2008 Annual Computer Security Applications Conference

MalTRAK: Tracking and Eliminating Unknown Malware

Amit Vasudevan
CyLab, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, USA
amitvasudevan@acm.org

Abstract

Malware or malicious code is a rapidly evolving threat to the computing community. Zero-day malware are exploiting vulnerabilities very soon after being discovered and are spreading quickly. However, anti-virus tools, which are the most widely used countering mechanism [30], are unable to cope with this. They are based on signatures which need to be computed for new malware strains. After a new malware strikes and before the signature is found allows sufficient time for the malware to perform its damage.

We propose a new framework, codenamed MalTRAK, which ,when deployed on a clean system, guarantees that any effects of a known or unknown malware can always be reversed and the system can be restored back to a prior clean state. Our framework also maintains detailed dependency lists of system operations which can be used for further forensic analysis. We are able to achieve this without imposing any restrictions on the nature of programs that can be executed by the user and without the user noticing any perceptible system slowdown due to the framework. Furthermore, we are able to track modifications to the system at a level that ensures that we can always monitor any changes to the system state even if a malware modifies the system during execution.

We implemented and evaluated MalTRAK on Windows, using 8 known malware assuming they were unknown strains. We then compared our results with two popular commercial anti-virus tools. We were able to successfully restore all the effects of the 8 malware, while the commercial tools, on an average were only able to restore 36% of all their effects put together. For one of the malware samples, the commercial tools could only detect it but could not repair any of its damage. Further, for two of the malware samples, the commercial tools were completely unable to detect or restore any of their effects. Our results show that signature based mechanisms in addition to not being able to prevent infection by new malware strains, are not very effective in removing an infection even after a signature has been developed. Our experience shows that non-signature based approaches, such as MalTRAK, are the next step towards combating the threat of ever-evolving malware.

1. Introduction

Malware, or malicious code, is widespread today and have devastating effects. They propagate by infecting a target system. Anti-virus software today are the most widely used tools to detect and counter malware [30]. They work by employing signatures- a unique form of identification of a malware and its related strains. The contents of the system (typically the filesystem and memory) are then checked against a database of such signatures to detect and remove any infection.

The first line of defense anti-virus tools provide is a real-time system monitoring mechanism which can automatically detect a malware before it can infect a system. They monitor the underlying filesystem for access/modifications and perform file scanning the moment it is modified or before it is executed. While real-time monitoring is an effective tool against known signatures of malware, they are rendered useless against new malware or variants of the malware whose signature is not within the tools signature database.

The number of new malware or malware strains are rapidly increasing year after year. A recent report showed an increase of more than 50% from the latter half of 2006 to the first half of 2007 for new malware being discovered [8]. Zero-day malware are becoming more and more prevalent, exploiting vulnerabilities within moments of their discovery and spreading quickly [27]. With underground exploit tools such as MPack [16] being constantly developed and refined, malware writers do not have to write code to discover the vulnerabilties themselves, but rather rely on a tool that is constantly developed in order to churn out malware by the numbers. This situation is further worsened by the fact that the major cause of malware proliferation are the end users themselves [23]. Users either disable anti-virus programs, install applications downloaded off the internet or execute attachments within emails which can potentially contain malicious software. Thus before the signature of an unknown malware is discovered , the malware can do most of its damage.

To this end, anti-virus tools provide a passive component which can scan the system after an infection has taken place in order to detect and remove it. However, this approach has serious limitations. Only previously recognized and analyzed malware can be detected and removed. Further, even if a malware is detected, it could do irreversible damage to the system by deleting files or overwriting them beyond repair. While the percentage of such malware in existence today are relatively small, a recent report from a leading anti-virus vendor suggests that the number of such malware are increasing [21]. Also signature-based approaches are not very effective in the realm of coding techniques such as polymorphism and metamorphism (which most if not all current generation malware employ) which are used by malware to hide themselves in memory.

There have been a few recent works geared towards a solution

1063-9527/08 $25.00 © 2008 IEEE
DOI 10.1109/ACSAC.2008.44

to this problem [9, 28, 19] which attempt to detect and remove malware without employing signatures. However they have one or more of the following limitations: (a) they entail the user to place a policy (rule) or rely on an external system to classify trusted vs untrusted programs, based on which the system attempts to defend against unknown malware. This can be potentially dangerous as a user can place a policy which can allow a malware to get into the system or an external system can classify an application as trusted, but a malware can exploit a vulnerability in that application, (b) they incur considerable runtime overhead which makes them unsuitable for end system deployment, and (c) they do not capture the modifications to the system at all times and may miss changes. Section 2 discusses these approaches and other related work in more detail.

We propose a new framework, codenamed MalTRAK, which when deployed on a clean system, guarantees that any effects of a known or unknown malware can always be reversed thereby allowing the system to be restored back to a prior clean state. The salient features of our framework are: (a) it does not require the user to place any policy beforehand. In other words the user can run any applications without worrying about a possible infection, (b) the user notices very minmal or no perceptible difference in the system speed due to our framwork, (c) the framework monitors system operations at a level that makes it very difficult to subvert and (d) the framework maintains state information regarding the system modification. This information is used to revert the system to a prior clean state as well as provide the user with a detailed dependency information. This information can then be used for forensic analysis as well as salvage more clean data if necessary.

A user typically interacts with MalTRAK when malware effects are being perceived (either via recurrent crashes, system slowdown, network traffic, or exhaustion of system resources etc.) or on a routine basis in order to make sure that the system has not been infected (this would be the case where the user is an administrator and the system is a server). Disinfection of an infected system typically involves the user to select an operation or a group of operations whose cascading effects are then removed automatically by our framework while preserving most of the clean data. While at the outset, the selection of an operation might seem a daunting task, our framework provides what we call alerts to assist the user. Alerts are framework determined possible starting operations (e.g a system driver that has been writen to the system folder whose entry is then set in the configuration to autoload the driver on system startup). Alerts are based on the notion that there are only a finite number of ways a malware could try to remain persistent and active on a given OS (see Section 4.4.1).

We summarize the major contributions of our paper:

- We propose a mechanism which dynamically maintains various states of the system with minimal overhead in terms of both disk space and runtime latency. This mechanism allows us to execute any code on the system without prior knowledge of its integrity while enabling us to track the changes it makes to the system. This allows us to capture any form of malicious activity either as a process in itself or by exploiting another process.
- We propose a new mechanism which captures modifications to the system at the lowest possible level while still being able to maintain high level mappings. Thus, while existing approaches cannot handle malware such as Rustock and Nailuj (which are using techniques that are touted to become standard pretty soon [11]), we are able to easily capture their behavior.
- We propose a mechanism which maintains a mapping between code and its interaction with the system and allows restoration without the risk of reinfection. We leverage this mechanism in order to detect probable malicious activity at runtime, to allow the activity to take place and restore the sytem to a prior good state later.
- We have implemented a prototype of MalTRAK on the Windows OS and have evaluated it using 8 known malware assuming they were unknown strains. We then compared our results with two popular commercial anti-virus tools. We were able to successfully restore all the effects of the 8 malware, while the commercial tools, on an average were only able to restore 36% of their effects put together. For one of the malware samples, the commercial tools could only detect it but could not repair any of its damage. Further, for two of the malware samples, the commercial tools were completely unable to detect or restore any of their effects.

