Back to the Future: A Framework for Automatic Malware Removal and System Repair *

Francis Hsu Hao Chen Thomas Ristenpart† Jason Li‡ Zhendong Su
University of California, Davis
{fhsu,hchen,ristenpa,lija,su}@cs.ucdavis.edu

Abstract

Malware, software with malicious intent, has emerged as a widely-spread threat to system security. It is difficult to detect malware reliably because new and polymorphic malware programs appear frequently. It is also difficult to remove malware and repair its damage to the system because it can extensively modify a system.

We propose a novel framework for automatically removing malware from and repairing its damage to a system. The primary goal of our framework is to preserve system integrity. Our framework monitors and logs untrusted programs' operations. Using the logs, it can completely remove malware programs and their effects on the system. Our framework does not require signatures or other prior knowledge of malware behavior. We implemented this framework on Windows and evaluated it with seven spyware, trojan horses, and email worms. Comparing our tool with two popular commercial anti-malware tools, we found that our tool detected all the malware's modifications to the system detected by the commercial tools, but the commercial tools overlooked up to 97% of the modifications detected by our tool. The runtime and space overhead of our prototype tool is acceptable. Our experience suggests that this framework offers an effective new defense against malware.

1. Introduction

Malware has become an epidemic problem. A recent study showed that a significant number of computers running Windows in a major research university were infected with one or more malware programs [19]. Another recent study showed that one in three computers has malicious code on it [15]. A major reason for the malware problem is the proliferation of software applications and the diversity of their vendors. Many cutting-edge applications come from vendors with questionable reputations. For example, many P2P applications carry code that will install adware or spyware that is very difficult to remove [8].

The most common defense against malware is detection. However, since most detectors search for malicious code patterns (static signatures) of known malware, they cannot reliably detect new malware or variants of known malware (also known as polymorphic malware). Naive users ignore or disable working detection programs to install and run malware programs when trying to use applications bundled with malware. As these malware programs accumulate, the computer often becomes unusable due to slow response time, exhausted storage, and frequent application crashes. In short, even good malware detectors cannot protect the user from running malware programs.

In the case that the user cannot avoid running malware on his system, the next defense is to remove it once the user notices its adverse effect on his computer. Typically, removing a malware program involves removing all the components installed by this program and restoring all the data modified or deleted by this program. Common approaches include:

- Running an anti-malware program to remove all the components of the malware. However, because it relies on known malware signatures, this approach cannot reliably remove new or polymorphic malware, nor can it restore infected data.
- Taking periodic snapshots of the system, and restoring the infected system to the last clean snapshot. This approach will destroy all the new data created after the snapshot, even if they are clean. Although the user may avoid this problem by saving the clean data, manually determining which data are clean is laborious and unreliable.

*This research is supported in part by Intel IT Research, and by the I3P and the Office of Science and Technology at the Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Office of Science and Technology.

†Author is currently at University of California, San Diego. Email: tristenp@cs.ucsd.edu

‡Author is currently at Microsoft Corporation. Email: jasonli@microsoft.com

- Formatting the disk and reinstalling the operating system. This drastic approach will destroy all the user data and configurations. Unfortunately, since most other approaches fail to remove all the components of the malware program, this approach is often advised and followed.

These problems call for a better approach, one that can remove all the components of both known and unknown malware, that can restore data infected by malware while preserving clean data, and that requires minimal user intervention. We introduce *Back to the Future*, a framework for achieving these goals. The framework monitors and logs operations of untrusted programs designated by the user, and can remove all the components of the untrusted programs and restore the infected data at the user's request. In other words, this framework allows the user to run untrusted programs without compromising the integrity of the system. If an untrusted program turns out to be spyware, the framework can remove all the components of the malware automatically and reliably; if the untrusted program turns out to be a virus, the framework can also restore all the infected files automatically. We name this framework *Back to the Future* because conceptually we have first rolled back the system to a prior good state. From there, we then bring only the trusted processes back to their pre-recovery state (for the prior good state this is the future).

