5 **IRP_MJ_QUERY_INFORMATION** : 0x827d6094

Figure 2: Drivers.



Modules

MODULE NAME	IMAGEBASE	SIZE
\Windows\System32\rpcrt4.dll	0x76240000	0xa1000
\SystemRoot\system32\DRIVERS\asyncmac.sys	0x94f1e000	0x9000
\Windows\System32\ole32.dll	0x76a40000	0x15c000
\SystemRoot\System32\Drivers\Beep.SYS	0x8b45b000	0x7000
\SystemRoot\System32\Drivers\ksecpkg.sys	0x86d19000	0x25000
\SystemRoot\System32\drivers\vga.sys	0x8b462000	0xc000
\SystemRoot\system32\drivers\portcls.sys	0x8c462000	0x2f000
\Windows\System32\wininet.dll	0x762f0000	None
\SystemRoot\system32\DRIVERS\umbus.sys	0x8bb5c000	0xe000
\Windows\System32\imm32.dll	0x76890000	0x1f000
\SystemRoot\System32\drivers\dxg.sys	0x8f400000	0x17000
\SystemRoot\system32\drivers\intelide.sys	0x8698f000	0x7000
\SystemRoot\system32\drivers\msisadrv.sys	0x868d3000	0x8000
\SystemRoot\System32\Drivers\dump_atapi.sys	0x8c50a000	0x9000
\SystemRoot\System32\Drivers\msrpc.sys	0x86b4d000	0x2b000
\SystemRoot\System32\drivers\tcpipreg.sys	0x91afe000	0xd000
\SystemRoot\system32\DRIVERS\skmscan.sys	0x8b5d4000	0xc000
\SystemRoot\system32\DRIVERS\mouclass.sys	0x8b944000	0xd000
\Windows\System32\ws2_32.dll	0x76050000	0x35000
\SystemRoot\System32\Drivers\crashdmp.sys	0x8c4f2000	0xd000
\SystemRoot\system32\DRIVERS\raspppoe.sys	0x8bad6000	0x18000

Figure 3: Modules.

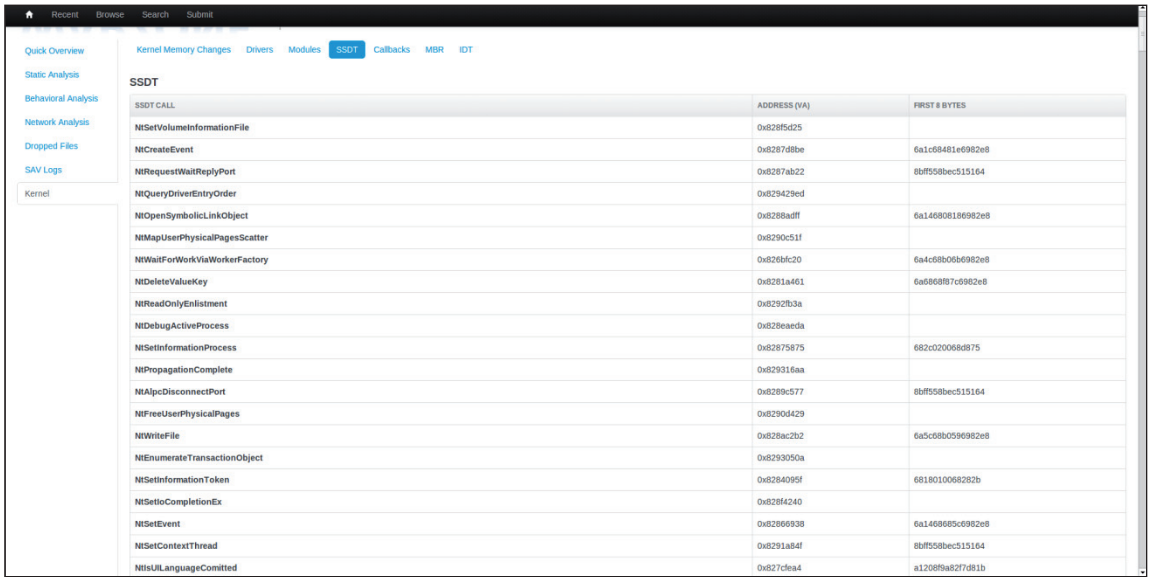
each callback we list its type, the driver path that installed it, the address and SHA1 of the driver.

MBR

This section contains information available in the MBR and its SHA1. In addition to listing each partition's detailed information by parsing the partition table entries, we also list the SHA1 of the VBR that this partition points to and the address of the last sector of the formatted file system and the last sector of disk.

IDT

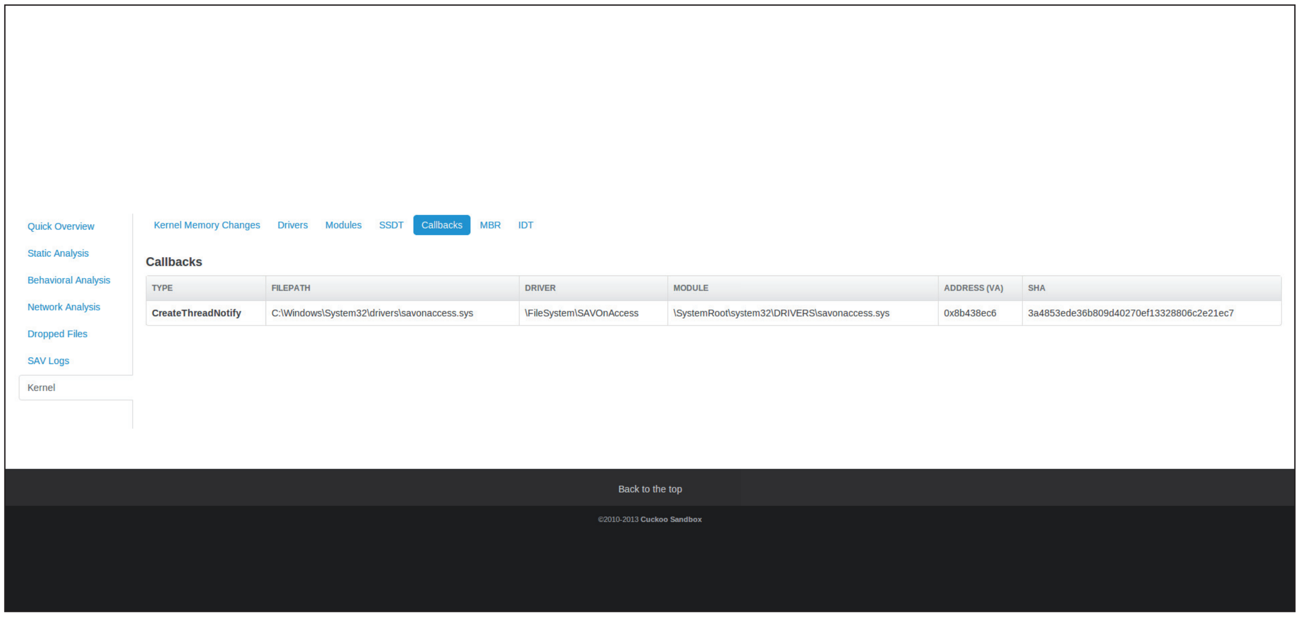
This section shows the descriptors in the interrupt descriptor table. For each descriptor we show its type and address.



SSDT

SSDT CALL	ADDRESS (VA)	FIRST 8 BYTES
NtSetVolumeInformationFile	0x828f5d25	
NtCreateEvent	0x8287d8be	6a1c68481e6982e8
NtRequestWaitReplyPort	0x8287ab22	8bff558bec515164
NtQueryDriverEntryOrder	0x829429ed	
NtOpenSymbolicLinkObject	0x8288adff	6a146808186982e8
NtMapUserPhysicalPagesScatter	0x8290c51f	
NtWaitForWorkViaWorkerFactory	0x826bfc20	6a4c68b06b6982e8
NtDeleteValueKey	0x8281a461	6a6868f87c6982e8
NtReadOnlyEnlistment	0x8292fb3a	
NtDebugActiveProcess	0x828eaeda	
NtSetInformationProcess	0x82875875	682c020068d875
NtPropagationComplete	0x829316aa	
NtAlpcDisconnectPort	0x8289c577	8bff558bec515164
NtFreeUserPhysicalPages	0x8290d429	
NtWriteFile	0x828ac2b2	6a5c68b0596982e8
NtEnumerateTransactionObject	0x8293050a	
NtSetInformationToken	0x8284095f	6818010068282b
NtSetIoCompletionEx	0x828f4240	
NtSetEvent	0x82866938	6a1468685c6982e8
NtSetContextThread	0x8291a84f	8bff558bec515164
NtIsUILanguageComitted	0x827cfea4	a1208f9a82f7d81b

Figure 4: SSDT.



Callbacks

TYPE	FILEPATH	DRIVER	MODULE	ADDRESS (VA)	SHA
CreateThreadNotify	C:\Windows\System32\drivers\savonaccess.sys	\FileSystem\SAVOnAccess	\SystemRoot\system32\DRIVERS\savonaccess.sys	0x8b438ec6	3a4853ede36b809d40270ef13328806c2e21ec7

Figure 5: Callbacks.

3. RESULTS

Approach

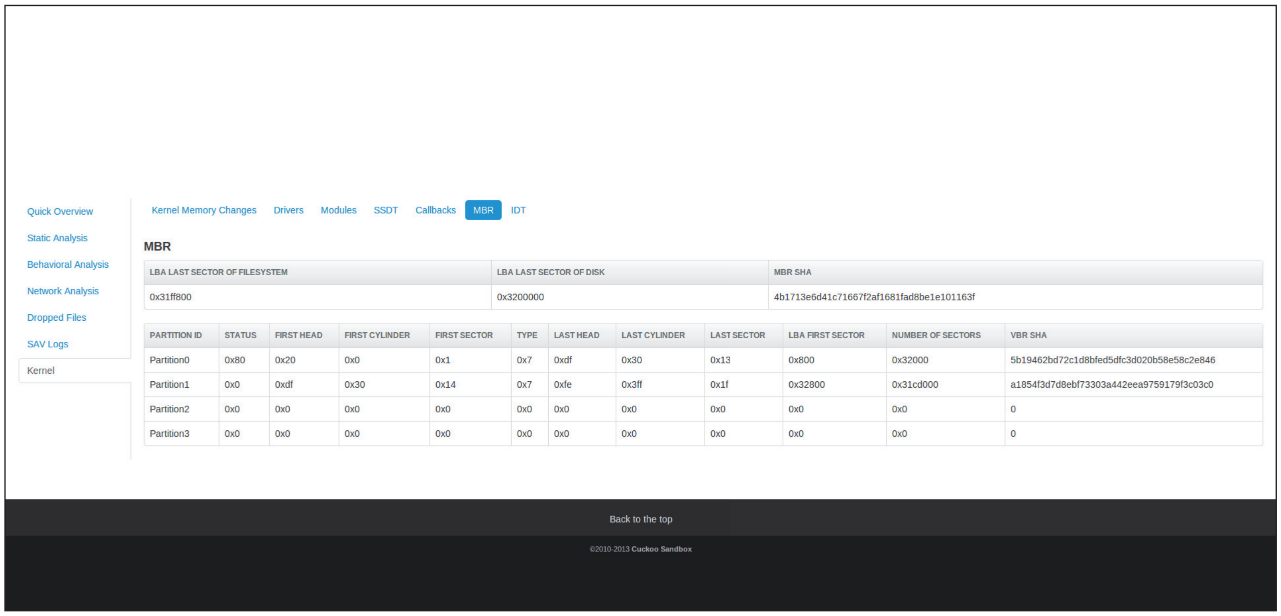
To assess the effectiveness of the system we tested three groups of samples. First, we picked a group of rootkits that were thoroughly documented and compared the results obtained from running the samples in our systems versus the behaviour documented by researchers. We also ran a set of known malicious driver files and a set of known clean driver files to analyse the results. Finally, we ran a set of known malicious standard executables and analysed the results obtained from them.