The rest of the paper is organized as follows: We first present some background and related work in Section 2. In Section 3 we present the design of our framework. We follow that by discussing our implementation under the Windows OS in Section 4. In Section 5 we present the evaluation of MalTRAK. This is followed by a discussion on the security aspects of the framework in Section 6. We then conclude the paper in Section 7 summarizing our contributions.

2 Related Work

There have been several works related to the area of logging and recovery in general. Broadly they can be categorized into: (a) trusted/untrusted code execution approaches, (b) VM/Snapshot based approaches, (c) filesystem based approaches and (d) behavior/semantic based approaches.

Trusted vs Untrusted code execution: These approaches require running processe(s) to be categorized into trusted and untrusted. Trusted procese(s) are allowed to run as is while the untrusted processes are run in a sandbox like environment which employs some tainting analysis to track trustworthiness of the data.

SEE [28, 19] proposed the idea of one-way isolation where untrusted processes modify a temporary copy of the filesystem rather than the original. SEE allows the commit of such operations once the untrusted processe(s) complete. SEE allows an untrusted program to run to completion but might not be able to commit changes. Back to the Future [9] is a framework for automatic malware removal and system repair. Unlike SEE it allows untrusted processe(s) to write to the filesystem immediately but might not be able to run the process to completion.

The Taser [7] intrusion recovery system logs all processes, files and network operations. It then uses this information to revert changes to the file system depending on a signal from a IDS. Since the IDS may have false positives, Taser may never respond to a trusted process being tainted. Moreover, it results in the loss of too much clean data in the event that the IDS does not respond immediately after intrusion happens.

This approach has several limitations: (a) classifying processes as untrusted and trusted is not a trivial task, (b) a user may misclassify a process as trusted, but a malware could exploit a vulnerability in the process which goes undetected and (c) the runtime latency of running an untrusted process is typically in the order of 2 times or more than its normal execution speed. This increases as more processes are being run as untrusted. Further, since there is no guarantee that a trusted process cannot have exploits, a user most of the time may tend to mark processes as untrusted which lead to latency that are unsuitable for normal system operation.

Virtual Machine and Snapshot based approaches: These approaches use a virtual machine container or periodically take snapshots of the whole system for recovery purposes. King et. al [14] add support for virtual machine monitor in the Linux kernel to achieve high performance. Revirt [5] runs applications inside a virtual machine to log their events. Then it analyzes the intrusion by replaying the logged events. TTVM [13] runs the operating system inside a virtual machine for debugging purposes. Backtracker [12] identifies potential sequence of steps that occured in an intrusion. It runs the operating system as a guest and uses a virtual machine monitor to log events.

Using virtual machines to run untrusted programs has its shortcomings. First untrusted processes cannot access resources created by program running out of the virtual machine, which may break many programs, secondly virtual machines are resource expensive.

ROC is a framework for recovering from system component failure and operator errors [1, 25]. It contains three stages: rewind, repair and replay. It has a large overhead for logging and recovery and incorporates expensive snapshots of the system. Windows System Recovery [32] is another approach which maintains periodic absolute snapshots of the system to be able to restore to a previously saved snapshot.

Current snapshot based approaches contain no state information which can result in the user reverting to a snapshot that might cause reinfection and/or resulting in the loss of too much clean data. It also results in considerable overhead in terms of both time and space during the snapshot.

Filesystem based approaches: These approaches implement a seperate filesystem for crash recovery purposes. LFS [26] is a filesystem which is structured like a log. This speeds up both file writing and crash recovery. Reparirable File Service [33] uses a similar idea to repair compromised file servers such as NFS. It interposes a RFS server which implements logs to restore file operations later.

Filesystem based approaches in general do not maintain any state information about the files such as parent/child relationships (Repairable File Service however uses client-side logging to track parent/child relationship). Filesystem based approaches only provide unconditional restore feature. Further, they are tied only to the filesystem and do not provide configuration restoration such as the registry. Finally, they are not designed to be implemented on top of a regular file system such as those found in commodity OSes such as Windows and Linux.

Behavior/Semantic based approaches: These approaches employ a combination of static and dynamic techniques (disassembly and monitoring) in order to characterize a program. Behavior based spyware detection [15] uses detects known and unknown spyware which use the Browser Helper Object (BHO) extensions. It is based on the notion that such extensions normally do not leak data beyond their operating space. However, it only detects spyware and is not applicable to general malware. Giffin et al. [6] disassemble a binary to remotely detect manipulated system calls in a malware. Kruegel et al. [17] employ static binary analysis to detect kernel-level rootkits. SAFE [3] and Semantic-Aware Algorithm [2] are other examples of malware detection algorithms employing similar static analysis techniques. The problem with the above approaches is that they incur considerable runtime overhead to be deployed dynamically. While they can be used as a passive component (to scan after an infection), they suffer from limitations regarding disassembly (obfuscation [20] and instruction overlap [4]). Further, a malware could hide itself once active and prevent such operations in the first place.

In comparison, our framework does not require the user to make distinction between untrusted and trusted and does not use virtual machine based or absolute snapshots which take up too much resources. It allows all processes to run to completion while at the same time is capable of restoring a system back to a previous state in case of any infection. It incurs very little or no perceptible overhead as seen from our performance measurements. Last but not the least, it tracks system operations from the lowest possible level making it very difficult to subvert.

3 Design

3.1 Overview

Figure 1 illustrates the components of our framework: a view engine, a mapper engine, an interceptor engine and a disinfection engine. The view engine maintains different logical states of the system which can be switched to, back and forth. The mapper engine ties the different views together with state information that is used as an aid during disinfection. The interceptor engine endows the framework with the ability to capture changes that occur to the system state dynamically at the lowest possible level. The disinfection engine is responsible for removing the effects of malware (if any) and revert the system back to a clean state. The following sections describe our design goals and the framework components in further detail.

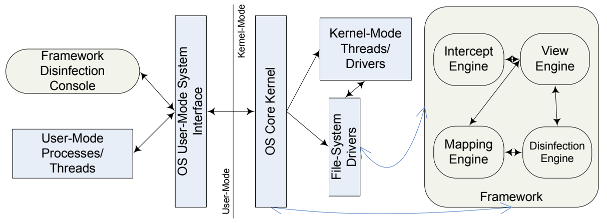


Figure 1. Framework Overview

3.2 Design Principles, Assumptions and Goals

The design of our framework is based on the following core principle: *Design for Protection against Persistence*. Our assumption is that: *The framework will initially be deployed on a clean system.*

From our design principle, our framework should prevent a

malware from persisting in the system. We argue that a malware, in most if not all cases, will make an effort to remain persistent in the system so that it can get control most of the time and propagate and cause damage. We believe that our assumption is reasonable since if a user already had an infection and an anti-virus tool is unable to detect or clean the infection, the user would in any case restore the system from a backup or a fresh installation. Note that, from our principle we do not take into account the runtime integrity of the system. In other words, a malware is free to modify the system or even grind it to a halt, however using our framework, a simple restart will be capable of removing it from the system.
Given the above principle and assumption, the following are our design goals:

- Ensure that the user can always revert back to a system state that is devoid of any malicious footprints
- The user, most of the time, would find no perceptible difference in system speed when running the framework as compared to doing so without the framework.
- Ensure that the loss of clean data is minimized during disinfection.
- Ensure that the framework can capture all interactions that modify system state at all times.