The primary security goal of our framework is *integrity*: we want to preserve the integrity of the system while the user is running malware programs. In some cases, our framework can also provide availability: by completely removing malware from the system, it will free the resources usurped by the malware. Our framework does not aim to provide confidentiality. However, if the user can indicate confidential information on his system, our framework can incorporate this information and provide confidentiality. Furthermore, in the process of preserving system integrity we may stop running malware before it discloses confidential information. Our framework may seem similar to sandboxing environments; however, unlike a typical sandboxing environment, our framework does not require system or application specific rules about what operations are allowed (see Section 6 for further discussion).

Our framework monitors untrusted processes, and removes them and their effects on the system automatically at the user's request. However, our framework needs the user to decide which programs are trusted and which are untrusted. On the surface, this requirement seems as difficult as malware detection, but in fact, our framework only expects the user to evaluate the trustworthiness of a program conservatively: when in doubt, the user should consider the program as untrusted. In practice, there are often sound heuristics for deciding if a program is trusted. It is reasonable to consider programs from reliable sources as trusted, such as all pre-installed applications on a new computer from a reputable vendor. There is no harm in misclassifying a non-malware program as untrusted, except for incurring some performance penalty. (We will discuss performance issues in Section 4.)

We summarize the major contributions of our paper:

- We propose a new framework for preserving system integrity while allowing the user to run untrusted programs. The framework monitors and logs the operation of untrusted programs, and uses these logs for removing the untrusted programs and their effects completely and automatically. Since this framework does not need any prior knowledge about the untrusted program, it can defend against both known and unknown malware.

- Our framework provides a transparent environment for running both trusted and untrusted programs. The user does not need to modify any existing programs. No program should notice that it is running in our framework.

- We have implemented a prototype of our framework on Windows, where the threat of malware is greatest, and evaluated it with seven spyware, trojan horses, and email worms. Comparing our tool with two popular commercial anti-malware tools, we found that our tool detected all the malware's modifications to the system detected by the commercial tools, but the commercial tools overlooked up to 97% of the modifications detected by our tool.

2. Framework

2.1. Overview

Figure 1 illustrates the three components of Back to the Future: a monitor, a logger, and a recovery agent. The monitor intercepts each monitored process's read and write operations. The logger records some write operations of the untrusted processes. When the monitor determines that an untrusted process may harm a trusted process, it invokes the recovery agent to restore system integrity.

Our framework needs to solve two challenges. First, how does it determine when an untrusted program may violate the integrity of the system? Second, how does it remove all the effects of an untrusted program? Intuitively, after recovery, the system should look as if only the trusted applications have run, and the untrusted applications have never been installed or run. The next two sections describe our solutions to these two challenges.

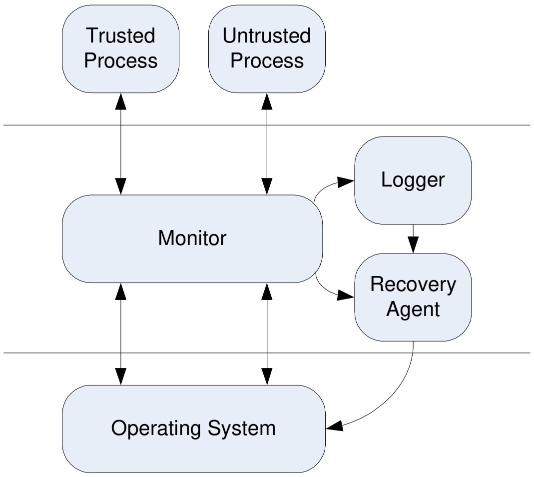

Figure 1. Framework for monitoring, logging and recovery.

2.2. System Integrity

This section defines the notion of system integrity, describes a criterion for checking when an untrusted program may violate system integrity, and discusses how to preserve system integrity.

2.2.1. Integrity Model

We start with Biba's integrity model [3], which says that no subject can read objects of lower integrity levels, and no subject can write objects of higher integrity levels. Our framework defines two integrity levels: *trusted* and *untrusted*. Applying Biba's model, our framework would require that trusted processes should not read untrusted data, and untrusted processes should not write trusted data.

Strictly following Biba's model, however, considerably limits the user's ability to run untrusted programs. For example, the framework would have to stop an untrusted process immediately when the process tries to overwrite trusted data. If no trusted process will ever read this data again (e.g., temporary scratch data), stopping the untrusted process is unnecessary. Even if some trusted process will read this data, the framework does not have to intervene until just before the read operation happens.