3.1 Known malicious rootkits

We picked a group of four well documented and researched rootkit samples to assess against our system. In cases where the report or blog article about the sample contained SHAs, we used those, otherwise we selected SHAs which we had previously analysed.

TDL

TDL has many variants. We tested a variant that employs an MBR infection technique to load its malicious driver. TDL4 is discussed thoroughly in [10, 11]. According to [10], on an x86

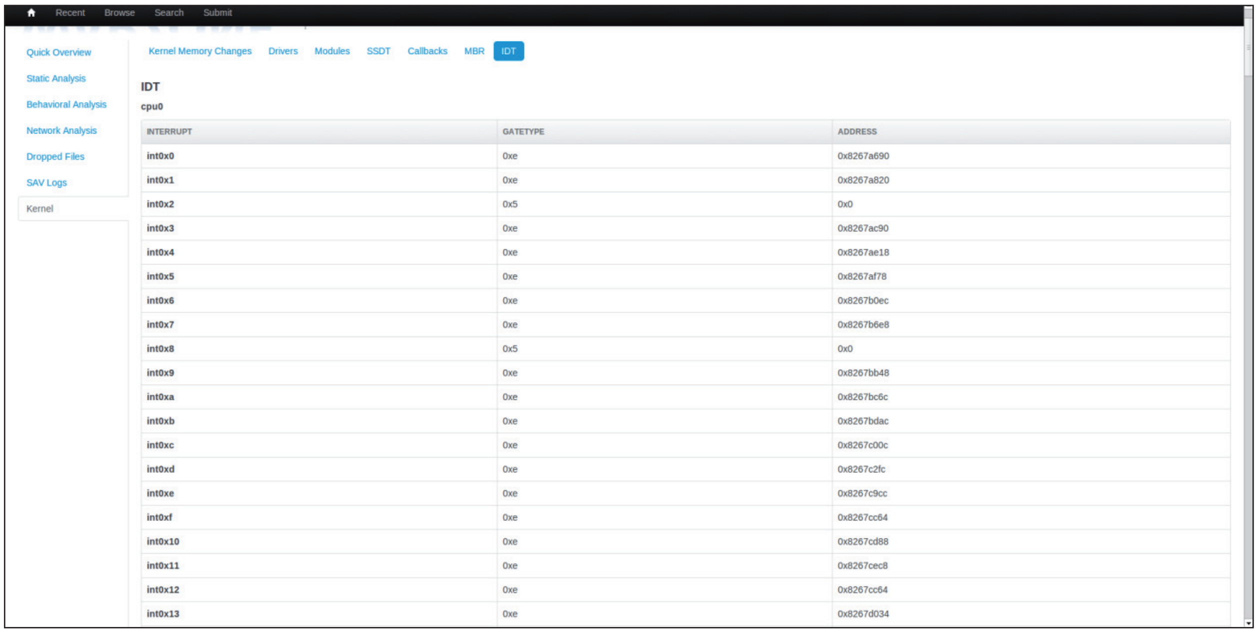


MBR

LBA LAST SECTOR OF FILESYSTEM	LBA LAST SECTOR OF DISK	MBR SHA
0x31ff800	0x3200000	4b1713e6d41c71667f2af1681fad8be1e101163f

PARTITION ID	STATUS	FIRST HEAD	FIRST CYLINDER	FIRST SECTOR	TYPE	LAST HEAD	LAST CYLINDER	LAST SECTOR	LBA FIRST SECTOR	NUMBER OF SECTORS	VBR SHA
Partition0	0x80	0x20	0x0	0x1	0x7	0xdf	0x30	0x13	0x800	0x32000	5b19462bd72c1d8bfed5dfc3d020b58e58c2e846
Partition1	0x0	0xdf	0x30	0x14	0x7	0xfe	0x3ff	0x1f	0x32800	0x31cd000	a1854f3d7d8ebf73303a442eea9759179f3c03c0
Partition2	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0
Partition3	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0

Figure 6: MBR.



IDT

cpu0

INTERRUPT	GATETYPE	ADDRESS
int0x0	0xe	0x8267a690
int0x1	0xe	0x8267a820
int0x2	0x5	0x0
int0x3	0xe	0x8267ac90
int0x4	0xe	0x8267ae18
int0x5	0xe	0x8267af78
int0x6	0xe	0x8267b0ec
int0x7	0xe	0x8267b6e8
int0x8	0x5	0x0
int0x9	0xe	0x8267bb48
int0xa	0xe	0x8267bc6c
int0xb	0xe	0x8267bdac
int0xc	0xe	0x8267c00c
int0xd	0xe	0x8267c2fc
int0xe	0xe	0x8267c9cc
int0xf	0xe	0x8267cc64
int0x10	0xe	0x8267cd88
int0x11	0xe	0x8267cec8
int0x12	0xe	0x8267cc64
int0x13	0xe	0x8267d034

Figure 7: IDT.

system, the kernel-mode drivers are tasked with maintaining the hidden file system to store the bootkit components, injecting the payload into system processes, and performing self-defence. The dropper tries to employ the print spooler provider load method to load its driver in kernel memory. The sample we looked at employed direct writes to disk by getting a handle to PhysicalDrive0 using NtCreateFile, and then used a sequence of DeviceIOControls to write to the disk, as explained in [11].

The SHA of the sample we analysed was c46ac210e06aad56dda12df3c0ccebb45108a745. Figures 8 and 9 show the changes introduced in the kernel.

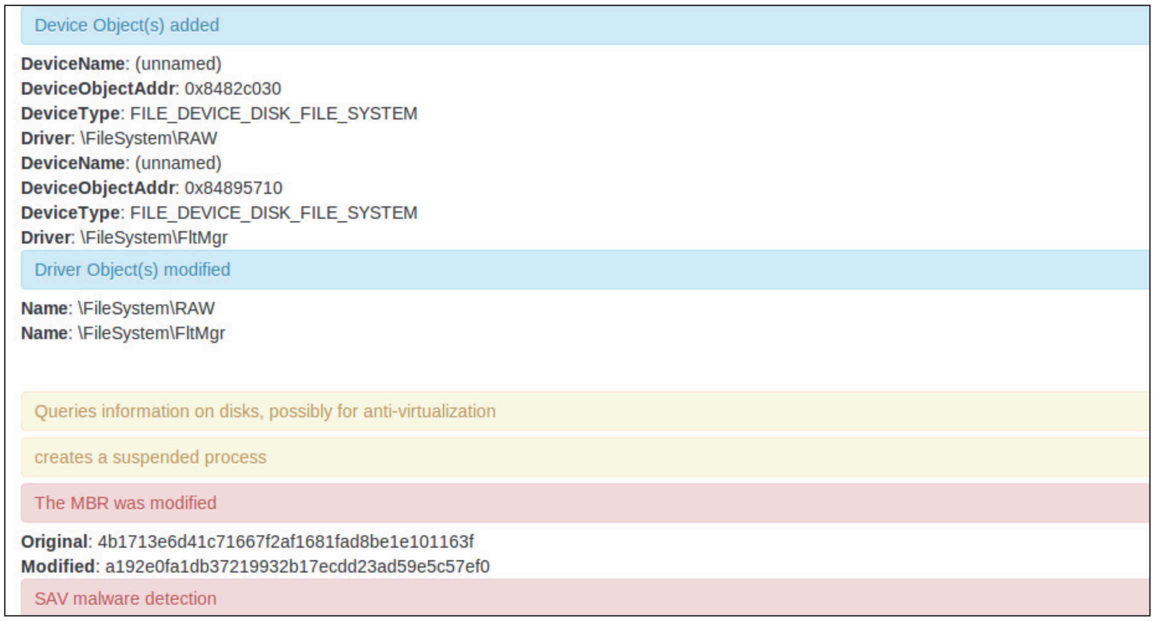
```
\Filesystem\RAW
{u'devicename':u'(unnamed)',
u'devobj_extension':
  {u'deviceobject': u'0x8482c030'},
u'driverobject': u'0x83e1f1b8',
u'deviceobject': u'0x8482c030',
u'attacheddevice':
     {u'devicename': u'(unnamed)',
     u'devobj_extension':
        {u'attachedtoname': u'(unnamed)',
        u'attachedtoobject': u'0x8482c030',
        u'deviceobject': u'0x84895710'},
     u’driverobject’: u’0x84a41c40’,
     u’deviceobject’: u’0x84895710’,
     u’drivername’: u’\\FileSystem\\FltMgr’,
     u’type’: u’FILE_DEVICE_DISK_FILE_SYSTEM’},
u’drivername’: u’\\FileSystem\\RAW’,
u’type’: u’FILE_DEVICE_DISK_FILE_SYSTEM’}}]}

\Filesystem\FltMgr
{u’devicename’: u’(unnamed)’,
u’devobj_extension’:
     {u’attachedtoname’: u’(unnamed)’,
     u’attachedtoobject’: u’0x8482c030’,
     u’deviceobject’: u’0x84895710’},
u’driverobject’: u’0x84a41c40’,
u’deviceobject’: u’0x84895710’,
u’drivername’: u’\\FileSystem\\FltMgr’,
u’type’: u’FILE_DEVICE_DISK_FILE_SYSTEM’}}]}
```

What we observe from the changes to the driver objects are the changes seen in the DEVICE_OBJECTs relating to two drivers: \FileSystem\FltMgr and \FileSystem\RAW. If we examine the changes above, we see that a single device has been created and attached to the device pointed to by \Filesystem\RAW. That same device is now pointed to by \FileSystem\FltMgr. The DEVICE_OBJECT is represented as the one at address 0x84895710.

Another change is in the MBR SHA which indicates that the sample has written the malicious MBR to disk:

```
[{u’Original’:u’4b1713e6d41c71667f2af1681fad8be1e1011
63f’},
{u’Changed’: u’a192e0fa1db37219932b17ecdd23ad59e5c57e
f0’}]
```

Finally, there are changes in the SSDT that are the result of paging in parts of the kernel. These are marked as paged in during processing of the raw results. While they might not necessarily be system calls made by the sample, they could lie within a page that was paged in and thus show up in the differencing (see Table 2).


Device Object(s) added

DeviceName: (unnamed)
DeviceObjectAddr: 0x8482c030
DeviceType: FILE_DEVICE_DISK_FILE_SYSTEM
Driver: \FileSystem\RAW
DeviceName: (unnamed)
DeviceObjectAddr: 0x84895710
DeviceType: FILE_DEVICE_DISK_FILE_SYSTEM
Driver: \FileSystem\FltMgr

Driver Object(s) modified

Name: \FileSystem\RAW
Name: \FileSystem\FltMgr

Queries information on disks, possibly for anti-virtualization

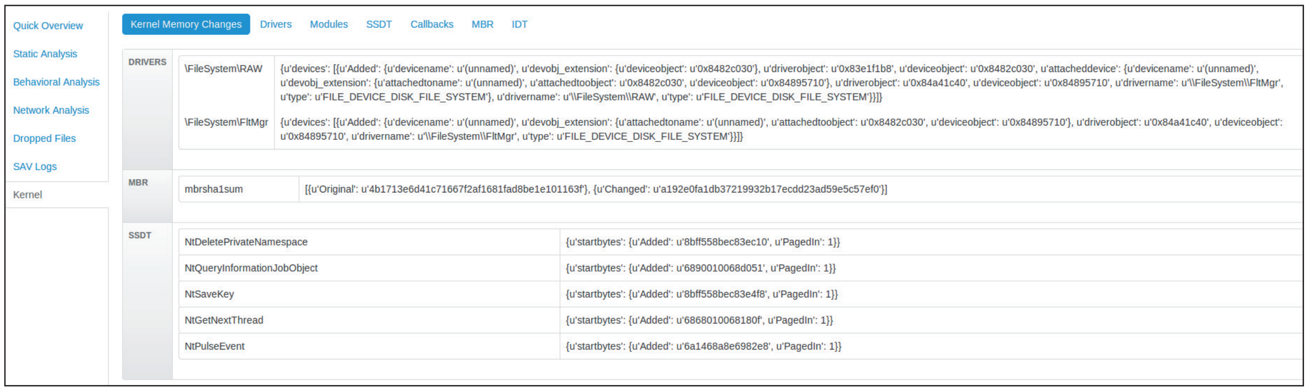
creates a suspended process

The MBR was modified

Original: 4b1713e6d41c71667f2af1681fad8be1e101163f
Modified: a192e0fa1db37219932b17ecdd23ad59e5c57ef0

SAV malware detection

Figure 8: Changes introduced in the kernel.