3.3 Views

A view is a logical representation of the state of the system at any given point in time. By state we mean the persistent state (e.g filesystem and configuration information). It is represented as V_n where n is the view identifier.

Views are composed of objects. An object could be a file, a folder or a configuration key or value. An object view represents the contents of the object that is visible to the entire system for a given view. An object session is defined as the time from when the object is opened for access to when it is closed/deleted.

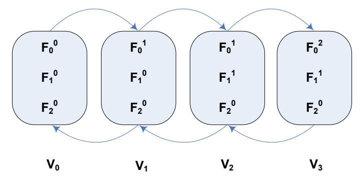
The framework creates a new view whenever a modification (write) is detected on the current view. This is similar to the copy-on-write(COW) mechanism that is used in virtual memory systems. This approach has the advantage that we can track modifications to the system state at a very fine level. Note that, creation of new objects within a view will not go through this COW mechanism. The COW is only employed when a modification is detected to an existing object within a view. Further, the COW is performed on an object session as a whole, which means that once the object is opened, further accesses to it occur on the COWed copy.

While on the surface the COW mechanism might look like an overkill, in fact it is not so. Consider the case when an object is a file. Modifications are very seldom done to executable files but only to data files in the normal course of system operation. Further, typically an application opens a file during start, reads/writes to it and closes it once it is done. These facts make our approach very efficient (see Section 5.2 for the framework performance evaluation)

For an object O_n, the object view is denoted by: O_n^m where m represents the view level of the object. The current view level of an object is essentially the number of times the object has been COWed. The view level of an object O_n that has never been COWed is 0 and is denoted using O_n^0. For the rest of the paper, we will use F_n to denote a file object, D_n to denote a directory object and C_n to denote a configuration object and O_n to denote an object in general.

```
 1. OPEN F0
 2. OPEN F2
 3. READ F2 AT OFFSET a
 4. WRITE F0 AT OFFSET b
 5. READ F2 AT OFFSET a+512
 6. WRITE F0 AT OFFSET b+512
 7. READ F2 AT OFFSET a+1024
 8. WRITE F0 AT OFFSET b+1024
 9. CLOSE F0
10. OPEN F2
11. WRITE F2 AT OFFSET c
12. CLOSE F2
13. OPEN F0
14. WRITE F0 AT OFFSET b+4096
15. CLOSE F0
```

(a)



(b)

Figure 2. Framework Design: Views; (a) System Operations on a group of files, and (b) Corresponding views (logical states) that can be switched to back and forth

The concept of views and how they operate is best explained using an example. Let us consider pseudo-code in Figure 2a. The pseudo-code describes a system performing a set of operations on a certain group of file objects F_0, F_1 and F_2. Figure 2b shows the resulting views within our framework.

View V_0 consists of file object views F_0^0, F_1^0 and F_2^0. When file object F_0 is opened in line 1, Figure 2a, a new view V_1 is created consisting of file object views F_0^1, F_1^0 and F_2^0. Since, file object F_0 was opened for read and write, it is COWed within view V_1 resulting in the file object view F_0^1. Operations in lines 3–9, Figure 2a are then performed on file object views in V_1. Note that though these set of operations result in multiple writes to file object F_0, they do not result in the creation of new views since it is still in the same file object session. When file object F_1 is opened for write in line 10, Figure 2a, a new view V_2 with file object views F_0^1, F_1^1 and F_2^0 is created. The opening of file object F_0 in line 13, Figure 2a results in the creation of a new view V_3 with file object views F_0^2, F_1^1 and F_2^0 since file object F_0 was closed and reopened again for write signifying a new file session. Hence, it was COWed within view V_3 resulting in file object view F_0^2.

The views shown in figure 2b thus represent the various logical states of the system — states that can be switched to and fro.

3.4 Mappings

Mappings are a mechanism which provide a way to link object views within views. Essentially a mapping seeks to find the object view O_n^m which is responsible for creating or modifying an object view O_y^x. We write this relationship as:

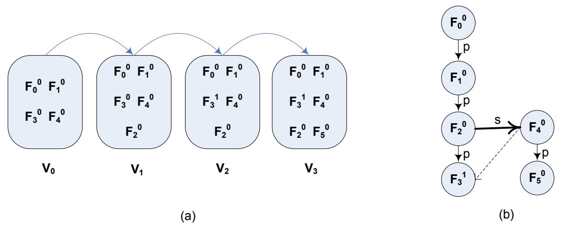
$$p(O_y^x) = O_n^m$$

For example, let us consider a target code stream that results in the creation of a file object view F_y^x within a view V_a. If the target code stream resides in a regular executable, a user- or kernel-mode dynamic library or, a kernel-driver, $p(F_y^x) = F_n^m$, where F_n^m is the file object view for the executable image file. If the target code responsible for resides in an allocated range of memory, then $p(F_y^x) = F_n^m$, where F_n^m is the file object view of the executable image file that allocated the range of memory.

An object view O_n^m can also have an indirect relationship with object view O_y^x. This could be the case when O_n^m accessed an object view that was created or modified by object view O_y^x or vice versa. We write this relationship as:

$$s(O_y^x) = O_n^m, s(O_n^m) = O_y^x$$

Let us consider Figure 3a which shows various views in a system. Let us assume that within view V_0 a malware comes as an email attachment which consists of a file view object F_1^0. When the user clicks this attachment, the email client whose executable image file object view is F_0^0, executes F_1^0. Further, lets assume that F_1^0 creates a kernel driver F_2^0 and loads it into the system. This results in the creation of view V_1. Assume that F_2^0 now writes to an executable F_3^0 and infects it creating F_3^1. This results in the creation of view V_2. Then let us assume that the user runs an executable compression program stored in F_4^0 which reads F_3^1 and creates a output executable F_5^0, resulting in the creation of view V_3.



F_0^0 F_1^0 F_3^0 F_4^0
F_0^0 F_1^0 F_3^0 F_4^0 F_2^0
F_0^0 F_1^0 F_3^1 F_4^0 F_2^0
F_0^0 F_1^0 F_3^1 F_4^0 F_2^0 F_5^0
V_0
V_1
V_2
V_3
(a)
F_0^0
p
F_1^0
p
F_2^0
s
F_4^0
p
p
F_3^1
F_5^0
(b)

Figure 3. Mappings; (a) Different views consisting of file object views in the system, and (b) the relationship between the various file object views

Figure 3b shows the mapping for this scenario. The direct relationships are indicated via a single line whereas the indirect relationships are indicated via a double line. Further the arrows in the indirect relationships signify the direction of access. In our example, F_4^0 reads F_3^1 which was created by F_2^0. Hence the arrowhead points to F_4^0 signifying that it is the consumer of the data.

Mappings are useful in two ways: the first is to aid in disinfection in order to remove the effects of a starting operation automatically. The second is to provide the user with a detailed dependency list of all the object views that were created/modified in the system and in order to minimize the amount of clean data that could be lost during disinfection.

3.5 Intercepts

Intecepts provide a unified mechanism by which system operations can be redirected for purposes of tracking modification to the system state.

Contemporary operating systems (e.g Windows and Linux) provide a layered architecture for access to various system components. Figure 4a depicts the layering on the Windows OS. For a system operation, the applications typically interact with a user-mode components, which invoke a kernel-mode OS front end (the system call layer) and the request is either handled in the core OS kernel (e.g a configuration operation) or passed on to supporting components such as a filesystem driver (e.g file operation) which support the desired filesystem (fat32, ntfs, ext3 etc.)