Hence, we adopt a relaxed integrity model. Our model only requires that no trusted process should read untrusted data, but untrusted processes can freely write (or overwrite) any data they desire. This model can be viewed as a *lazy* Biba's model: it does not enforce integrity until the point where untrusted data could flow into trusted processes. The laziness in our model allows the user to run more untrusted applications without interference from the integrity policy.

2.2.2. Preserving System Integrity

To preserve system integrity, the framework must intervene when a trusted process is about to read untrusted data. We argue that a good intervening approach should satisfy the following properties:

- *Preserving the consistency of processes*: The approach should preserve the consistency of both trusted and untrusted processes. This means that if the approach allows a process to continue, it should not change the process's behavior. For example, the approach should not selectively deny certain operations of the process.
- *Allowing processes to run as long as possible*: The approach should allow a process to run as long as possible until it cannot preserve system integrity or the consistency of some processes.

We propose the following options for preserving system integrity:

- *Deny the operation*: This option preserves system integrity by denying this read operation. To preserve the consistency of the trusted process that issued the read operation, the framework must also terminate the trusted process.
- *Allow the operation*: We can allow this read operation but still preserve system integrity with the following approaches:
  - *Terminate the untrusted process*: We terminate the untrusted process that has written the untrusted data, and restore the old data at the same location. If the restored data is still untrusted, we terminate the process that had written it, restore the old data, and repeat this procedure until the data that we have restored is trusted. This solution preserves system integrity by replacing untrusted data with trusted data.
  - *Mark the trusted process as untrusted*: We begin to treat the trusted process as untrusted. Notice, however, that we do not need to remove the data written by this process in the past. Since this process is now untrusted, we allow the read operation to continue, as untrusted processes can read any data. This solution preserves system integrity by reducing the set of trusted processes.

Sometimes we may want to preserve good effects of untrusted applications. Under such cases, we can mark the untrusted data as trusted and let the read operation continue. As an example, consider media files downloaded by malware-laden P2P applications. If the user is confident that the media files will not affect trusted applications, we can allow a trusted application, say a media player, to play these media files.

2.3. System Recovery

This section describes how the framework removes all the effects of untrusted programs on the system.

2.3.1. Basic Approach for System Recovery

We first describe a basic approach for system recovery that is conceptually simple and serves as a reference for reasoning about the correctness of a more efficient but complex approach. During monitoring, the framework logs all the operations of both trusted and untrusted processes; during recovery, the framework first reverses all the logged operations of both trusted and untrusted processes reverse-chronologically, and then reapplies all the logged operations of only the trusted processes chronologically.

We next elaborate on this approach. Given a definition of the state of a system (e.g., the state consists of the file system and the registry), we can divide the operations of all the processes into two categories: read operations (which do not change the system state) and write operations (which do change the system state). Since our goal is to remove all the effects of the untrusted processes on the system, the framework needs to log only the write operations. This approach requires the framework to log the write operations of both trusted and untrusted processes. Moreover, since the framework needs to undo the write operations during the recovery phase, it needs to log the old data overwritten by each write operation during the monitoring phase.

One can argue that after recovery this basic approach brings the system to a state that looks as if the untrusted processes have never run. However, this approach is inefficient, because during recovery it first undoes each write operation by the trusted processes and later redoes the same operation. For most write operations, undoing them followed by redoing them will have no net effect. We could save time by avoiding undoing and redoing these write operations, and save space by not logging these write operations.

2.3.2. Refined Approach for System Recovery

We refine the basic approach by avoiding the recovery operations with no net effect. We motivate the refined approach by two examples, where a trusted process T and an untrusted process U write to the same data location:

- **Example 1**: T writes before U writes. During recovery, the framework only needs to undo U's write operation; it does not need to undo and then to redo T's write operation, and it does not need to log this operation during monitoring.
- **Example 2**. U writes before T writes. During recovery, the framework does not need to undo either U's write or T's write, because T's trusted data has overwritten U's untrusted data.

These two examples suggest that we can detect unnecessary recovery operations by tracking the order in which trusted and untrusted processes write to the same data location. In fact, it suffices to track whether each location contains trusted or untrusted data. In this refined approach, during monitoring:

- When a trusted process writes to a data location, mark the new data in the location as trusted.
- When an untrusted process writes to a location:
  - If the location contains trusted data, log this write operation, save the old data, and mark the new data in this location as untrusted.
  - If the location contains untrusted data, do nothing.
  - If the location contains no data, log this write operation, and mark the new data in this location as untrusted.