DRIVERS	\FileSystem\RAW	{u'devices': [{u'Added': {u'devicename': u'(unnamed)', u'devobj_extension': {u'deviceobject': u'0x8482c030'}, u'driverobject': u'0x83e1f1b8', u'deviceobject': u'0x8482c030', u'attacheddevice': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x8482c030', u'deviceobject': u'0x84895710'}, u'driverobject': u'0x84a41c40', u'deviceobject': u'0x84895710', u'drivername': u'\\FileSystem\\FltMgr', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}, u'drivername': u'\\FileSystem\\RAW', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}}]}
	\FileSystem\FltMgr	{u'devices': [{u'Added': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x8482c030', u'deviceobject': u'0x84895710'}, u'driverobject': u'0x84a41c40', u'deviceobject': u'0x84895710', u'drivername': u'\\FileSystem\\FltMgr', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}}]}
MBR	mbrsha1sum	[{u'Original': u'4b1713e6d41c71667f2af1681fad8be1e101163f'}, {u'Changed': u'a192e0fa1db37219932b17ecdd23ad59e5c57ef0'}]
SSDT	NtDeletePrivateNamespace	{u'startbytes': {u'Added': u'8bff558bec83ec10', u'PagedIn': 1}}
	NtQueryInformationJobObject	{u'startbytes': {u'Added': u'6890010068d051', u'PagedIn': 1}}
	NtSaveKey	{u'startbytes': {u'Added': u'8bff558bec83e4f8', u'PagedIn': 1}}
	NtGetNextThread	{u'startbytes': {u'Added': u'6868010068180f', u'PagedIn': 1}}
	NtPulseEvent	{u'startbytes': {u'Added': u'6a1468a8e6982e8', u'PagedIn': 1}}

Figure 9: Kernel memory changes.

```
NtDeletePrivateNamespace     {u'startbytes': {u'Added': u'8bff558bec83ec10', u'PagedIn': 1}}
NtQueryInformationJobObject  {u'startbytes': {u'Added': u'6890010068d051', u'PagedIn': 1}}
NtSaveKey                    {u'startbytes': {u'Added': u'8bff558bec83e4f8', u'PagedIn': 1}}
NtGetNextThread              {u'startbytes': {u'Added': u'6868010068180f', u'PagedIn': 1}}
NtPulseEvent                 {u'startbytes': {u'Added': u'6a1468a8e6982e8', u'PagedIn': 1}}
```

Table 2: Changes in the SSDT that are marked as paged in during processing of the raw results.

Gapz

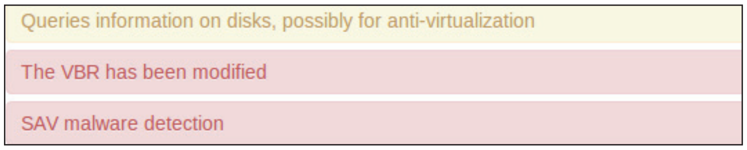
Gapz is a bootkit that is famous for its sophistication. It was thoroughly analysed in [12]. Gapz comes as two different variants: an MBR infector and a VBR infector. The MBR-infecting version of the bootkit hijacks the boot process to hook the int 13 handler. The VBR-infecting variant only alters a DWORD: the 'hidden_sectors' value in the BIOS Parameter block of the VBR, which will instruct the VBR code to transfer control to a malicious loader instead of the standard IPL. The kernel-mode changes include writing blocks of its malicious kernel module to kernel memory in addition to hooks to IRP_MJ_INTERNAL_DEVICE_CONTROL and IRP_MJ_DEVICE_CONTROL handlers of the disk miniport driver. Both variants employ the same kernel-mode component.

The SHA of the VBR infector we used was 1f206ea64fb3ccbe0cd7ff7972bef2592bb30c84. As documented in [12], this sample injects the shellcode responsible for writing to disk into explorer.exe. The sample gets a handle to the volume and to the disk by calling NtCreateFile on GLOBALROOT\ArcName\multi(0)disk(0)rdisk(0)partition(1) and PhysicalDrive0, respectively. As with TDL, it issues a series of DeviceIOControls to write to them.

As can be seen in Figures 10 and 11, the differencing identified a change in the VBR for partition 0.

```
NtDeletePrivateNamespace    {u'startbytes': {u'Added': u'8bff558bec83ec10', u'PagedIn': 1}}
NtSaveKey                   {u'startbytes': {u'Added': u'8bff558bec83e4f8', u'PagedIn': 1}}
NtPulseEvent                {u'startbytes': {u'Added': u'6a1468a8e6982e8', u'PagedIn': 1}}
```

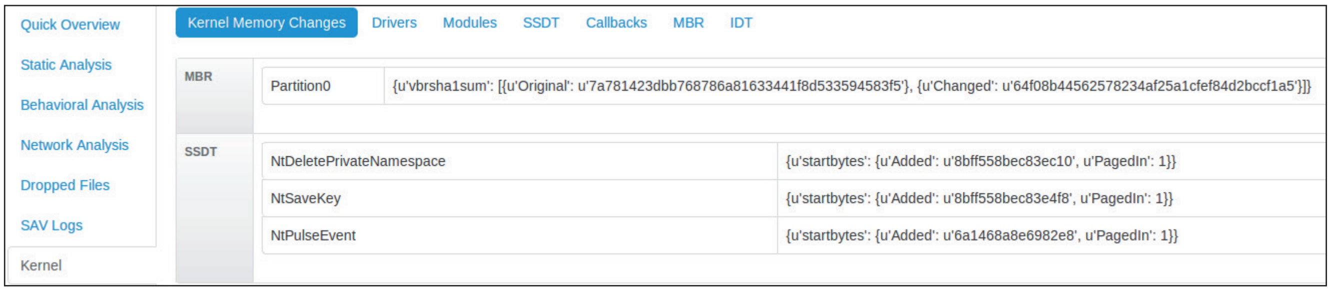
Table 3: Changes in the SSDT that are showing as paged in calls.


Queries information on disks, possibly for anti-virtualization

The VBR has been modified

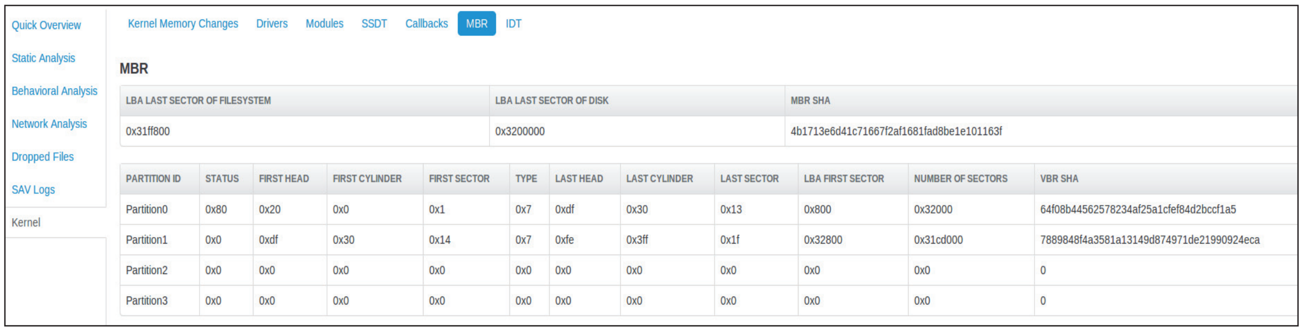
SAV malware detection

Figure 10: A change in VBR is noted.



MBR	Partition0	{u'vbrsha1sum': [{u'Original': u'7a781423dbb768786a81633441f8d533594583f5'}, {u'Changed': u'64f08b44562578234af25a1cfef84d2bccf1a5'}]}
SSDT	NtDeletePrivateNamespace	{u'startbytes': {u'Added': u'8bff558bec83ec10', u'PagedIn': 1}}
	NtSaveKey	{u'startbytes': {u'Added': u'8bff558bec83e4f8', u'PagedIn': 1}}
	NtPulseEvent	{u'startbytes': {u'Added': u'6a1468a8e6982e8', u'PagedIn': 1}}

Figure 11: A change in the VBR for partition 0 is identified.



MBR

LBA LAST SECTOR OF FILESYSTEM	LBA LAST SECTOR OF DISK	MBR SHA
0x31ff800	0x3200000	4b1713e6d41c71667f2af1681fad8be1e101163f

PARTITION ID	STATUS	FIRST HEAD	FIRST CYLINDER	FIRST SECTOR	TYPE	LAST HEAD	LAST CYLINDER	LAST SECTOR	LBA FIRST SECTOR	NUMBER OF SECTORS	VBR SHA
Partition0	0x80	0x20	0x0	0x1	0x7	0xdf	0x30	0x13	0x800	0x32000	64f08b44562578234af25a1cfef84d2bccf1a5
Partition1	0x0	0xdf	0x30	0x14	0x7	0xfe	0x3ff	0x1f	0x32800	0x31cd000	7889848f4a3581a13149d874971de21990924eca
Partition2	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0
Partition3	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0

Figure 12: MBR.

```
{u'vbrsha1sum': [{u'Original': u'7a781423dbb768786a816
33441f8d533594583f5'},
{u'Changed': u'64f08b44562578234af25a1cfef84d2bccf1a5
'}]}
```

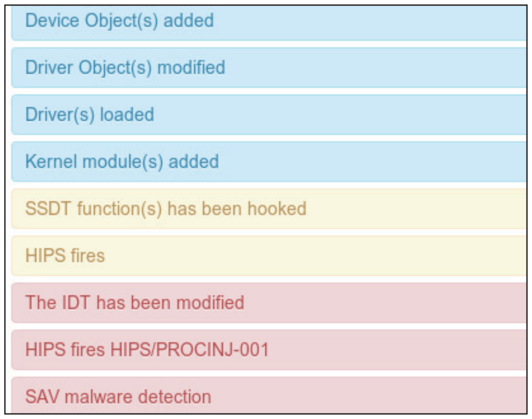
We can also see changes in the SSDT that are showing as paged in calls (see Table 3).

In our standard system set-up we did not capture the changes introduced in kernel memory during our standard test. We caught the write to the VBR, which is sufficient indication of malicious behaviour. The reason we could not see the kernel memory changes is because the dropper requests a reboot in order to kick off the VBR/MBR loading of its kernel-mode component.