The reference to a file or a configuration entry is mostly done via what are termed handles. These are external interfaces to OS defined structure of maintaing information about a particular file or

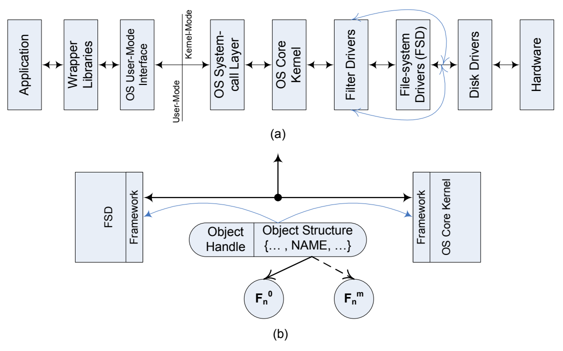


Figure 4. Intercepts: (a) Layering of System Operations on the Windows OS, and (b) Redirecting control at the lowest possible level in order to track modifications to the system state

configuration entry (such as the name, attributes etc.). The OS uses the structure on the lower levels of its implementation (e.g filesystem driver or the core kernelponent) and employs the handles on the higher levels (e.g system call interface) and user-mode. Applications use file/configuration names to obtain a handle and use the handle to communicate with the OS components thereafter.

Figure 4b), illustrates the intercept mechanism. Our framework interposes itself directly at the lowest possible levels in order to capture filesystem and configuration operations. This is the file system driver (FSD) for the filesystem and the kernel configuration manager in case where the configuration is supported in memory and disk (e.g windows registry).For systems that maintain configuration information using the filesystem (e.g Linux) it will still be routed through the FSD itself. This ensures that our framework always gets control irrespective of which points a code invokes access to the filesystem/configuration. This is unlike previous approaches which interpose at the system call level or the filter level. A code stream could directly invoke the file system driver bypassing all of the rest thereby evading monitoring.

When a request arrives at the FSD or at the kernel configuration manager, the framework gets control first. It then determines the name of the file/configuration item and uses the OS allocated structure to store the corresponding file/configuration object view information and stores the original file/configuration name in its internal memory. Thus, from the code-stream's perspective, the handle refers to the original file/configuation name, whereas from the OS perspective it refers to the current file/configuration object view name. This scheme ensures that the same OS structure is used throught the system and does not entail modification at different points with the kernel code. This is of greater importance in systems that are not open sources (such as Windows). Any request which results in the OS allocating a handle (create and open operation) are intercepted to perform this functionality. For any query on the file name (query operation), the framework replaces the original name within the request buffer so that the application does not see the difference. The rest of the read/write and other functionalities that employ the handle are directly passed on to the respective component (the FSD or the kernel configuration manager). This model also allows us to invoke the original code for the meat of the functionality while maintaining a minimal information for the intercepts.

A point to note that, the disk driver is the lowest level in the layers of system operations. However, we do not intercept at the disk driver since at that point there is no longer enough information that is passed to the driver which helps us to tie the disk request to a given file. We would then have to deal raw disk blocks which would result in a COW everytime a disk block is modified and result in increased space and time cost.

3.6 Disinfection

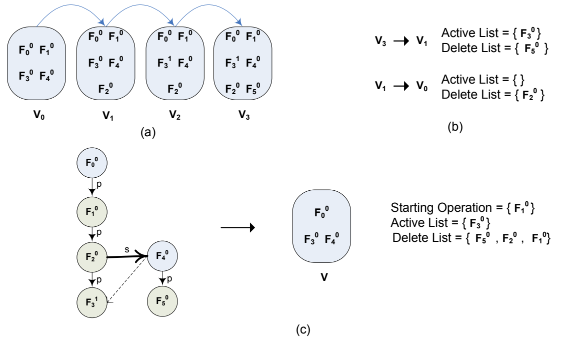
Removing the effects of malware in our framework requires the system to be reverted to a previous state. The step of reverting back to a previous system state can be done in two ways using our framework: (a) by switching unconditionally to a different view from the list of views maintained or (b) by switching conditionally to a system state comprising of various views depending upon a starting operation. In most cases, a user would go with option (b) as it completely eliminates the effect of the starting operation on the system. However, in certain cases, the user might chose a more fine-grained manual approach by trying to eliminate one view at a time in order to try and salvage some important clean data that might be missed if option (b) is employed. The choice of (a) and/or (b) depends upon a particular scenario and the malware(s) but based on our experience we find that option (b) works in most cases.

The starting operation for option (b) is usually selected from the framework alerts. Once this point is selected, the framework automates the process of removing all the effects of that operation on the system. Alerts are nothing but possible starting points that the framework automatically generates and records, which could have a malicious effect on the system. They are based on the notion that for a particular operating system, there are only a finite number of ways in which a malware could try to remain persistent. Section x lists various scenarios that our prototype implementation employs for the Windows OS. Alerts can be active or passive. If active, the user is immediately given a chance to either continue or abort the operation and in the passive mode, the framework simply records the information for use during disinfection if needed.

Figure 5 illustrates the disinfection options (a) and (b) for the scenario as discussed in Section 3.4. In both cases the disinfection engine computes two lists. The first list is called a delete list and the second a active list. The delete list represents the list of file views that need to be deleted from the current view while the active list represents the list of file views that need to be made active in order to switch to a view that represents a clean system state.

Using option (a) the user can go from view V_3 to any of the possible views V_2 through V_0. The framework will automatically compute the active and delete lists to switch to the target view. As an example, as shown in Figure 5b, the user moves from view V_3 to V_1 and then to V_0 resulting in the corresponding active and delete lists. This form of view switching is called unconditional since the user is switching from view n to view m without any knowledge of the relationship between the files in the view.

However, in option (b) the user selects a starting point, in this case the creation of file object view F_1^0 (which causes an alert as it is a kernel mode driver that has been linked to start up automatically on the next boot) and the framework uses the mapping information to conditionally switch from V_3 to V_0 automatically,



F_0^0 F_1^0 F_3^0 F_4^0
V_0
F_0^0 F_1^0 F_3^0 F_4^0 F_2^0
V_1
F_0^0 F_1^0 F_3^1 F_4^0 F_2^0
V_2
F_0^0 F_1^0 F_3^1 F_4^0 F_2^0 F_5^0
V_3
(a)
$V_3 \rightarrow V_1$
Active List = { F_3^0 }
Delete List = { F_5^0 }
$V_1 \rightarrow V_0$
Active List = { }
Delete List = { F_2^0 }
(b)
F_0^0
p
F_1^0
p
F_2^0
s
F_4^0
p
F_3^1
p
F_5^0
F_0^0 F_3^0 F_4^0
V
Starting Operation = { F_1^0 }
Active List = { F_3^0 }
Delete List = { F_5^0 , F_2^0 , F_1^0 }
(c)

Figure 5. Disinfection: (a) Views of the system, (b) Conditional view switching switches to a view by eliminating required view objects, and (b) Conditional view switching uses relationships and switches to a view by eliminating the effects of a starting operation.

eliminating all effects of the file object view F_1^0 in the system (Figure 5c). Note that using option (b), F_4^0 might read something that F_2^0 produced, that might not be directly incorporated into F_5^0, however disinfecion will result in F_5^0 being deleted. While this might be thought of as losing more clean data, in practice it is not. If an application reads contents of an executable, chances are that it is using it for packing or encrpytion in which case the resulting executable should be deleted in any case. If an application is reading data that is written by a possible malware, and is writing output data, there is a good chance that the input information will be embedded in some form in the output, in which case it is a good idea to remove those effects. However, there might always be situations where an application reads data that is written to but does not use that data for some of its outputs. In this case, disinfecting the system using option (b) will result in the loss of some clean data using our framework which needs to be salvaged using option (a) if desired.