During recovery, the framework examines each logged write operation reverse-chronologically. Recall that the framework only logs write operations by untrusted processes. For each logged write operation, the recovery agents restores the old data from the log only if the location currently contains untrusted data.

Proof of Correctness We prove that this refined approach achieves the same result as the basic approach. Given a data location, let the entire sequence of write operations at this location before system recovery be $O_1, \ldots, O_n$. We consider two cases, depending on whether the last operation O_n is from a trusted or an untrusted process:

- **Case 1**: The last write operation O_n is from a trusted process. Using the basic approach, the framework will first undo $O_n, \ldots, O_1$, and then redo only the operations in $O_1, \ldots, O_n$ that are from trusted processes, in that order. Since O_n is from a trusted process and is the last operation performed during recovery, this location will contain the data written by O_n after recovery. Using the refined approach, the framework will notice that the location already contains trusted data, so it will do nothing on this location. Since before recovery this location already contains data written by O_n, after recovery using the refined approach, this location will contain the same data as when using the basic approach.

- **Case 2**: The last write operation O_n is from an untrusted process. Let O_t be the last write operation by a trusted process in this sequence. Now the sequence is $O_1, \ldots, O_t, O_{t+1}, \ldots, O_n$ where all $O_{t+1}, \ldots, O_n$ are from untrusted processes. Using the basic approach, the framework will first undo $O_n, \ldots, O_1$, and then redo only the operations in $O_1, \ldots, O_t$ that are from trusted processes. Since O_t is from a trusted process, it will be the last operation that the framework redoes on this location, so this location will contain the data written by O_t after recovery. Using the refined approach, during monitoring the framework will log O_{t+1}, but will not log any operation after O_{t+1}, because O_{t+1} writes untrusted data into this location. During recovery, the framework will first undo O_{t+1} by replacing the data in this location with the data that was in this location before O_{t+1}, which was exactly the data written by O_t. After that, this location contains trusted data because O_t is from a trusted process, so the framework will not change the data in this location any more. Therefore, both the basic and the refined approach restore the same data into this location.

3. Implementation

To evaluate our framework, we have developed a prototype implementation for the Windows XP operating system. The implementation consists of the three essential components of the framework: a monitor, logger, and recovery agent. Our monitor is a Windows kernel driver that hooks relevant system services and can therefore capture most of the interactions between user processes and the operating system. The logger and recovery agent are user applications that interact with the driver.

3.1. Monitoring

3.1.1. System Service Hooking

In Windows NT 4, 2000, and XP, user applications rely on the interface exposed from a set of libraries, such as *kernel32.dll* and *user32.dll*, to access operating system services. This interface is known as the *Win32 API*. Applications may also call function in *ntdll.dll* known as the Native API [14]. The Native API functions perform system calls in order to have the kernel provide the requested service.

When the kernel traps system service interrupts, it uses a unique identifier found in the call to look up a function pointer in the service dispatch table. Kernel drivers can modify this table to wrap system services with arbitrary code. This technique, known as API hooking, allows us to intercept all the system service calls made by any process [13, 21]. Our framework hooks the system services that access the file system and registry, and those that create new processes.

3.1.2. Tracking Untrusted Data

A significant component of the monitor tracks which data are untrusted as both trusted and untrusted processes execute, because our integrity model requires that no trusted process should read untrusted data. In the implementation of this component, two key issues are granularity and metadata: to what granularity does this component track untrusted data, and how is the trustworthiness recorded?

To determine the best granularity, we need to strike a balance between precision and overhead. For the registry, we chose a granularity of one value, because most registry values are small. On the other hand, files can become very large, so using a file-level granularity would be too coarse-grained. Thus, we track the ranges of untrusted data in each file.

Our implementation maintains a table of all files and registry entries that contain untrusted values. For each file, an associated data structure describes which ranges in this file contain untrusted data. The monitor uses this table to determine if a trusted process will read untrusted data. The logger (Section 3.2) and the recovery agent (Section 3.3) will also use this table. Table 1 summarizes the actions taken by the monitor for various operations.