Turla (a.k.a. Uroburous, Snake)

In March 2014, German security firm *G Data* released a red paper which contained an analysis of the Uroburous rootkit [13]. Shortly afterwards, *BAE Systems* also released a comprehensive white paper on the rootkit [4]. The *G Data* paper indicated that the rootkit may have been involved in a Russian cyber attack against the US. Due to tensions between Russia and Ukraine at the time, multiple news companies covered the story, including the *New York Times* [14].

As described in [4] and [13], the rootkit comprises a driver and a virtual file system that is contained in a file on the disk. The rootkit creates an entry in the Interrupt Descriptor Table (IDT).



Device Object(s) added
Driver Object(s) modified
Driver(s) loaded
Kernel module(s) added
SSDT function(s) has been hooked
HIPS fires
The IDT has been modified
HIPS fires HIPS/PROCINJ-001
SAV malware detection

Figure 13: Results of sample analysis.

The code for several functions in the SSDT is then hooked with an INT instruction (see [8] for details) that triggers the malicious software interrupt. A parameter is passed to the interrupt, identifying from which function it was called. This is used to look up a function in a custom dispatch table which executes the appropriate code for the hook.

From running an analysis on sample 39e492e839ad47ab6b5860 8f0f6b7290b52122eb, we get the results shown in Figure 13. The results tell us that SSDT functions have been hooked, and that the IDT has been modified, which is consistent with the behaviour of Turla.

Upon closer inspection of the SSDT (see Figure 14), we can see that a number of functions have been modified. Functions like NtQuerySystemInformation and NtReadFile are often targeted by rootkits.

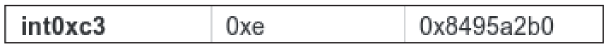
The diff is showing that there is a difference in the first eight bytes of each function. Closer examination of the NtCreateThread start bytes reveals that it is only the first five bytes that have changed.

```
{u'startbytes': [{u'Original': u'682403006820b4'},
{u'Changed': u'6a08cdc3906820b4'}]}
```

On translating the modified bytes into assembly language we get:

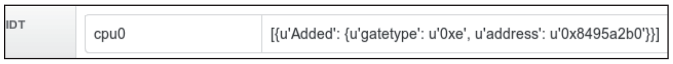
```
6A08    push byte +0x8
CDC3    nt 0xc3
90      nop
```

Looking up the IDT entry at index 0xc3 on the IDT tab (see Figure 15), we see that the code for the interrupt is located at 0x8495a2b0. Cross referencing this address with the IDT entry in the kernel diff (Figure 16), we see that 0xc3 is the newly added IDT entry.



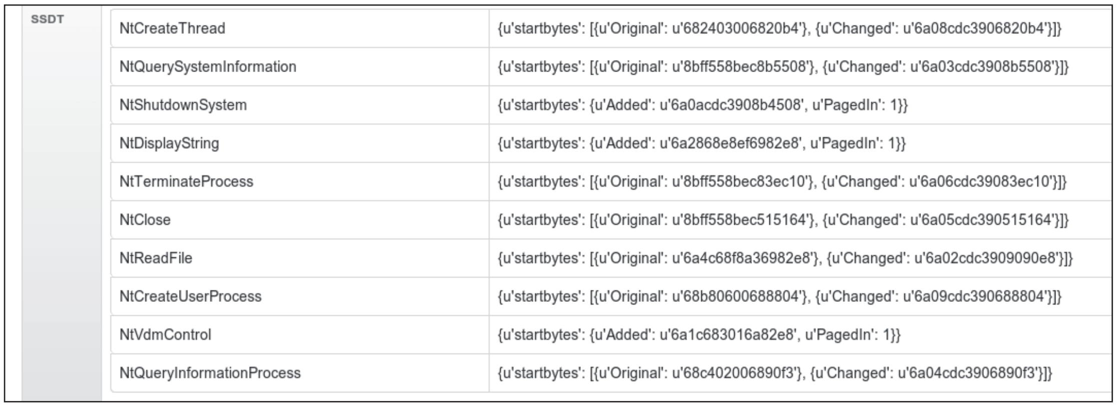
int0xc3	0xe	0x8495a2b0

Figure 15: Looking up the IDT entry at index 0xc3 on the IDT tab.



IDT	cpu0	[{u'Added': {u'gatetype': u'0xe', u'address': u'0x8495a2b0'}}]

Figure 16: Cross referencing 0x8495a2b0 with the IDT entry in the kernel diff.



SSDT	NtCreateThread	{u'startbytes': [{u'Original': u'682403006820b4'}, {u'Changed': u'6a08cdc3906820b4'}]}
	NtQuerySystemInformation	{u'startbytes': [{u'Original': u'8bff558bec8b5508'}, {u'Changed': u'6a03cdc3908b5508'}]}
	NtShutdownSystem	{u'startbytes': {u'Added': u'6a0acdc3908b4508', u'PagedIn': 1}}
	NtDisplayString	{u'startbytes': {u'Added': u'6a2868e8ef6982e8', u'PagedIn': 1}}
	NtTerminateProcess	{u'startbytes': [{u'Original': u'8bff558bec83ec10'}, {u'Changed': u'6a06cdc39083ec10'}]}
	NtClose	{u'startbytes': [{u'Original': u'8bff558bec515164'}, {u'Changed': u'6a05cdc390515164'}]}
	NtReadFile	{u'startbytes': [{u'Original': u'6a4c68f8a36982e8'}, {u'Changed': u'6a02cdc3909090e8'}]}
	NtCreateUserProcess	{u'startbytes': [{u'Original': u'68b80600688804'}, {u'Changed': u'6a09cdc390688804'}]}
	NtVdmControl	{u'startbytes': {u'Added': u'6a1c683016a82e8', u'PagedIn': 1}}
	NtQueryInformationProcess	{u'startbytes': [{u'Original': u'68c402006890f3'}, {u'Changed': u'6a04cdc3906890f3'}]}

Figure 14: A number of functions have been modified.

Necurs

The Necurs rootkit was first discovered in May 2011 [15], but became well established in 2012. It was one of the first widespread rootkits to target x64 systems. It was originally packaged with fake anti-virus malware, but recently it has been used with the infamous Gameover Zeus variant [16]. Although it has been around for a while, the rootkit is as prevalent as ever.

Necurs is the name for the kernel-mode part of the rootkit. It consists of a driver file which, when installed on a system, works alongside other malware to hide its presence and make it difficult to remove. The way this is achieved differs depending on the processor architecture and operating system of the host computer. As our analysis system is only running on x86 *Windows 7*, we will only discuss the effects of the rootkit on this particular set-up.

When Necurs is installed on *Windows 7*, a service is created and the malicious driver is loaded. The driver hides the service and the driver file on disk by installing a file system filter driver. It also installs a LoadImageNotify callback to control which programs can be loaded.

By looking at the signatures (Figures 17 and 18) after running a Necurs sample (412769f14bbc629a16208cda5fd6dc25e5f8e8bb), we see that a number of kernel modifications have occurred. These indicate that a new driver and a new callback have been installed.

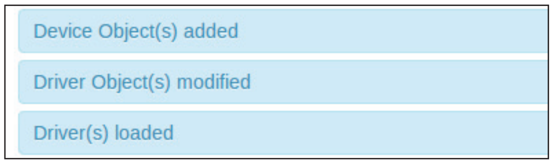

Device Object(s) added
Driver Object(s) modified
Driver(s) loaded

Figure 17: Signatures after running a Necurs sample.

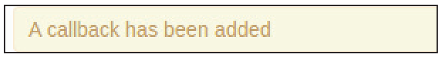
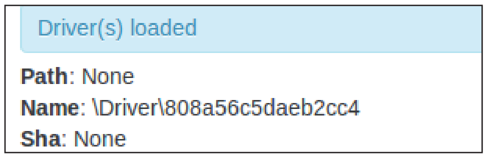

A callback has been added

Figure 18: Signatures after running a Necurs sample.

Closer examination of the driver object (Figure 19) shows a name but no path. This means that our system could not access the driver. The most likely reason for this is that the driver is hidden. Something is hiding it from the file system, yet the driver object exists in memory. While this is not necessarily a malicious trait, it is certainly suspicious, and is worth investigating further.


Driver(s) loaded

Path: None
Name: \Driver\808a56c5daeb2cc4
Sha: None

Figure 19: Closer examination of the driver object.

The next point of interest is the addition of device objects. By looking at the kernel diff (Figure 20) we see that a device has been added to /FileSystem/FtlMgr, which is the file system filter manager. This is definitely of interest. Examination of the added data shows that the driver object related to the added device belongs to the newly added driver.

Similarly, examining the callback that has been added (Figure 21) also reveals that the code for the callback is contained in the newly added driver.

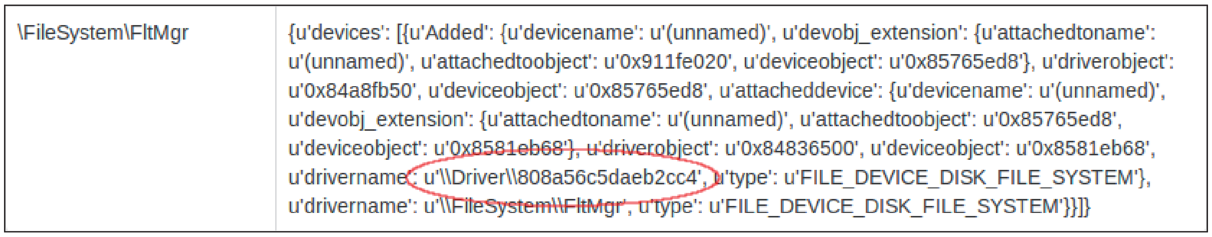
To summarize, the scan has revealed:

- A new driver that is hidden
- A file system filter
- A LoadImageNotify callback

This collection of traits is indicative of rootkit-like behaviour, and is consistent with the behaviour of the Necurs rootkit.

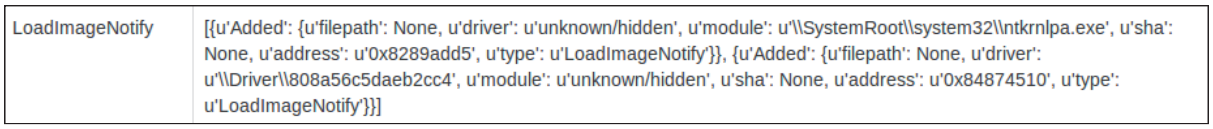
3.2 Building a malware index using kernel memory differencing

To assess whether kernel memory differencing can be used to build a metric to decide if a sample is malicious, we performed two experiments. In the first experiment, we ran a group of malicious and clean 32-bit driver files and observed the changes in the kernel introduced by running the drivers. In the second experiment we ran a set of malicious standard executable files to find out what proportion of them introduced kernel memory changes.



\FileSystem\FltMgr	{u'devices': [{u'Added': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x911fe020', u'deviceobject': u'0x85765ed8'}, u'driverobject': u'0x84a8fb50', u'deviceobject': u'0x85765ed8', u'attacheddevice': {u'devicename': u'(unnamed)', u'devobj_extension': {u'attachedtoname': u'(unnamed)', u'attachedtoobject': u'0x85765ed8', u'deviceobject': u'0x8581eb68'}, u'driverobject': u'0x84836500', u'deviceobject': u'0x8581eb68', u'drivername': u'\\Driver\\808a56c5daeb2cc4', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}, u'drivername': u'\\FileSystem\\FltMgr', u'type': u'FILE_DEVICE_DISK_FILE_SYSTEM'}}]}

Figure 20: A device has been added to /FileSystem/FtlMgr.