4 Implementation

To evaluate our framework we have developed a prototype implementation for the Windows XP operating system. The implementation consists of a kernel mode driver and a user-mode application. The kernel-mode driver is the framework which encompasses all the four subsystems: the view, mapping, intercept and recovery engines while the user-mode application is the recovery console that interacts with the user to deliver alerts and to perform recovery.

For implementation purposes, our prototype modifies different parts of the Windows OS kernel and supporting components. Since Windows is a commercial OS, such modifications have to be done in binary. We use redirection, a method similar to target function overwriting with trampolines [31, 10]. The basic idea is to disassemble enough instructions and implant a series of instructions on the target intercept location which ultimately transfer control to our framework internal function. This allows us to invoke the original internal function within our framework. Thus we can perform framework specific operations while chaining to the original internal functions for the meat of the processing.

4.1 Views

Views are implemented by using object view stacks. A object view stack is a collection of object views for a particular type of object (such as a file, configuration key etc.). A object view stack is a LIFO structure that operates in the same way as a program stack, where information can be pushed into and popped out of the stack. Each object view stack has a top of stack pointer called the object view stack pointer.

A view is then defined to be the list of all objects that do not have a object view stack (i.e they have not been modified) and the set of object views which form the top of stack element of their respective object view stacks. We call this list the access vector for the corresponding view. Thus, for a given view, simply by looking at the access vector, the framework can determine which physical file the requests must be routed to.

4.2 Mappings

The framework uses call-chains and call-contexts in order to implement mappings.

A call-chain is simply a list of function entry points that ultimately lead to the actual functionality. In our case we use call-chains only for filesystem and configuration functionalities. For example, if one considers the CreateFileA API, it follows a series of API invocation until it reaches the FSD with the request to create a file. A call chain is only created comprising of modules which are always visible in the address space of every process. Thus, we do not take into account dlls that might wrap around the APIs.

A call-context essentially describes information about the originator of a particular call to either the file system or the configuration manager. This is used to identify the target file fmn which is responsible for the invocation. The framework establishes several hooks within the host OS kernel in order to build call-contexts.

It intercepts process/thread initiation and termination APIs in both user and kernel-mode. More specifically it hooks ZwCreateProcess, ZwCreateThread, ZwTerminateProcess, and PsCreateSystemThread calls in order to keep track of the processes/threads that are being created. The framework also intercepts APIs such as ZwReadProcessMemory, KeStackAttachProcess and ZwWriteProcessMemory in order to keep track of interprocess memory writes and maintains a list of memory regions and the originating process/module information. This ensures that a situation where a malicious process injects code into a benign process in order to create files on its behalf [18] is properly tackled.

The framework also intercepts APIs such as LdrLoadLibrary, LdrFreeLibrary, LdrGetModuleHandle, LdrGetProcAddress in order to keep track of the modules that are being loaded/ unloaded in user-mode. It also intercepts ServiceManager APIs as well as ZwLoadDriver in order to keep track of the kernel-mode drivers and modules that are being loaded.

The framework intercepts calls to ExAllocPool and ZwVirtualAlloc in order to track memory regions that are being allocated by the running code-streams in both user and kernel-mode. Thus it is able to track if a given request is coming from code being executed in these memory regions. Finally, the framework also hooks all the entrypoints within the filesystem/configuration call-chains and keeps track of the return address.

We hook entry points within the host OS and do not modify the host OS structures directly for two reasons. First, from an implementation perspective we would like to stay as much close to documentedness as possible. The kernel structures for the above operations keep changing and messing with them only leads to system instability. Second, even if a malware hooks the above functions, it still needs to chain to it since implementing these functions is not a trival task for malware writers. So we ensure that we always get control while keeping our implementation as stable as possible.

```
If (previousExecutionMode == KernelMode){
   if (request came from allocated memory pool)
      p(O_y^x)= F_n^m from CALLCONTEXT
   else if (request came from module space)
      p(O_y^x) = F_n^m of module image
   else if (request came from core OS kernel and
      no module  has been loaded after the framework)
      p(O_y^x) = F_n^m of core OS kernel image
   else if (request came from core OS kernel and
      module has been loaded after the framework)
      for (each loaded module)
         p(O_y^x) += F_n^m of module image
}
```

(a)

```
if (request came from allocated memory pool)
   p(O_y^x)= F_n^m from CALLCONTEXT
else if (request came from dll that was created
       and/or modified)
   p(O_y^x) = F_n^m of dll image
else if (request came from standard/system dlls){
   if (code modified)
      p(O_y^x) = F_n^m from CALLCONTEXT
   else
      p(O_y^x) = F_n^m of dll image
}else if (request came from exeutable space)
   p(O_y^x) = F_n^m of executable image
```

(b)

Figure 6. Mappings: Generating Object View Relationships

When a filesystem or configuration object is created, opened, modified or deleted, the framework within the call-chain hooks for these functionalities, in kernel-mode employs the logic as shown in Figure 6a in order to determine the mapping. We assume that the object (filesystem or configuration) that is being accessed is O_x^y. Similary, in user-mode the framework employs the logic as shown in Figure 6b. Similar logic is employed when a filesystem or configuration object is read, in order to determine the sibling mappings.

We treat kernel-mode and user-mode seperately. The idea is that within kernel mode all modules are always visible to themselves and every process. Thus, we need to distinguish between requests that came from kernel-mode versus request that came from user-mode which included the kernel-mode point in the call-chain. We do so using the ExGetPreviousMode windows kernel function which returns the previous operating mode of the processor. We make use of the fact that a caller cannot call from the kernel pretending to be from user as that would break the existing implementation of Windows OS kernel.

4.3 Intercepts

Before we discuss our implementation of intercepts, a few words regarding how the Windows kernel is structured is in order. The Windows OS kernel is completely object based. Every element from files to processes to buffers are all encapsulated in the form of an object. Thus, for all files, directories, volumes etc, we deal with the FILE_OBJECT while for processes we deal with the EPROCESS object and so on. The implementation of intercepts on the Windows OS kernel consists of file system and registry (configuration) intercepts.

Filesystem Intercepts: A file system is implemented as a file system driver within the Windows kernel. A FSD has entry points which the Windows kernel invokes in response to a file access either in user- or kernel mode. As an example, invoking CreateFile within user-mode ultimately translates to a IRP_MJ_CREATE request at the FSD. These entry points are also called dispatch routines. Internally the windows kernel always uses I/O Request Packets to describe any kind of request. Thus the FSD always deals with I/O Request Packets for all filesystem operation. One such information that can be found in the I/O Request Packet is the FILE_OBJECT of the corresponding file that is being operated on.