In addition to tracking untrusted data, the monitor also tracks and monitors processes spawned by untrusted processes, which are also considered as untrusted.

3.2. Logging

The second component of the implementation is logging. During recovery, the framework uses logged information to remove malware programs and to restore infected data on the system. As discussed in Section 2.3, the framework only needs to log write operations from untrusted processes. The monitor makes appropriate backups and forwards information to the logger.

3.3 Recovery

The final portion of the implementation is recovery. Given the data created by the logging mechanism, the recovery tool will roll back the effects of each entry until the desired system state is reached. The tool also uses trustworthiness information about data from the monitor to determine what portions of data it should restore.

4. Experiments

We evaluated our tool's effectiveness in detecting malware, removing malware, and restoring infected data, and

Process	Process's Operation	Old Status of Target Data	Monitor's Action
Trusted	Delete file	Trusted	Allow
		Untrusted	Remove file from watch list
	Write data	Trusted	Allow
		Untrusted	Mark new data as trusted
	Read data	Trusted	Allow
		Untrusted	Warn integrity violation
	Create process	Any	Allow
Untrusted	Delete file	Trusted	Mark file as deleted
		Untrusted	Mark file as deleted
	Write data	Trusted	Mark new data as untrusted
		Untrusted	Allow
	Read data	Trusted	Allow
		Untrusted	Allow
	Create process	Any	Monitor new process as untrusted

Table 1. Tracking untrusted data and new processes.

its performance during monitoring and recovery. We tested our tool on a suite of malware programs consisting of:

- *Adware and spyware*: *eZula*, *Gator*, and *BonziBuddy*. They are normally bundled with other benign programs, such as a P2P application. When the user installs the benign programs, the installers furtively install these malware programs.

- *Trojan horse*: *NetBus*. Trojan horses are normally packaged with innocuous decoy programs. When the decoy programs are executed, they install and run the bundled Trojan horses. NetBus configures the system to allow remote access and control.

- *Email worms*: *Netsky* and *Beagle*. Email worms depend on deceived users to execute email attachments to install and propagate the worms. *Netsky* and *Beagle* caused two major email worm outbreaks in 2004.

- *Hybrid malware*: *Happy99*. *Happy99* acts both as a trojan horse and a worm, since it purports to be an entertaining screen saver, and it propagates via email behind the scenes.

4.1. Recovery

During recovery, our tool should remove all the files and registry entries installed by the malware, and restore the original data in the infected files and registry entries. We evaluated the effectiveness of our tools's recovery function by comparing it with two popular commercial tools: Spybot[2] and Symantec Norton AntiVirus[1]. Spybot handles *eZula*, *Gator*, and *BonziBuddy*, and Symantec Norton Anti-Virus handles the rest of the malware programs used in our experiments. We compared them in two experiments:

- First experiment: after running a malware program, we first invoke the recovery function of our tool, and then we run a commercial tool to detect any residual traces of this malware.

- Second experiment: after running a malware program, we first run a commercial tool to detect and remove the program, and then we examine whether the commercial tool has removed all the files and registry entries created by the malware program as logged by our tool.

In the first experiment, for each malware program, we found that neither commercial tool could detect the malware after we ran our tool to remove it. Since both commercial tools could detect the malware before we removed it using our tool, we conclude that our tool has removed the malware to the satisfaction of the commercial tools. In the second experiment, we found that the commercial tools failed to remove all the files and registry entries that the malware programs had created. Table 2 compares the number of files and registry entries modified by the malware programs that were detected by our tool with those that were detected by the commercial tools. The table shows that, for some malware, our tool can identify more files and registry entries modified by the malware than commercial tools can. These include:

- Original files and registry keys that malware has deleted from the system. E.g., W32.Netsky deleted a registry key associated with a component of Microsoft Internet Explorer.

- Temporary files created by malware during its installation. E.g., eZula created temporary files while it was retrieving data from the network. These files were not deleted even after the commercial tool claimed to have removed eZula.

- Modifications made by other system components on behalf of the malware. E.g., Bonzi Buddy asked Microsoft Agent Services to modify the file system, but the commercial tool failed to detect the modified files.