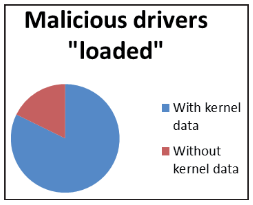
LoadImageNotify	[{u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'\\SystemRoot\\system32\\ntkrnlpa.exe', u'sha': None, u'address': u'0x8289add5', u'type': u'LoadImageNotify'}}, {u'Added': {u'filepath': None, u'driver': u'\\Driver\\808a56c5daeb2cc4', u'module': u'unknown/hidden', u'sha': None, u'address': u'0x84874510', u'type': u'LoadImageNotify'}}]

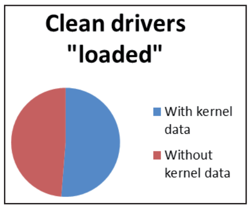
Figure 21: Examining the callback.

Running a set of driver files

We observed from our first experiment that not all successfully loaded drivers produced kernel memory data. From the set of successfully loaded malicious drivers, 82.3% produced kernel memory data that could be used later for differencing. From the clean set, 51.2% returned kernel memory data.

The reasons for failure were shared by clean and malicious drivers. A common reason was insufficient time for the log to be generated. Because the scan starts 105 seconds into the 120 seconds available for the analysis, and can take between 10 and 15 seconds, we sometimes miss the log by a second or two. Since the agent terminates before the machine shuts down fully, this is time taken from the 120 seconds. There is a trade-off between speed of analysis and data gathered. Our testing produced enough successful replication from malicious files for an automated analysis system.





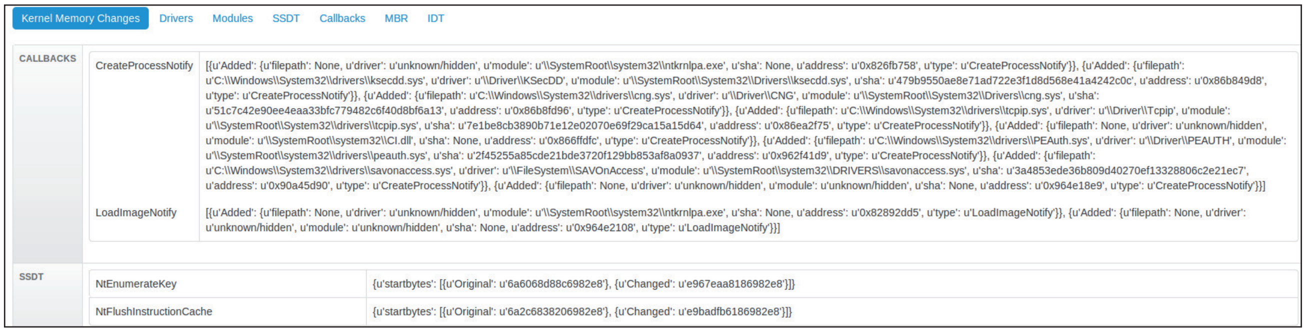
From a set of malicious driver files:

Total drivers successfully loaded: 1,854

Total tasks with kernel data retrieved: 1,525

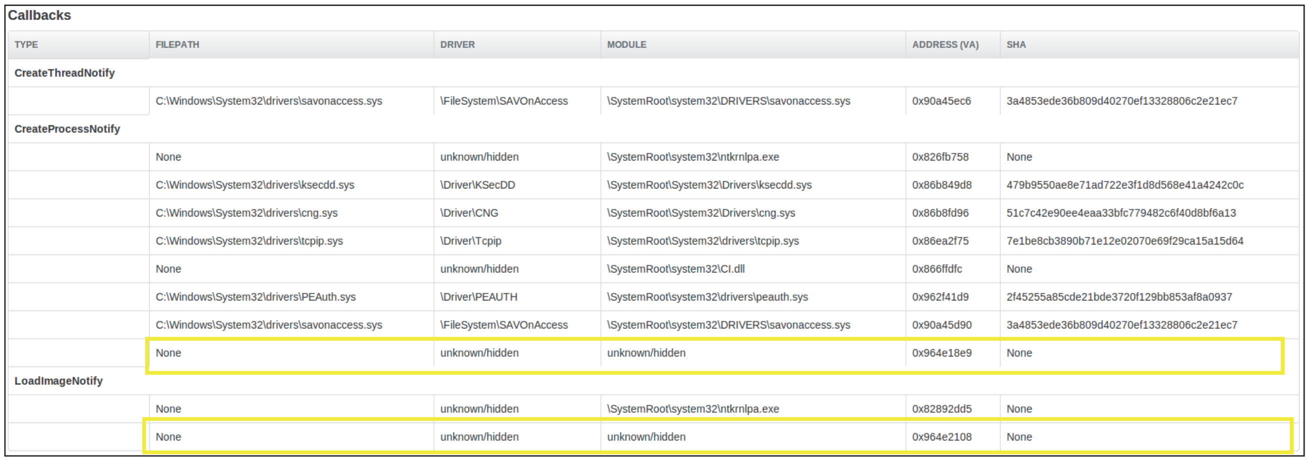
Signature	Number of hits	Percentage
generic_new_driver	1524	99.9
generic_modified_driver	1171	76.8
generic_deleted_driver	0	0
generic_new_module	1523	99.8
generic_deleted_module	0	0
generic_modified_module	0	0
generic_ssdt_hook	30	1.96
generic_idt_hook	0	0
generic_new_callback	1332	87.3
generic_modified_callback	0	0
generic_attached_device	1149	75.3
generic_new_device_links	1441	94.5
generic_modified_mbr	0	0
generic_modified_vbr	0	0
generic_modified_eod_size	0	0

We expect that a successfully loaded driver would create a driver object and a new module. All of the malicious drivers created a new driver object on loading, except for one. This driver installed 'CreateProcessNotify' and 'LoadImageNotify' callbacks pointing to an 'unknown/module', and installed inline hooks at 'NTEnumerateKey' and 'NtFlushInstructionCache'. This is indicative of highly suspicious behaviour.



CALLBACKS	CreateProcessNotify	[{u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'\\SystemRoot\\system32\\ntkrnlpa.exe', u'sha': None, u'address': u'0x826fb758', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': u'C:\\Windows\\System32\\drivers\\ksecdd.sys', u'driver': u'\\Driver\\KSecDD', u'module': u'\\SystemRoot\\System32\\Drivers\\ksecdd.sys', u'sha': u'479b9550ae8e71ad722e3f1d8d568e41a4242c0c', u'address': u'0x86b849d8', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': u'C:\\Windows\\System32\\drivers\\cng.sys', u'driver': u'\\Driver\\CNG', u'module': u'\\SystemRoot\\System32\\Drivers\\cng.sys', u'sha': u'51c7c42e90ee4eaa33bfc779482c6f40d8bf6a13', u'address': u'0x86b8fd96', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': u'C:\\Windows\\System32\\drivers\\tcpip.sys', u'driver': u'\\Driver\\Tcpip', u'module': u'\\SystemRoot\\System32\\drivers\\tcpip.sys', u'sha': u'7e1be8cb3890b71e12e02070e69f29ca15a15d64', u'address': u'0x86ea2f75', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'\\SystemRoot\\system32\\CI.dll', u'sha': None, u'address': u'0x866ffdfc', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': u'C:\\Windows\\System32\\drivers\\PEAuth.sys', u'driver': u'\\Driver\\PEAUTH', u'module': u'\\SystemRoot\\system32\\drivers\\peauth.sys', u'sha': u'2f45255a85cde21bde3720f129bb853af8a0937', u'address': u'0x962f41d9', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': u'C:\\Windows\\System32\\drivers\\savonaccess.sys', u'driver': u'\\FileSystem\\SAVOnAccess', u'module': u'\\SystemRoot\\system32\\DRIVERS\\savonaccess.sys', u'sha': u'3a4853ede36b809d40270ef13328806c2e21ec7', u'address': u'0x90a45d90', u'type': u'CreateProcessNotify'}}, {u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'unknown/hidden', u'sha': None, u'address': u'0x964e18e9', u'type': u'CreateProcessNotify'}}]
	LoadImageNotify	[{u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'\\SystemRoot\\system32\\ntkrnlpa.exe', u'sha': None, u'address': u'0x82892dd5', u'type': u'LoadImageNotify'}}, {u'Added': {u'filepath': None, u'driver': u'unknown/hidden', u'module': u'unknown/hidden', u'sha': None, u'address': u'0x964e2108', u'type': u'LoadImageNotify'}}]
SSDT	NtEnumerateKey	{u'startbytes': [{u'Original': u'6a6068d88c6982e8'}, {u'Changed': u'e967eaa8186982e8'}]}
	NtFlushInstructionCache	{u'startbytes': [{u'Original': u'6a2c6838206982e8'}, {u'Changed': u'e9badfb6186982e8'}]}

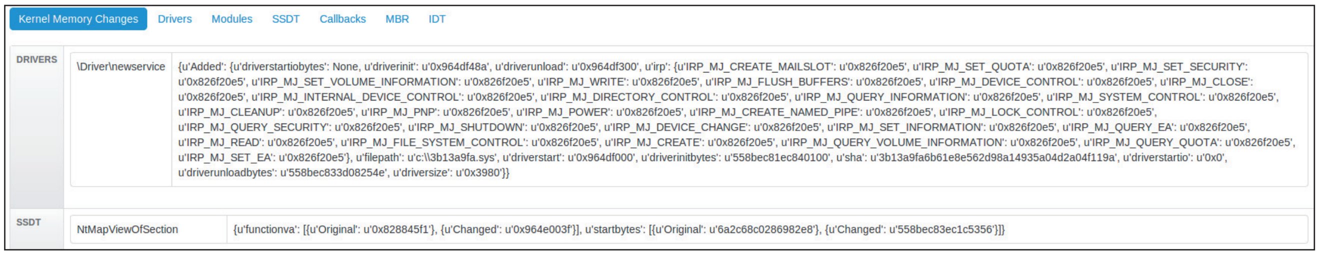
Figure 22: The driver installed 'CreateProcessNotify' and 'LoadImageNotify' callbacks.



Callbacks

TYPE	FILEPATH	DRIVER	MODULE	ADDRESS (VA)	SHA
CreateThreadNotify					
	C:\Windows\System32\drivers\savonaccess.sys	\FileSystem\SAVOnAccess	\SystemRoot\system32\DRIVERS\savonaccess.sys	0x90a45ec6	3a4853ede36b809d40270ef13328806c2e21ec7
CreateProcessNotify					
	None	unknown/hidden	\SystemRoot\system32\ntkrnlpa.exe	0x826fb758	None
	C:\Windows\System32\drivers\ksecdd.sys	\Driver\KSecDD	\SystemRoot\System32\Drivers\ksecdd.sys	0x86b849d8	479b9550ae8e71ad722e3f1d8d568e41a4242c0c
	C:\Windows\System32\drivers\cng.sys	\Driver\CNG	\SystemRoot\System32\Drivers\cng.sys	0x86b8fd96	51c7c42e90ee4eaa33bfc779482c6f40d8bf6a13
	C:\Windows\System32\drivers\tcpip.sys	\Driver\Tcpip	\SystemRoot\System32\drivers\tcpip.sys	0x86ea2f75	7e1be8cb3890b71e12e02070e69f29ca15a15d64
	None	unknown/hidden	\SystemRoot\system32\CI.dll	0x866ffdfc	None
	C:\Windows\System32\drivers\PEAuth.sys	\Driver\PEAUTH	\SystemRoot\system32\drivers\peauth.sys	0x962f41d9	2f45255a85cde21bde3720f129bb853af8a0937
	C:\Windows\System32\drivers\savonaccess.sys	\FileSystem\SAVOnAccess	\SystemRoot\system32\DRIVERS\savonaccess.sys	0x90a45d90	3a4853ede36b809d40270ef13328806c2e21ec7
	None	unknown/hidden	unknown/hidden	0x964e18e9	None
LoadImageNotify					
	None	unknown/hidden	\SystemRoot\system32\ntkrnlpa.exe	0x82892dd5	None
	None	unknown/hidden	unknown/hidden	0x964e2108	None