One approach to the implementation could simply change the pointer for the driver dispatch routines to point to its own thereby receiving control. However, due to the complexity of the FSD, the parameters passed to these dispatch routines are not easy to decipher. Instead, we chose to intercept at internal functions of the FSD which accept refined parameters which allow us to easily map the file object that was being manipulated. At a first glance it might seem that this method is not portable, but in our experiment we compared the two sample drivers provided in the windows ddk with that of the ntfs driver debug symbols and found that the entry points of the functionality we needed were common and took the same parameters. We intercepted the Create, Open, Delete, Close, QueryInfo and SetInfo dispatch routines in the FSD.

Registry Intercepts: Windows manages configuration information in what is called a registry. A registry is essentially a collection of system specific files that store almost every piece of configuration information that is essential to the systems. Registry is composed of hives and keys much like directory and files.

The windows registry is managed by the configuration manager which is a subsystem of the Windows kernel. The registry is stored in memory and flushed to disk periodically. However, these flushes to disk are made in a lazy fashion. which means that the copy on the memory and on the disk at any given point are not guaranteed to be exact, so we choose to intercept registry access in memory since it ensures that the value written to and read from are the same. We note that direct manipulation of the registry is not a possibility as the configuration manager opens it at start and keeps it open until Windows shuts down in exclusive mode. Further the organization of the registry is something that is not public and techniques employing such direct access is going to break in implementation. The framework intercepts the CreateKey, DeleteKey, Close, QueryKey and SetKey registry functions within the configuration manager, so that it gets control whenever the corresponding registry operation is initiated.

Unlike current approaches which employ filter drivers or system call table in order to monitor filesystem and configuration information, we employ redirection at the lowest possible level within the OS kernel or the filesystem driver itself. By obtaining control at the lowest possible level, we ensure that we always get control and can never be bypassed easily.

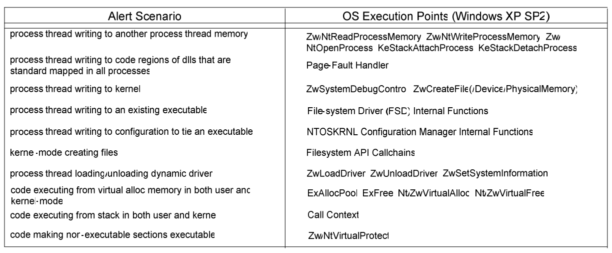
4.4 DisInfection

The implementation of Disinfection consists of the implementation of alerts and the disinfection console.

4.4.1 Alerts

Alerts are implemented by hooking various systemOS kernel points which can be used either directly or indirectly by a malware to infect a system. Figure 7 shows the scenarios and the OS execution points that are redirected by the framework for Windows XP SP2 OS in order to record alerts.

If an alert is configured to be passive, the framework simply records it and proceeds with the flow of execution. However, if the alert is configured to be active, the framework delivers the alert to



Alert Scenario	OS Execution Points (Windows XP SP2)
process thread writing to another process thread memory	Zw/NtReadProcessMemory Zw/NtWriteProcessMemory Zw/ NtOpenProcess KeStackAttachProcess KeStackDetachProcess
process thread writing to code regions of dlls that are standard mapped in all processes	Page-Fault Handler
process thread writing to kernel	ZwSystemDebugContro ZwCreateFile(/Device/PhysicalMemory)
process thread writing to an existing executable	File-system Driver (FSD) Internal Functions
process thread writing to configuration to tie an executable	NTOSKRNL Configuration Manager Internal Functions
kerne -mode creating files	Filesystem API Callchains
process thread loading/unloading dynamic driver	ZwLoadDriver ZwUnloadDriver ZwSetSystemInformation
code executing from virtual alloc memory in both user and kernel-mode	ExAllocPool ExFree Nt/ZwVirtualAlloc Nt/ZwVirtualFree
code executing from stack in both user and kerne	Call Context
code making non-executable sections executable	Zw/NtVirtualProtect

Figure 7. Alerts: Aid the user in the selection of a starting operation, whose effects are then automatically eliminated by the framework

the disinfection console (see Section 4.4.2) via an Asynchronous Procedure Call (APC) [24]. The user can then decide to continue or abort the operation. Upon the delivery of the APC, the disinfection console then enters the framework via an IOCTL and dequeues the spinlock that was held with the status of either continuing or abortting the operation. This scheme of using the APC to communicate with the recovery console ensures that, if the system entry point for an alert scenario is the FSD, no other filesystem requests are initiated until the response from the disinfection console.

4.4.2 Disinfection Console

When the user wishes to disinfect an infected system, he/she uses the disinfection console which then enters the framework into the disinfection mode. In this mode, any context switching is disabled and no operations are permitted on the system until disinfection finishes.

The disinfection begins with the construction of active and delete lists as explained in section . As the next step, the disinfection engine then sweeps through all the open file handles in the running processes and builds a list of file handles that need to be reopened when the system is switched to a different view. It then deferences all the FILE_OBJECTS associates with the file handles thereby terminating all mappings with the FSD. The handles themselves are not deleted. The recovery engine then scans the recovery and delete lists to see if any process/thread that is dependent on these lists is currently active in memory. The call-contexts are used for this purpose. The disinfection engine then forcibly tears down the process by sending it the terminate signal within kernel-mode. The recovery engine then applies the recovery and delete lists to the current view and moves the system into the desired state. Finally the recovery engine creates a new mapping for the FILE_OBJECTS and the handles in the running processes, enables interrupts and returns to the disinfection console. At this point, the system has been disinfected.

5 Evaluation

This section presents a qualitative as well as a quantitative evaluation of our framework. The qualitative evaluation shows the framework effectiveness in detecting and removing the effects of malware, while the quantitative evaluation shows the framework runtime cost to imperceptible.

Our testbed consisted of a clean system running Windows XP SP2 on an Intel 3.2 GHz processor with 1GB of system memory and 160GB hard-disk. The framework prototype was installed on the system and tested on a suite of malware consisting of: Adware/Spyware (Booked Space and Bargain Buddy), Viruses (W32/Kalb-ow, W32/Bacalid and W32/Detnat), E-mail Worms (W32/Klez) and Rootkits (Rustock and Nailuj). We also tested the above malware with two popular commercial anti-virus tools McAfee VirusScan Plus [22] and Norton Anti-virus [29] for comparison purposes on the same system configuration.

5.1 Qualitative Evaluation

We carried out the following experiment to demonstrate the effectiveness of our framework. We first took a complete snapshot of the base system with our framework installed. We then ran the malware sample and let it infect the system. We then used our framework to remove the effects of the malware. We then took a second complete snapshot of the system. We then compared the two snapshots (files, file contents in binary and configuration information) to check for any differences. We repeated the above experiment for all malware samples and for the framework and the two commercial tools.