4.2. Usability

Our tool monitors read and write operations of both trusted and untrusted processes. When a trusted process reads data that were written by an untrusted process, our tool will stop the process and alert the user. If this alert never happens, our tool will allow an untrusted process to run to completion (the user can still use our tool to remove the program and its effects at any later time). However, if this alert happens often, the usability of our tool will suffer, because each alert requires user intervention.

We never saw an alert when we used our tool to run the seven malware programs mentioned earlier. Examining of the logs carefully, we found that NetBus, W32.Beagle.AC, and W32.Netsky should have triggered alerts. They all write to the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, which is read by Windows during its boot. This modified key allows the malware to survive a system reboot, because the system will automatically restart all the programs listed in this registry key. These malware programs violate our integrity model, because they write untrusted data into this registry key, and the system will read these untrusted data during the next reboot. Our framework would detect this violation, if our monitor driver were loaded early in the boot sequence. However, due to limitations in Windows, no user visible notification could be given at this point of detection, and so the only allowable option would be to restore the previous registry key value.

4.3. Performance

Our tool monitors the execution of all the processes on the system. It intercepts and optionally logs all the system services from untrusted processes, and it also monitors trusted processes to prevent them from reading untrusted data. However, most system service calls pass through our monitor very quickly, and only the calls that modify the system state (such as the file system and the registry) may notice delays.

The timings in Table 3 reveal that while the overhead of our implementation does increase execution time for the tasks, the effect is reasonable when compared with the resource usage of a commercial anti-spyware of anti-virus program. Moreover, the performance numbers for the installers and unzip should be interpreted as a stress test of our system since they mainly consist of file operations. All measurements were conducted on an Intel Pentium 4 2GHz desktop with 256 MB RAM and a 7200rpm IDE hard disk running Windows XP Workstation SP1.

5. Discussion

5.1. Security of the Framework

Security Goals Security has three main goals: confidentiality, integrity, and availability [3]. Our framework focuses on maintaining integrity: it allows the user to run untrusted programs without compromising system integrity, and it can remove the untrusted programs and all their effects on the system completely and automatically. Our framework does not ensure availability directly, because it does not control resource usage by untrusted programs; however, since our framework can remove untrusted programs and their effects on the system, it provides availability indirectly. Our framework does not provide confidentiality, since it does not prevent untrusted programs from reading confidential information, nor does it monitor outgoing network traffic. As we discussed in Section 1, once the user starts to run untrusted programs, it is very difficult to maintain confidentiality in a usable way. However, we can enhance our framework to provide confidentiality. If the user can indicate what information is confidential on his system, we can incorporate this information by disallowing untrusted applications from reading confidential information. We leave this for future work.

Security of the Logging Mechanism Our framework logs write operations by untrusted processes so that it can reverse these operations in the future. An adversarial process may try to DOS attack our logging system by making numerous write operations. However, our system does not log each write operation by untrusted processes; it only logs those write operations that replace trusted data. More specifically, we divide the write operations by untrusted processes into three categories:

- The operation replaces trusted data. Our system logs this operation and the old data.
- The operation replaces untrusted data (i.e., data written earlier by an untrusted process). Our system logs nothing.
- The operation writes new data. Our system only logs that this operation took place.

The log size in the first case may be large because of potentially large old data, the log size in the third case is small, and the log size for the second case is zero. Therefore, an adversarial untrusted process cannot effectively DOS attack our logging system by writing a large amount of new data,

Malware	Our Tool		Commercial Tool			
	Detected Modifications		Detected Modifications		False Negative	
	File	Registry Key	File	Registry Key	File	Registry Key
eZula	242	195	42	61	83%	69%
Gator	385	129	151	4	61%	97%
BonziBuddy	112	2135	24	59	79%	97%
NetBus	2	1	2	1	0%	0%
Happy99.Worm	2	0	2	0	0%	0%
W32.Beagle.AC	44	1	44	1	0%	0%
W32.Netsky	336	8	330	1	2%	88%

Table 2. Comparison of our tool and commercial tools' ability to detect files and registry keys modified by malware.

Program	CPU Time			Log Size
	Not monitored	Monitored as trusted	Monitored as untrusted	
eZula installer	3.953s	4.516s	6.338s	4959 KB
Kazaa installer	48.965s	59.824s	101.466s	12552 KB
Happy99.Worm	4.858s	4.963s	4.937s	6 KB
unzip (5MB file)	0.535s	0.666s	1.013s	336 KB

Table 3. CPU time and disk space overhead of our tool while running benign and malware programs.

or repeatedly overwriting the same location. The only effective attack is to overwrite a large amount of trusted data, which we can deal with by limiting the maximum amount of data that an untrusted process may overwrite.