Figure 23: Callbacks pointing to an 'unknown/module'.



```
DRIVERS  \Driver\newservice  {u'Added': {u'driverstartiobytes': None, u'driverinit': u'0x964df48a', u'driverunload': u'0x964df300', u'irp': {u'IRP_MJ_CREATE_MAILSLOT': u'0x826f20e5', u'IRP_MJ_SET_QUOTA': u'0x826f20e5', u'IRP_MJ_SET_SECURITY':
                             u'0x826f20e5', u'IRP_MJ_SET_VOLUME_INFORMATION': u'0x826f20e5', u'IRP_MJ_WRITE': u'0x826f20e5', u'IRP_MJ_FLUSH_BUFFERS': u'0x826f20e5', u'IRP_MJ_DEVICE_CONTROL': u'0x826f20e5', u'IRP_MJ_CLOSE':
                             u'0x826f20e5', u'IRP_MJ_INTERNAL_DEVICE_CONTROL': u'0x826f20e5', u'IRP_MJ_DIRECTORY_CONTROL': u'0x826f20e5', u'IRP_MJ_QUERY_INFORMATION': u'0x826f20e5', u'IRP_MJ_SYSTEM_CONTROL': u'0x826f20e5',
                             u'IRP_MJ_CLEANUP': u'0x826f20e5', u'IRP_MJ_PNP': u'0x826f20e5', u'IRP_MJ_POWER': u'0x826f20e5', u'IRP_MJ_CREATE_NAMED_PIPE': u'0x826f20e5', u'IRP_MJ_LOCK_CONTROL': u'0x826f20e5',
                             u'IRP_MJ_QUERY_SECURITY': u'0x826f20e5', u'IRP_MJ_SHUTDOWN': u'0x826f20e5', u'IRP_MJ_DEVICE_CHANGE': u'0x826f20e5', u'IRP_MJ_SET_INFORMATION': u'0x826f20e5', u'IRP_MJ_QUERY_EA': u'0x826f20e5',
                             u'IRP_MJ_READ': u'0x826f20e5', u'IRP_MJ_FILE_SYSTEM_CONTROL': u'0x826f20e5', u'IRP_MJ_CREATE': u'0x826f20e5', u'IRP_MJ_QUERY_VOLUME_INFORMATION': u'0x826f20e5', u'IRP_MJ_QUERY_QUOTA': u'0x826f20e5',
                             u'IRP_MJ_SET_EA': u'0x826f20e5'}, u'filepath': u'c:\\3b13a9fa.sys', u'driverstart': u'0x964df000', u'driverinitbytes': u'558bec81ec840100', u'sha': u'3b13a9fa6b61e8e562d98a14935a04d2a04f119a', u'driverstartio': u'0x0',
                             u'driverunloadbytes': u'558bec833d08254e', u'driversize': u'0x3980'}}

SSDT     NtMapViewOfSection  {u'functionva': [{u'Original': u'0x828845f1'}, {u'Changed': u'0x964e003f'}], u'startbytes': [{u'Original': u'6a2c68c0286982e8'}, {u'Changed': u'558bec83ec1c5356'}]}
```

Figure 24: The sample installs an SSDT hook by altering the VA of the service function 'NtMapViewOfSection'.

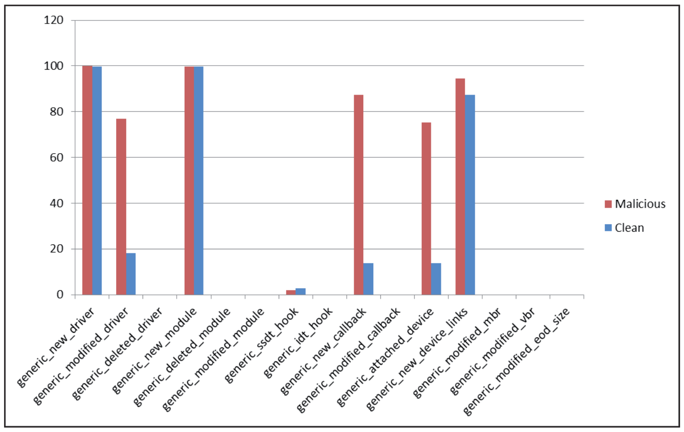
Another interesting find was a sample that created a new driver object upon loading, but which did not show a newly added module in the differencing data. Looking at the job we find that the sample installs an SSDT hook by altering the VA of the service function 'NtMapViewOfSection'. Again, this is highly suspicious behaviour.

From a set of running clean driver files:

Total drivers successfully loaded: 1,053

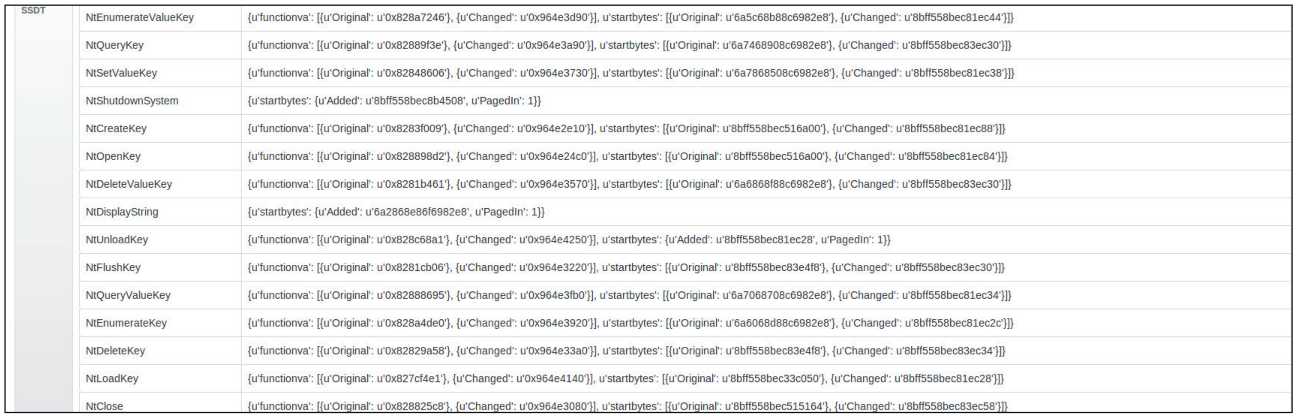
Total tasks with kernel data retrieved: 5,40

Signature	Number of hits	Percentage
generic_new_driver	539	99.8
generic_modified_driver	98	18.15
generic_deleted_driver	0	0
generic_new_module	539	99.8
generic_deleted_module	0	0
generic_modified_module	0	0
generic_ssdt_hook	15	2.77
generic_idt_hook	1	0.18
generic_new_callback	45	8.33
generic_modified_callback	0	0
generic_attached_device	74	13.70
generic_new_device_links	472	87.40
generic_modified_mbr	0	0
generic_modified_vbr	0	0
generic_modified_eod_size	0	0



These results show that the number of hits on newly created drivers corresponds to the hits on the new module. This makes sense, since a successfully loaded driver would also have added a module. An interesting finding was that on the *Windows 7* machine, the percentage of samples that installed SSDT hooks was higher. Delving into those clean samples showed that they were either security software drivers that installed hooks in the SSDT or ones used for general system administration. For example, in Figure 25 we can see the SSDT hooks installed by regmon.sys from *Microworld Technologies Inc*.

Another interesting sample was a clean driver that installed an IDT hook. Checking the results for that job shows that the driver also hooks NtOpenKey and NtCreateKey. Furthermore, it modifies the disk driver object by hooking an IRP, namely IRP_MJ_POWER. While it looks suspicious, this is actually a driver from *Aladdin Knowledge Systems* called 'aksfridge.sys'



```
SSDT  NtEnumerateValueKey  {u'functionva': [{u'Original': u'0x828a7246'}, {u'Changed': u'0x964e3d90'}], u'startbytes': [{u'Original': u'6a5c68b88c6982e8'}, {u'Changed': u'8bff558bec81ec44'}]}
      NtQueryKey           {u'functionva': [{u'Original': u'0x82889f3e'}, {u'Changed': u'0x964e3a90'}], u'startbytes': [{u'Original': u'6a7468908c6982e8'}, {u'Changed': u'8bff558bec83ec30'}]}
      NtSetValueKey        {u'functionva': [{u'Original': u'0x82848606'}, {u'Changed': u'0x964e3730'}], u'startbytes': [{u'Original': u'6a7868508c6982e8'}, {u'Changed': u'8bff558bec81ec38'}]}
      NtShutdownSystem     {u'startbytes': {u'Added': u'8bff558bec8b4508', u'PagedIn': 1}}
      NtCreateKey          {u'functionva': [{u'Original': u'0x8283f009'}, {u'Changed': u'0x964e2e10'}], u'startbytes': [{u'Original': u'8bff558bec516a00'}, {u'Changed': u'8bff558bec81ec88'}]}
      NtOpenKey            {u'functionva': [{u'Original': u'0x828898d2'}, {u'Changed': u'0x964e24c0'}], u'startbytes': [{u'Original': u'8bff558bec516a00'}, {u'Changed': u'8bff558bec81ec84'}]}
      NtDeleteValueKey     {u'functionva': [{u'Original': u'0x8281b461'}, {u'Changed': u'0x964e3570'}], u'startbytes': [{u'Original': u'6a6868f88c6982e8'}, {u'Changed': u'8bff558bec83ec30'}]}
      NtDisplayString      {u'startbytes': {u'Added': u'6a2868e86f6982e8', u'PagedIn': 1}}
      NtUnloadKey          {u'functionva': [{u'Original': u'0x828c68a1'}, {u'Changed': u'0x964e4250'}], u'startbytes': {u'Added': u'8bff558bec81ec28', u'PagedIn': 1}}
      NtFlushKey           {u'functionva': [{u'Original': u'0x8281cb06'}, {u'Changed': u'0x964e3220'}], u'startbytes': [{u'Original': u'8bff558bec83e4f8'}, {u'Changed': u'8bff558bec83ec30'}]}
      NtQueryValueKey      {u'functionva': [{u'Original': u'0x82888695'}, {u'Changed': u'0x964e3fb0'}], u'startbytes': [{u'Original': u'6a7068708c6982e8'}, {u'Changed': u'8bff558bec81ec34'}]}
      NtEnumerateKey       {u'functionva': [{u'Original': u'0x828a4de0'}, {u'Changed': u'0x964e3920'}], u'startbytes': [{u'Original': u'6a6068d88c6982e8'}, {u'Changed': u'8bff558bec81ec2c'}]}
      NtDeleteKey          {u'functionva': [{u'Original': u'0x82829a58'}, {u'Changed': u'0x964e33a0'}], u'startbytes': [{u'Original': u'8bff558bec83e4f8'}, {u'Changed': u'8bff558bec83ec34'}]}
      NtLoadKey            {u'functionva': [{u'Original': u'0x827cf4e1'}, {u'Changed': u'0x964e4140'}], u'startbytes': [{u'Original': u'8bff558bec33c050'}, {u'Changed': u'8bff558bec81ec28'}]}
      NtClose              {u'functionva': [{u'Original': u'0x828825c8'}, {u'Changed': u'0x964e3080'}], u'startbytes': [{u'Original': u'8bff558bec515164'}, {u'Changed': u'8bff558bec83ec58'}]}
```

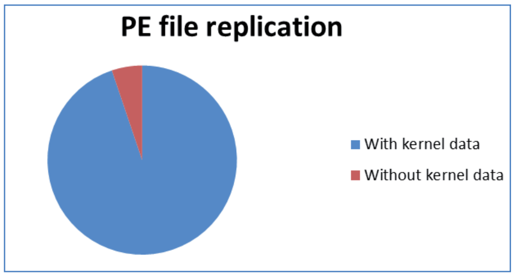
Figure 25: SSDT hooks installed by regmon.sys from Microworld Technologies Inc.

of ancillary function driver type. We are still investigating the function of this driver.