Malware	MalTRAK				Commercial Anti-Virus 1				Commercial Anti-Virus 2			
	Configuration Objects		File/Directory Objects		Configuration Objects		File/Directory Objects		Configuration Objects		File/Directory Objects	
	Detected	Restored	Detected	Restored	Detected	Restored	Detected	Restored	Detected	Restored	Detected	Restored
Booked Space	52	52	123	123	9	9	10	10	24	24	22	22
Bargain Buddy	43	43	236	236	8	8	8	8	20	20	59	59
W32/Bacalid	0	0	328	328	0	0	328	265	0	0	328	258
W32/Detnat	12	12	152	152	2	2	152	152	8	8	152	152
W32/Kalb-ow	5	5	173	173	0	0	173	0	3	3	173	0
W32/Klez	4	4	567	567	2	2	567	544	0	0	567	544
Rustock	10	10	1	1	0	0	0	0	0	0	0	0
Nailuj	13	13	8	8	0	0	0	0	0	0	0	0

(a)

	Malware	Booked Space	Bargain Buddy	W32/Bacalid	W32/Detnat	W32/Kalb-ow	W32/Klez	Rustock	Nailuj
Commercial Anti-Virus 1	Configuration Objects	17%	19%	100%	17%	0%	50%	0%*	0%*
	File/Directory Objects	8%	4%	80.7%	100%	0%	96%	0%*	0%*
Commercial Anti-Virus 2	Configuration Objects	46%	47%	100%	67%	60%	0%	0%*	0%*
	File/Directory Objects	18%	25%	79%	100%	0%	96%	0%*	0%*
MalTRAK	Configuration Objects	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
	File/Directory Objects	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

* These malware were not detected at all

(b)

Figure 8. Comparison of MalTRAK and two popular commercial Anti-Virus tools; (a) ability to detect and restore modifications to the filesystem and configuration elements, and (b) percentage of malware effects that are removed from the system.

Figure 8a shows the number of files and configuration elements that were affected by the malware that were detected and removed by our framework with those that were detected and removed by the commercial tools. The figure shows that for all the malware our tool can successfully remove all their effects completely than the commercial tools can. These include:

Files/Configuration elements that the malware modified/completely destroyed in the system. E.g W32/Kalb-ow, W32/Bacalid, W32/Detnat and W32/Klez. Our tool was able to recover all the elements while the commercial tools could only recover a fraction. Further for malware such as W32/Kalb-ow the commerical tools could not recover any of the original elements as they were completely overwritten.

Companion/Temporary elements that the malware created in the system. Eg. Booked Space, Bargain Buddy and W32/Klez. Our tool was able to recover all the elements whereas the commercial tools overlooked a portion of such elements.

Stealth malware: Eg. Rustock and Nailuj. Our tool was able to recover all elements whereas the commercial tool could not even detect these malware once they infected the system.

Thus, on an average, the commercial tools could only restore 36% of the effects of all the malware put together. What was more surprising to us was the fact that the commercial tools claimed to have the signatures for the Rustock and Nailuj, but for some reason they were not able to detect them or recover their effects once they had infected the system. Upon further investigation, we found that these malware directly intercept at the FSD level thereby completely bypassing the filter drivers that are used by the commercial tools. Further, these malware change the results of a file/directory query to the FSD to remove all its files from the result buffer. Thus, the anti-virus tools dont see the files used by these malware which explains the observed discrepancy. However, since MalTRAK intercepts at the FSD level beforehand, it receives control at the lowest-level (since the malware intercepts chain to the original FSD functions, which in our case is redirected by MalTRAK) and is therefore still able to detect such malware and remove their effects.

5.2 Quantitative Evaluation

Our framework monitors all activity to the filesystem as well as configuration subsystems systemwide. It records information about such activity in case of modifications. It also intercepts call-chains and certain system points in order to maintain state information about the executing code and the modifications done. Figure 9 shows the overhead of the frameowork in the whole system (both time and space) for several benign and malware programs. As seen from the measurements, the overhead of our implementation is minimal in the context of both runtime latency and disk space for both filesystem intensive and non-intensive workloads. The average runtime latency is very close to the actual execution time. For most programs which create new files, there is no disk space overhead. For programs which modify existing files (W32/Kalb-ow and Untar), the disk space overhead is reasonable. Note that for W32/Kalb-ow, the runtime latency and disk space overhead is quite large when compared to other programs. This is because the W32/Kalb-ow is an overwriting virus which scans the system and recursively overwrites all possible files periodically. Given the fact that our framework is able to recover from such damaging operations, the overhead is acceptable in this case.

Program	Execution Time		Disk Space Overhead
	Normal Execution	With MalTRAK	
FreeWire	15.238s	16.588s	0 KB
Kazaa	35.237s	37.098s	0 KB
Rustock	1.212s	1.378s	0 KB
W32/Kalb.ow	8.567s	14.245s	4276 KB
Untar	1.256s	1.987s	875 KB
Cygwin Install	95.165s	98.797s	0 KB

Figure 9. Execution time and disk space overhead of MalTRAK while running benign and malware programs.

5.3 Usability

There could be situations where it might be difficult for the user to pick the appropriate restore point to recover from the infected system. When this happens the usability of our framework will suffer. However, for all the malware that were detected and disinfected by our framework, we found that the framework provided alerts which made it easy to spot the recovery point. Another situation in which the usability of the system could suffer is if a malware messes with the recovery console, making it difficult for the user to interact with our framework. While one solution would be to move the console into kernel-mode, it is very difficult to implement a user interaction due to limitations of Windows. Further, a kernel-mode malware could mess with the console in kernel-mode. Thus, the only option in this case would be boot the system using a cdrom/bootdisk and run our recovery console via the command line.

The framework alerts are by default in passive mode. This means that our framework logs possible restore points without the need for the user to intervene. However, the user has the option to enable a particular class of alert to be active. As an example, one could set a system driver installation to be an active alert showing the user whether he/she is going to be installing a privileged component. A point to note is that, a user can of course safely go ahead and install the driver without having to worry about its nature irrespective of whether the driver might be benign or malicious.

6 Discussion

6.1 Framework Security

Security of Views: The implementation of views takes advantage of the fact that under normal system operations executables are never modified and data files are normally modified within a single session. This leads to a simple and effective implementation of maintaining COWed copies of modified files.

However, an adversary may try to ddos our approach by performing multiple opens and closes resulting in many COWed copies being created. This attack can be detected by monitoring the number and frequency of such recurring accesses and by placing an alert on such recurrent accesses. A normal application by our tests should in most cases never trigger an alert. An adversary could also ddos our attack by employing moderate opens and closes on huge files. However, in our experience applications which deal with such huge files are most likely to be database oriented and in which case have some internal rollback features embedded for the data. Thus, for huge files the framework only maintains a single COWed copy. However, we can enhance our framework to support this situation by maintaining incremental changes instead of cowed copies. We leave this for future work.

Another form of attack could try and locate the framework view files and tamper with them. However, in order to do so, a malware has to go though the file system driver. Our framework intercepts access to the view files and returns values which mimic their non-existence.

Security of Mappings: Our framework can map a system operation to an executing code-stream using call-contexts. If the malware code is in the form of an executable, dll, kernel mode driver, allocated pool, the framework mappings always tie them into the parent infomation of the request to a particular file or configuration object as described in section . A malware could however exploit a vulnerability and execute code on the stack and this code could overwrite code in some other module in order to execute access to a file/ configuration object. In usermode, such modifications will be detected by the framework using COW, however in kernel-mode it is difficult. Further in user mode a malware could jump to a module from the stack in order to create a file. In this case the mapping would have no parents, and we can signify an orphan. But if there are modified modules, then we will link them into the parent. In kernel-mode normally there are no file creations done by standard modules, thus if there are no loaded modules after framework, then signify an orphan else link the loaded modules as the parent. In either case depending on where the file is created, most likely either in a system path since the malware wishes to be persistent, the framework can always issue an alert.