Security of the Dichotomy of Trustworthiness We assume that once the user considers a process trusted, it remains trusted until the user explicitly reclassifies it as untrusted. This ignores the possibility that a trusted but vulnerable process may become untrusted because malicious code has been injected into it.

5.2. Security of the Implementation

We discuss the security of our prototype implemented on Windows:

- *Read and write operations*: Our current prototype only considers read and write operations on the Windows registry and on the file system. It considers some IPCs mechanisms, such as communication through named pipes as combined read and write operations. Therefore, when an untrusted program sends a message to a trusted program, this message passing violates the integrity model. However, in our current implementation we do not monitor all IPC mechanisms including shared memory and Windows message passing. Since we cannot easily monitor device drivers installed by untrusted programs, we consider their installation as violating the integrity property. We also assume, optimistically, that after an untrusted program writes data to the network, the data will not be read by some trusted programs from the network later; therefore, we do not monitor read or write operation on the network.

- *Security of the monitoring mechanism*: The principle of complete mediation requires that untrusted programs should be unable to attack or circumvent the monitoring mechanism [18]. We install our monitor as a kernel driver before we run untrusted programs. Therefore, our monitor can intercept and control all the API calls made by untrusted programs from the user space. Our prototype treats all the processes spawned by untrusted processes as untrusted and transitively monitors the spawned processes. Therefore, we believe that our monitor is secure from tampering or circumvention by user-level processes. While our prototype can prohibit untrusted programs from installing kernel drivers by standard means, we do not prevent attacks on trusted processes that may install rootkits on the system.

- *Security of the logging mechanism*: We have discussed the security of the logging mechanism of the framework in Section 5.1. In the implementation, we need to ensure that no untrusted process can tamper with the logs. Since our framework hooks into all the API calls that access the file system, it protects the logs by denying access to it from all except the logging process.

- *Security of the recovery mechanism*: Since our framework maintains system integrity, recovery can always succeed. In particular, before the monitor begins recovery, it aborts the untrusted process (and any process spawned by it). Therefore, the process cannot interfere directly with the recovery mechanism.

6. Related Work

SEE [20, 12], proposed the idea of using *one-way isolation* to create a safe execution environment. Untrusted programs modify a separate temporary copy of the file system rather than the original. SEE allows the user the option to commit these changes to the original file system once the untrusted program finishes. We view SEE as a dual to our approach: SEE allows the untrusted programs to run to completion, but may not be able to commit some data back to the original file system. Our framework allows untrusted programs to write to the file system immediately, but our framework may prohibit some untrusted programs from running to completion.

Goel et al. designed a system to recover a file system after an intrusion is detected. The Taser [7] intrusion recovery system logs all process, file and network operations. It can then use this audit log to determine the resultant file system modifications after an intrusion. Once a compromised process is flagged by IDS network activity logs or filesystem changes, all changes to the file system depending on that process can be reversed. The dependency is derived from information flow between processes and files by system calls. While Taser has similar logging and recovery components compared with our approach, the main difference is the timeliness of response. Our approach can act immediately when an untrusted process taints a trusted one since we monitor the actual kernel objects in the operating system. Taser identifies malicious behavior by an IDS, and only acts after the IDS signals an intrusion. Since an IDS may have false negatives, Taser may never respond to the tainting of trusted processes. Moreover, if an IDS does not respond immediately after an intrusion happens, Taser would need to reverse all the operations of legitimate processes from the time of intrusion to the time of intrusion detection, resulting in the loss of any work done by the legitimate processes.

Our work is also related to the general isolation strategy with virtual machines, which provide an effective, reliable mechanism for isolating untrusted applications. King et al. added support for virtual machines monitors into the Linux kernel for achieving high performance [11]. However, using virtual machines to execute untrusted programs has its shortcomings. First, untrusted programs running inside a virtual machine cannot access resources created by programs running outside the virtual machine, which may break many programs. Second, virtual machines are expensive. Configuring each untrusted program to run in its own virtual machine with a complete operating system requires considerable amounts of system resources and human time.