PE results

We collected a set of malicious PE files (detected by a static detection) and ran them for analysis to see whether our system can identify rootkit behaviour from executables.

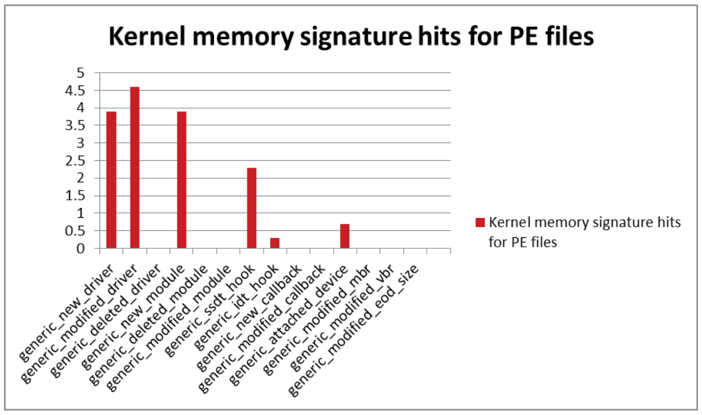
First, we observed that the percentage of jobs that produced kernel memory data was much higher than any of the driver replication sets. This is positive news for us, since PE files constitute a large proportion of malicious files.



Number of malicious PEs: 319

Number of malicious PEs with kernel memory log: 303

Signature	Number of hits	Percentage
generic_new_driver	12	3.9
generic_modified_driver	14	4.6
generic_deleted_driver	0	0
generic_new_module	12	3.9
generic_deleted_module	0	0
generic_modified_module	0	0
generic_ssdt_hook	7	2.3
generic_idt_hook	1	0.3
generic_new_callback	0	0
generic_modified_callback	0	0
generic_attached_device	2	0.7
generic_modified_mbr	0	0
generic_modified_vbr	0	0
generic_modified_eod_size	0	0



It is clear that the number of malicious PEs that induced rootkit-like behaviour was not large. We can still infer useful information by delving into the samples that triggered some of those signatures.

We can see an example of a sample that created two new driver objects in addition to installing an SSDT hook (Figures 26 and 27). This sample is detected as Troj/NtRootK-DJ. The sample drops two driver files in %SYSTEM32%\drivers\. We could see the two objects for the two drivers created in addition to the SSDT hook installed below in the kernel memory differencing. By comparing the VAs, we see that the SSDT hook lies in the second driver.

Figure 28 is a screen capture of the kernel memory differencing produced by a sample of Mal/DownLdr-BZ. The sample drops a malicious driver file in several locations. The dropped driver file is detected as Mal/Rootkit-X. We can see that in the kernel memory differencing there is a newly attached device object to the PnP Manager driver. This device object is of type FILE_DEVICE_CONTROLLER. This could indicate that the sample is trying to hide the existence of malicious files. The type of the device in addition to the location to which it is attached is sufficiently interesting to warrant further investigation.

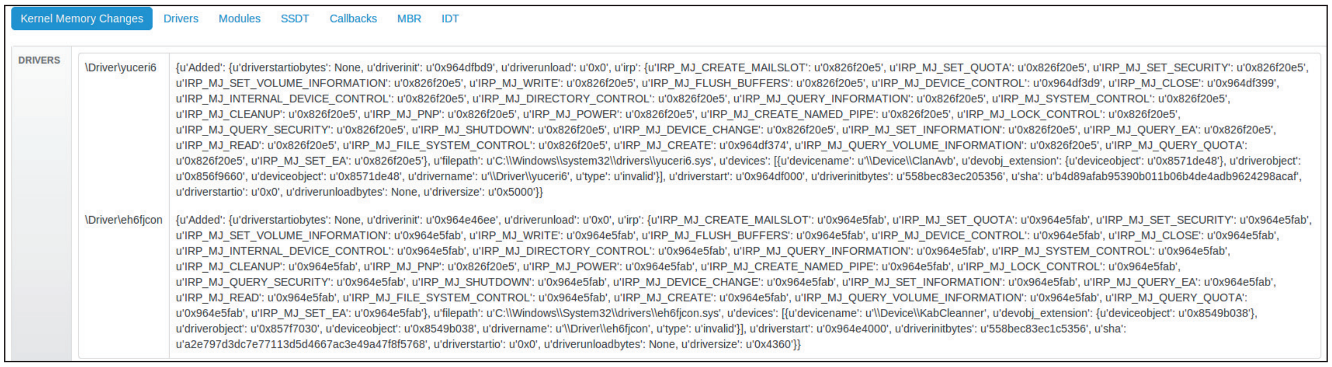
Ranking severity

Combining the data for all the malicious files versus all the clean files, we get a clearer picture of the high-contributing signatures. It is important to note that our set of malicious files is larger than the set of the clean files.

Total number of malicious files with kernel data: 1,828

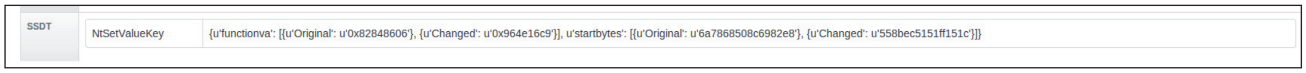
Total number of clean files with kernel data: 540

Signature	Malicious	Clean	Total hits
generic_new_driver	1536	539	2075
generic_modified_driver	1185	98	1283
generic_deleted_driver	0	0	0
generic_new_module	1535	539	2074
generic_deleted_module	0	0	0
generic_modified_module	0	0	0
generic_ssdt_hook	37	15	52
generic_idt_hook	1	1	2
generic_new_callback	1332	45	1377
generic_modified_callback	0	0	0
generic_attached_device	1151	74	1225
generic_modified_mbr	0	0	0
generic_modified_vbr	0	0	0
generic_modified_eod_size	0	0	0



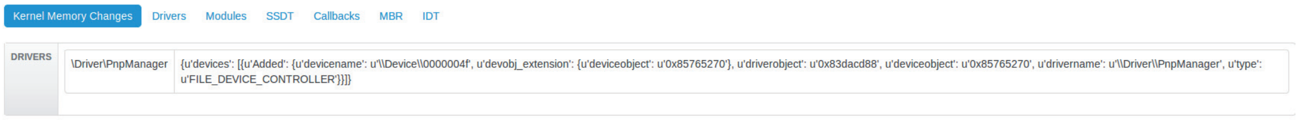
```
\Driver\yuceri6    {u'Added': {u'driverstartiobytes': None, u'driverinit': u'0x964dfbd9', u'driverunload': u'0x0', u'irp': {u'IRP_MJ_CREATE_MAILSLOT': u'0x826f20e5', u'IRP_MJ_SET_QUOTA': u'0x826f20e5', u'IRP_MJ_SET_SECURITY': u'0x826f20e5',
                   u'IRP_MJ_SET_VOLUME_INFORMATION': u'0x826f20e5', u'IRP_MJ_WRITE': u'0x826f20e5', u'IRP_MJ_FLUSH_BUFFERS': u'0x826f20e5', u'IRP_MJ_DEVICE_CONTROL': u'0x964df3d9', u'IRP_MJ_CLOSE': u'0x964df399',
                   u'IRP_MJ_INTERNAL_DEVICE_CONTROL': u'0x826f20e5', u'IRP_MJ_DIRECTORY_CONTROL': u'0x826f20e5', u'IRP_MJ_QUERY_INFORMATION': u'0x826f20e5', u'IRP_MJ_SYSTEM_CONTROL': u'0x826f20e5',
                   u'IRP_MJ_CLEANUP': u'0x826f20e5', u'IRP_MJ_PNP': u'0x826f20e5', u'IRP_MJ_POWER': u'0x826f20e5', u'IRP_MJ_CREATE_NAMED_PIPE': u'0x826f20e5', u'IRP_MJ_LOCK_CONTROL': u'0x826f20e5',
                   u'IRP_MJ_QUERY_SECURITY': u'0x826f20e5', u'IRP_MJ_SHUTDOWN': u'0x826f20e5', u'IRP_MJ_DEVICE_CHANGE': u'0x826f20e5', u'IRP_MJ_SET_INFORMATION': u'0x826f20e5', u'IRP_MJ_QUERY_EA': u'0x826f20e5',
                   u'IRP_MJ_READ': u'0x826f20e5', u'IRP_MJ_FILE_SYSTEM_CONTROL': u'0x826f20e5', u'IRP_MJ_CREATE': u'0x964df374', u'IRP_MJ_QUERY_VOLUME_INFORMATION': u'0x826f20e5', u'IRP_MJ_QUERY_QUOTA':
                   u'0x826f20e5', u'IRP_MJ_SET_EA': u'0x826f20e5'}, u'filepath': u'C:\\Windows\\system32\\drivers\\yuceri6.sys', u'devices': [{u'devicename': u'\\Device\\ClanAvb', u'devobj_extension': {u'deviceobject': u'0x8571de48'}, u'driverobject':
                   u'0x856f9660', u'deviceobject': u'0x8571de48', u'drivername': u'\\Driver\\yuceri6', u'type': u'invalid'}], u'driverstart': u'0x964df000', u'driverinitbytes': u'558bec83ec205356', u'sha': u'b4d89afab95390b011b06b4de4adb9624298acaf',
                   u'driverstartio': u'0x0', u'driverunloadbytes': None, u'driversize': u'0x5000'}}

\Driver\eh6fjcon   {u'Added': {u'driverstartiobytes': None, u'driverinit': u'0x964e46ee', u'driverunload': u'0x0', u'irp': {u'IRP_MJ_CREATE_MAILSLOT': u'0x964e5fab', u'IRP_MJ_SET_QUOTA': u'0x964e5fab', u'IRP_MJ_SET_SECURITY': u'0x964e5fab',
                   u'IRP_MJ_SET_VOLUME_INFORMATION': u'0x964e5fab', u'IRP_MJ_WRITE': u'0x964e5fab', u'IRP_MJ_FLUSH_BUFFERS': u'0x964e5fab', u'IRP_MJ_DEVICE_CONTROL': u'0x964e5fab', u'IRP_MJ_CLOSE': u'0x964e5fab',
                   u'IRP_MJ_INTERNAL_DEVICE_CONTROL': u'0x964e5fab', u'IRP_MJ_DIRECTORY_CONTROL': u'0x964e5fab', u'IRP_MJ_QUERY_INFORMATION': u'0x964e5fab', u'IRP_MJ_SYSTEM_CONTROL': u'0x964e5fab',
                   u'IRP_MJ_CLEANUP': u'0x964e5fab', u'IRP_MJ_PNP': u'0x826f20e5', u'IRP_MJ_POWER': u'0x964e5fab', u'IRP_MJ_CREATE_NAMED_PIPE': u'0x964e5fab', u'IRP_MJ_LOCK_CONTROL': u'0x964e5fab',
                   u'IRP_MJ_QUERY_SECURITY': u'0x964e5fab', u'IRP_MJ_SHUTDOWN': u'0x964e5fab', u'IRP_MJ_DEVICE_CHANGE': u'0x964e5fab', u'IRP_MJ_SET_INFORMATION': u'0x964e5fab', u'IRP_MJ_QUERY_EA': u'0x964e5fab',
                   u'IRP_MJ_READ': u'0x964e5fab', u'IRP_MJ_FILE_SYSTEM_CONTROL': u'0x964e5fab', u'IRP_MJ_CREATE': u'0x964e5fab', u'IRP_MJ_QUERY_VOLUME_INFORMATION': u'0x964e5fab', u'IRP_MJ_QUERY_QUOTA':
                   u'0x964e5fab', u'IRP_MJ_SET_EA': u'0x964e5fab'}, u'filepath': u'C:\\Windows\\System32\\drivers\\eh6fjcon.sys', u'devices': [{u'devicename': u'\\Device\\KabCleanner', u'devobj_extension': {u'deviceobject': u'0x8549b038'},
                   u'driverobject': u'0x857f7030', u'deviceobject': u'0x8549b038', u'drivername': u'\\Driver\\eh6fjcon', u'type': u'invalid'}], u'driverstart': u'0x964e4000', u'driverinitbytes': u'558bec83ec1c5356', u'sha':
                   u'a2e797d3dc7e77113d5d4667ac3e49a47f8f5768', u'driverstartio': u'0x0', u'driverunloadbytes': None, u'driversize': u'0x4360'}}
```

Figure 26: Two driver files are dropped in %SYSTEM32%\drivers\.