Security of Interceptors: The intercepts within our framework are directly within the file system driver itself. A malware could intercept within the FSD, but would still have to invoke the framework since we intercept the FSD at the lowest level where we can grab enough information about the files being manipulated. A malware could try and interpose below us, however in our opinion it is a non-trivial process and would severely be compromising the stability of the system as these internal routines and the structures they employ vary depending on the file system driver being used. A malware cannot dynamically load another FSD for the same volume as the OS will not allow it to do so. It could install a filter, but then will have to invoke our framework in the chain. A malware could overwrite the filesystem driver image in order to be activated in the next bootup, but our framework can capture such a modification and will issue an alert.

Security of Recovery: Since we keep track of all modifications to our system state and we ensure that these information cannot be tampered with, recovery will always succeed. The recovery process aborts all processes that are related to the view being switched from and hence cannot be interrupted. This also prevents the risk of reinfection.

6.2 Limitations

While our framework is capable of capturing all filesystem and configuration changes to the system, there are currently some limitations.

The framework maintains internal memory structures related to views and mappings. It periodically flushes such information to disk. However, a malware running in kernel-mode could tamper with these memory structures resulting in system instability. While, we can still restore the system to a stable and clean state on the next bootup, the tampering could lead to loss of more clean data than usual. A malware could also tamper with the recovery console (which is a user-mode interface to the user) in order to prevent restoration. However, in such situations we can always restore the system after a reboot.

The framework employs COW mechanism on the entire file in order to maintain its views. This can have severe impact on the system when dealing with big files. The performance of the framework can also be impacted with high volume of small file creations and modifications. One solution to this problem would be to employ COW at a smaller granularity (such as a multiple of 512 or 1024 bytes) . This would ensure that only parts of a file that are changed are COWed rather than the whole file thereby reducing the disk space required to maintain views.

A malware running in kernel-mode could directly access the disk bypassing the filesystem driver altogether. However, as we discussed at that level a malware will have to incorporate the complete filesystem code within itself in order to traverse the disk to create or read/write files. This is a non-trivial task. Further, accessing the disk directly could lead to filesystem inconsistencies due to the fact that the file system drivers often cache directory entry information (which describes file/folder information). The same principle holds true with modifications to the registry memory areas/disk structures as the configuration manager caches most entries which will lead to inconsitencies and system instability.

The framework mappings help in minimizing the amount of clean data that is lost, but it is still very coarse-grained. In other words, the mapping does not exactly pinpoint the originator of the operation. Doing so, requires complete control over the executing code streams and the modifications it makes which is a heavy duty process, especially for code running in kernel-mode.

7 Conclusions

We have described MalTRAK, a framework for tracking and eliminating known and unknown malware. The framework allows the user to run any program without requiring policies or rules to be places apriori, while guaranteeing the capability of restoring the system to a clean state in case of an infection. Furthermore, it does so with minimal runtime overhead and by minimizing the amount of clean data lost during disinfection. The framework achieves these goals by establishing different logical views of the system during runtime and by maintaining a relationship between the views depending upon the system operations. It can then switch to a clean system state upon infection by switching to the appropriate view before the infection took place. The framework monitors system operations at the lowest possible level ensuring that it is very difficult (almost impossible) to bypass. We implemented MalTRAK on Windows and tested our prototype on 8 real world malware and compared it with two popular commercial antivirus tools. With minimal overhead (both disk space and runtime latency) we were able to completely remove their effects on the system while the commercial tools, on an average were only able to restore 36% of all their effects put together. For one of the malware samples, the commercial tools could only detect it but could not repair any of its damage. Further, for two of the malware samples, the commercial tools were completely unable to detect or restore any of their effects.

References

[1] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the Usenix Annual Technical Conference*, 2003.

[2] M. Christodorescu, S. Jha, S. Shesia, D. Song, and R. Bryant. Semantic aware malware detection. In *Proceedings of the IEEE Symposium on SEcurity and Privacy*, 2005.

[3] M. Christodorescu and J. S. Static analysis of executables to detect malicious patterns. In *Proceedings of the USENIX Security Symposium*, 2003.

[4] F. Cohen. Operating system protection through program evolution. In *Available at (http://all.net/books/ip/evolve.html)*, 1998.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual machine logging and replay. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[6] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the USENIX Security Symposium*, 2002.

[7] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *Proceedings of the ACM Symposium on Opersting System Principles (SOSP)*, pages 163–176, 2005.

[8] A. Gostev. Malware evolution: January - july 2007. *Kaspersky Lab Report*, October 2007.

[9] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security and Applications Conference (ACSAC)*, 2006.

[10] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of USENIX Windows NT Symposium*, 1999.

[11] K. Kasslin. Kernel malware: The attack from within. *Association of Anti-virus Asia Researchers (AVAR)*, 2006.

[12] S. King and P. Chen. Backtracking intrusions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[13] S. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Usenix Annual Technical Conference*, 2005.

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *Proceedings of the Usenix Annual Technical Conference*, 2003.

[15] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior based spyware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2007.

[16] B. Krebs. Mpack exploit tool slips through security holes. *The Washington Post*, June 2007.

[17] C. Kruegel, W. Robertson, and C. Vigna. Detecting kernel level rootkits through binary analysis. In *Proceedings of the Annual Computer Security and Applications Conference*, 2004.

[18] R. Kuster. Three ways to inject your code into another process. *Code Project (http://www.codeproject.com/KB/threads/winspy.aspx)*, August 2003.

[19] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of Annual Computer Security and Applications Conference (ACSAC)*, 2003.

[20] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of ACM Conference on Computer and Communication Security*, 2003.

[21] McAfee Inc. Top 10 computer virus threats in 2007. *Rediff (http://www.rediff.com/money/2006/nov/30spec.htm)*, November 2006.

[22] McAfee Inc. Virusscan plus: Anti-virus and anti-spyware. *(http://www.mcafee.com)*, 2007.

[23] mi2g. Five solutions to the rising identity theft and malware problem. *mi2g alerts (http://www.mi2g.com/cgi/mi2g/press/240304.php)*, March 2004.

[24] Microsoft Corp. Windows asynchronous procedure calls. *MSDN (http://msdn2.microsoft.com/en-us/library/ms681951.aspx)*, 2007.

[25] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques, and case-studies. In *Technical Report UCB/CSD021175*, 2002.

[26] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions of Computer Systems (TOCS)*, volume 10(1), pages 26–52, 1992.

[27] E. Skoudis. 10 emerging malware trends for 2007. *Search Security*, January 2007.

[28] W. Sun, Z. Liang, V. Venkatakrishnan, and R. Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of Network and Distributed Systems Symposium (NDSS)*, 2005.

[29] Symantec Corp. Norton anti-virus. *(http://www.symantec.com)*, 2007.

[30] P. Szor. The art of antivirus research. *Wiley Publishers*, 2005.

[31] A. Vasudevan and R. Yerraballi. Sakthi: A retargetable dynamic framework for binary instrumentation. In *Proceedings of the Hawaii International Conference in Computer Science (HICCS)*, 2004.

[32] M. Wise. Windows xp system restore. *Microsoft Technet Library*, 2002.

[33] N. Zhu and T. C. Chiueh. Design, implementation and evaluation of repairable file service. In *International Conference on Dependable Systems and Networks (DSN)*, 2003.