Our framework is inspired by recovery-oriented computing (ROC), which is a framework for recovering from system component failure and operator errors [16, 4]. It contains three stages: rewind, repair, and replay. Its threat model is that any component in the system may fail, and that the operator may make a mistake at any time. Since our goal is to run untrusted programs safely, we need a different threat model: we assume that most applications on the system are trustworthy, so we can focus on monitoring and logging a few untrusted applications. Therefore, our framework has a much smaller overhead for logging and recovery. Our framework also avoids possibly expensive snapshots required in ROC.

Logging has been used for replaying system events. ReVirt uses logging for intrusion detection. It runs applications inside a virtual machine and logs their events. Then, it analyzes intrusions by replaying the logged events [6]. King et al. uses logging for debugging operating systems [10]. They run an operating system inside a virtual machine, log all its events, and use the logs to debug the operating system. We use logging for a different purpose: we want to recover the system to a safe state, rather than replay the events. This difference requires that we design our logging system differently. We do not need to take a snapshot of the system; we just log all the events. During recovery, we start from the current state of the system and undo each offending event in the reverse chronological order. In this approach, we have avoided taking a system snapshot, which may be very expensive. Logging has also been used for system recovery. A log-structured file system [17] takes this idea even further: the entire file system is in a log-like structure, which speeds up both file writing and crash recovery. It influenced the design of file system recovery in our framework.

Reparable file service (RFS) [22] uses a similar idea of logging and recovery to repair compromised network file servers, such as NFS servers. It interposes a RFS server between the NFS server and clients for logging file update operations, and these logs can be used later for rolling back these operations. It is used for a different purpose from that of our approach, and as such, it is more complicated: it requires modifying all the NFS clients as well as interposing the RFS server between the NFS server and its clients.

Our logging mechanism employs a simple tainting analysis to track trustworthiness of data. Similar ideas have been used for many other purposes. Chow et al. proposed to use whole-system simulation with tainting analysis to analyze how sensitive data are handled in large programs [5]. Newsome et al. used dynamic taint analysis for automatic detection of overwrite attacks in processes. BackTracker [9] identified automatically potential sequences of steps that occurred in an intrusion. Starting with a single detection point, it identified files and processes that could have affected that detection point. In comparison, our framework tracks the propagation of untrusted data for preserving system integrity and removing malware.

7. Conclusion

We have described Back to the Future, a novel framework for automatically removing malware and repairing its damage to the system. The framework preserves system integrity while the user is running untrusted programs, and allows untrusted programs to run as long as possible until they may harm trusted programs. The framework achieves these goals by monitoring untrusted programs, logging their operations, and using the logs to remove malware and to restore infected data. We implemented this framework on Windows and tested our prototype on real spyware, adware, Trojan horses, and email worms. With acceptable runtime and storage overhead, we detected all the malware's modifications found by commercial tools, while the commercial tools overlooked up to 97% of the modifications found by our tool.

Acknowledgments

We thank Hong Li at Intel IT Research for helpful discussions.

References

[1] Norton AntiVirus. `http://www.symantec.com/avcenter`.

[2] Spybot. `http://www.safer-networking.org/en`.

[3] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

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

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[6] 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 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[7] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[8] E. L. Howes. Anti-spyware testing. `http://www.spywarewarrior.com/`, 2004.

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

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

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

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

[13] Mark Russinovich and Bryce Cogswell. Description of regmon tool. `http://www.sysinternals.com/ntw2k/source/regmon.shtml`.

[14] G. Nebbett. *Windows NT/2000 Native API Reference*. Que, 2000.

[15] One in Three Computers Has Malicious Code. `http://www.marketingvox.com/archives/2004/10/06/one_in_three_computers_has_malicious_code`, 2004.

[16] 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. Technical Report UCB//CSD-02-1175,, UC Berkeley Computer Science, 2002.

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

[18] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 17(7), 1974.

[19] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the First Symposium on Networked Systems Design and Implementation - NDSI'04*, Mar. 2004.

[20] 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.

[21] Sven B. Shcreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*, volume 1. Addison-Wesley, Upper Saddle River, NJ, 1st edition, 2001.

[22] N. Zhu and T.-C. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)*, 2003.