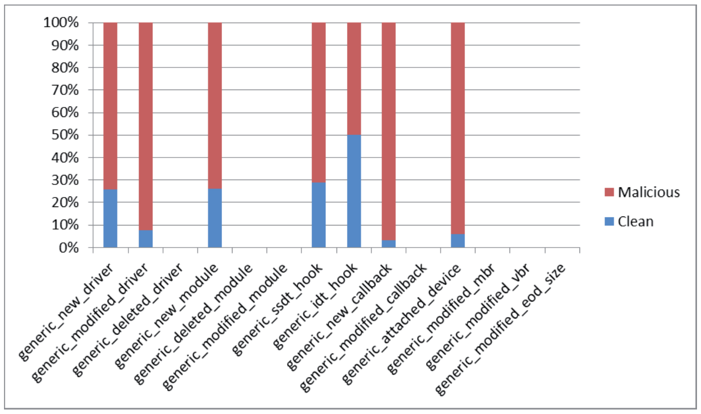
```
NtSetValueKey    {u'functionva': [{u'Original': u'0x82848606'}, {u'Changed': u'0x964e16c9'}], u'startbytes': [{u'Original': u'6a7868508c6982e8'}, {u'Changed': u'558bec5151ff151c'}]}
```

Figure 27: The SSDT hook lies in the second driver.



```
\Driver\PnpManager    {u'devices': [{u'Added': {u'devicename': u'\\Device\\0000004f', u'devobj_extension': {u'deviceobject': u'0x85765270'}, u'driverobject': u'0x83dacd88', u'deviceobject': u'0x85765270', u'drivername': u'\\Driver\\PnpManager', u'type':
                      u'FILE_DEVICE_CONTROLLER'}}]}
```

Figure 28: Kernel memory differencing produced by a sample of Mal/DownLdr-BZ.



Combining the results above with our experience in analysing rootkits, we assigned maliciousness indexes to each signature following the *Cuckoo* signatures severity style. In *Cuckoo*, a signature's severity field indicates its maliciousness. For example, a signature that triggers to highlight an action that is usually performed by clean software will have the lowest severity value of '1'. An action frequently performed by clean software will trigger a signature of a higher severity value, '2'. Actions that are largely only exhibited by malicious software will trigger a signature that has the highest severity value of '3.'

Our previous assumption that modifying the IDT or the SSDT should mainly be performed by malicious drivers proved to be incorrect. It transpires that the registration of a new callback is more suspicious than, for example, an SSDT hook. We showed earlier how a clean driver installed an IDT hook in addition to hooking the IRP table of the disk driver. An expected finding for us was the low severity of generic_new_driver and generic_new_module, since this is behaviour that can be induced by both clean and malicious files. Generic_ssdt_hook was also relatively low, but we attribute that to the high number of clean software security drivers in our clean set. Most of these will add SSDT hooks for monitoring purposes. We decided to make the severity of this signature medium. Generic_modified_driver, generic_new_callback and generic_attached_device provided most results from this experiment. We can assign a medium severity to those, since they are more likely to be performed by malicious files. For the rest of the samples where we did not get any hits, we assigned a severity based on past experience and our understanding of how malicious rootkits operate.

It became obvious from the results that no single signature from those that fired is a clear indication of maliciousness. Some were weighted when it came to malicious behaviour, but none were sufficiently high on their own. Some signatures failed to trigger so we had to resort to our experienced opinion to assign a severity level. Table 4 is a breakdown of the severity levels we assigned to indicate maliciousness.

4. CONCLUSION

We have presented an automated analysis system that performs kernel memory differencing, focusing on identifying rootkit-like behaviour. After testing this system against some prevalent rootkits and a set of clean and malicious drivers we can draw several conclusions.

Primarily, we proved that the system successfully revealed rootkit manipulations in the cases of the prevalent rootkits tested. Although the system did not capture all the changes

Signature name	Severity
generic_new_driver	1
generic_modified_driver	2
generic_deleted_driver	1
generic_new_module	1
generic_deleted_module	1
generic_modified_module	2
generic_ssdt_hook	2
generic_idt_hook	2
generic_new_callback	2
generic_modified_callback	3
generic_attached_device	2
generic_modified_mbr	3
generic_modified_vbr	3
generic_modified_eod_size	3

Table 4: breakdown of the severity levels we assigned to indicate maliciousness.

documented by other researchers, we believe that in a high-throughput cluster environment, the amount of information provided will suffice to indicate maliciousness. Analysts using the system can then perform extensive research on the sample. Since the signatures can also match on specific data, this system will provide the opportunity to cluster samples based on induced kernel changes. For example, the system allows clustering of samples that add devices to \Filesystem\FltMgr or hook IRPs of the disk miniport driver. The signatures can even validate IRPs outside drivers by checking the size and address of ntoskrnl. Correlating changes in the kernel and validating them can be done on static snapshot data, which ensures authenticity, unlike trying to query volatile information on a live system.

4.1 Future work

Although we are getting good results, work remains to be done before the system is complete. First and foremost, the system must reliably be able to replicate every file that it receives.

Exploring other areas of the kernel

It is still worth exploring the possibility of examining other areas of kernel memory. For instance, checking the functions exported by ntoskrnl for hooks (TDSS hooks IofCallDriver, which is a function exported by the kernel). As part of our future work we intend to improve the identification of modifications to modules. We currently only present the image base, size and name of a module. Additionally, research will be done into comparing individual kernel objects, to see if any useful information can be obtained as long as the analysis processing time is not increased too much.

Adding features

An obvious enhancement to the system would be the ability to handle 64-bit files. At the moment the system does not have this functionality, but with common rootkits [2] now infecting x64 systems, this would be a worthwhile improvement.

Straying from the kernel, investigation will be conducted into the ability to detect user-mode rootkits building on the system we have already. Families like Beta Bot [17] utilize user-mode API hooking and other rootkit techniques in order to remain on system. This would be useful information to get from an automated analysis.

Finally, the reporting needs improvement in order to show where the same device is being attached to several driver objects. With more data being produced through the kernel memory scan like objects, ntoskrnl exports, we also intend to give more focus to devices.

Overall, we believe that such an automated system will provide greater visibility into the state and prevalence of rootkits in the wild. It will enhance the tracking of families and the identification of rootkit behaviour whether by clean or malicious files. It functions as a system that could flag malicious behaviour and also as an enabler to flag samples that could prove to be worthy of further investigation.

Improvements to sample clustering

Sample clustering is not just a problem for rootkits, but for all malware, with numerous papers being penned on the subject. Identifying common behaviours and traits in large sample sets is a problem every anti-virus vendor faces. The system presented in this paper is not intended to solve sample clustering problems. However, we believe that it can serve as a platform for identifying behaviour exhibited by samples in ways that other systems cannot. For example, many rootkits will typically install a service and load a driver. This information alone cannot be relied on for clustering purposes. Our system exposes information per service and/or driver loaded, such as device names and types, that can be used to cluster samples. While the usual static and dynamic analysis methods provide a lot of data that can be used to cluster samples, the ability to access, dump, and diff areas of kernel memory gives a whole new dimension to the dataset.

Using the data exported by our system (as described in Section 2.1), it is possible to write signatures (as described in Section 2.4) which can be used to identify common traits. Clustering samples that, for example, commonly hook the same APIs, or insert devices into the device stack of the same drivers is a good way to group rootkits into the respective families.

As the next stage of this project we would like to expose the signature output to be utilized by our existing sample clustering system to increase the accuracy of clustering rootkit samples.

REFERENCES

[1] Rootkit. Dictionary.com. http://dictionary.reference.com/browse/rootkit.

[2] Ferrie, P. The curse of Necurs – Part 1. Virus Bulletin, April 2014, p.4. http://www.virusbtn.com/pdf/magazine/2014/201404.pdf.

[3] DrWeb. BackDoor.Tdss.565 and its modifications (aka TDL3). http://www.drweb.com/static/BackDoor.Tdss.565_%28aka%20TDL3%29_en.pdf.

[4] BAE Systems. Snake campaign and cyber espionage toolkit. 2014. http://info.baesystemsdetica.com/rs/baesystems/images/snake_whitepaper.pdf.

[5] Gmer.net. http://www.gmer.net/.

[6] Cuckoo Sandbox. http://docs.cuckoosandbox.org/en/latest/.

[7] Russinovich, M. E.; Solomon, D. A.; Ionescu, A. Windows Internals, Fifth Edition. 2009.

[8] Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[9] OSDev.org. Partition Table. http://wiki.osdev.org/Partition_Table.

[10] Rodionov, E.; Matrosov, A. The Evolution of TDL: Conquering x64. http://www.eset.com/us/resources/white-papers/The_Evolution_of_TDL.pdf.

[11] Johnson, J. Alureon: The First ITW 64-Bit Windows Rootkit. Virus Bulletin. 2010. https://www.virusbtn.com/pdf/conference_slides/2010/Johnson-VB2010.pdf.

[12] Rodionov, E.; Matrosov, A. Mind the Gapz. http://www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf.

[13] G Data SecurityLabs. Uroburos Highly complex espionage software with Russian roots. 2014. https://public.gdatasoftware.com/Web/Content/INT/Blog/2014/02_2014/documents/GData_Uroburos_RedPaper_EN_v1.pdf.

[14] New York Times. Suspicion Falls on Russia as ‘Snake’ Cyberattacks Target Ukraine’s Government. 2014. http://www.nytimes.com/2014/03/09/world/europe/suspicion-falls-on-russia-as-snake-cyberattacks-target-ukraines-government.html .

[15] malware-lu. Necurs Quick Analysis. 2012. https://code.google.com/p/malware-lu/wiki/en_necurs_analysis.

[16] Wyke, J. Notorious “Gameover” malware gets itself a kernel-mode rootkit... NakedSecurity. February 2014. http://nakedsecurity.sophos.com/2014/02/27/notorious-gameover-malware-gets-itself-a-kernel-mode-rootkit/.

[17] MacDermid, K. Beta Bot – A Code Review. Arbour Networks. June 2013. http://www.arbornetworks.com/asert/2013/11/beta-bot-a-code-review/.

[18] Monthly Malware Statistics, May 2011. Securelist. 2011. http://www.securelist.com/en/analysis/204792214/Monthly_Malware_Statistics_May_2011.

[19] Zaki, A. Guntior bootkit up to new tricks. Naked Security. June 2013. http://nakedsecurity.sophos.com/2013/06/12/guntior-bootkit-up-to-new-tricks/.