Manitou: A Layer-Below Approach to Fighting Malware

Lionel Litty
Department of Computer Science
University of Toronto
llitty@cs.toronto.edu

David Lie
Department of Electrical and Computer Engineering
University of Toronto
lie@eecg.toronto.edu

ABSTRACT

Unbeknownst to many computer users, their machines are running malware. Others are aware that strange software inhabits their machine, but cannot get rid of it. In this paper, we present *Manitou*, a system that provides users with the ability to assign, track and revoke execution privileges for code, regardless of the integrity and type of operating system the machine is using.

Manitou is implemented within a hypervisor and uses the per-page permission bits to ensure that any code contained in an executable page corresponds to authorized code. Manitou authenticates code by taking a cryptographic hash of the content of a page right before executing code contained in that page. Our system guarantees that only authorized code can be run on the system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Human Factors

Keywords

Manitou, malware removal, malware protection, security architecture

1. INTRODUCTION

In recent years, many computer users have had to increasingly deal with undesirable software installed on their home or even work computers. This software, referred to as *malware*, performs a variety of nefarious deeds: it can leak credentials or private information, pester the user with ads, or use the computer's resources in a botnet. These botnets are then used to further disrupt the Internet by sending spam or performing distributed denial-of-service (DDoS) attacks. To make matters worse, the quality of malware code is often low, impacting the reliability of the whole system. Alarmingly, recent data has shown that millions of computers are now running some form of malware [7].

While some malware installs itself by exploiting vulnerabilities in applications or the operating system, in many cases, the malware comes bundled with enticing software such as games or screensavers, and is unwittingly installed by users themselves [8]. Unfortunately, once malware infects a system, it can be difficult to remove. A senior Microsoft employee has even acknowledged that in some cases removal is impossible and advises customers to reinstall their machines from scratch [9].

We believe that operating systems currently fail users by not giving them adequate control over what software is run on their computer. They rely on the unreasonable assumption that users do not make mistakes. Thus, when users do make mistakes, they provide no mechanism to revoke the execution of undesirable code, and as a result offer little recourse for users who were tricked into installing software they did not really want. In addition, because most operating systems allow code to be loaded in the kernel, malware is able to modify the privileged code base, wedging itself deeper into the system and defeating malware detection and removal tools. We feel that it is malware that deliberately targets the privileged code in the kernel, which represents the gravest danger to users.

In this paper, we present a mechanism that enables users to reliably identify the code running on their machine, as well as always be able to revoke execution privileges for any application, even if their machine has been infected by malware. Motivated by the observation that malware is able to tamper with the operating system, we advocate an OS-independent approach to controlling the software that runs on a system. We show that this goal is not unrealistic by outlining a system called Manitou, which runs on current commodity hardware and operating systems.

Manitou utilizes per-page permission bits, which are available on current commodity hardware. Manitou is located in a hypervisor and uses these bits to ensure that any code contained in an executable page corresponds to authorized code. An application becomes authorized when the user adds hashes of its code pages to a list that represents all authorized code for the system. By restricting the execution of code to that which is present in the list authorized by the user, Manitou can guarantee that only binary code explicitly blessed by the user can run. In addition, Manitou permits the user to revoke or suspend an application's execution rights at any time by removing its hashes from the list. The user maintains this list via a channel with Manitou, which installed software cannot tamper with.

We begin by outlining the mechanisms we use to identify and enforce program execution control in Sections 2 and 3. Then, we discuss how users can install new software on their machines in Section 4. We outline preliminary results in Section 5. We examine the current limitations of our work in Section 6 and we finish with related work and conclusions in Sections 7 and 8.

2. IDENTIFYING CODE

Ideally, computer systems should make decisions about which programs are allowed to execute based on the actions each program will take. Unfortunately, any policy based on such a requirement is untenable because trying to determine if a program will perform malicious actions is a generally undecidable problem[1]. Thus, a more practical alternative is to use the instructions that make up the program to authenticate it. This method is in common use – for years, cryptographic hashes have been used to protect the integrity of program images from corruption and tampering.

The key challenge Manitou addresses is to use this simple mechanism to ensure that no code can ever be executed without first having had a hash of its image authenticated. By identifying programs just before execution, Manitou avoids several problems with identifying programs by examining their file system image. For example, once the malware is resident in memory, it can remove any evidence of itself from the file system by deleting images of its executable or undoing any modifications it has made to other programs. By doing this, one might think that such malware could simply be removed by a reboot. However, such malware typically re-installs itself on the file system again right before system shutdown, after all security software has been turned off by the OS, and thus survives the reboot [6]. Such malware will evade detection techniques that rely on periodic checking of integrity.

In other cases, malware operates by injecting code into other running programs. By exploiting code injection bugs or abusing privileges that enable them to load code into a kernel, malware can execute by piggy-backing on programs whose file system images were authenticated as correct when they were loaded in memory. In contrast, Manitou avoids all these problems by validating code just before execution. Regardless of how the attacker tries to hide the presence of their malware, Manitou will always be able to detect the presence of malware if and when it is executed.

3. ENFORCEMENT MECHANISM

Aside from being able to enforce control over the execution of code on the machine, we also want to make Manitou independent of the OS; both in the sense that Manitou should not depend on the OS functioning correctly, since malware frequently targets the OS kernel, and that Manitou should work regardless of what OS the user has installed. To illustrate how Manitou is able to do this, we first assume the existence of a list of hashes that corresponds to authenticated executable code. In this section, we describe how Manitou efficiently prevents code that is not on this list from running. We defer the discussion on how Manitou constructs and maintains this list to the next section.

To efficiently identify and authenticate code just before execution, Manitou leverages two features currently found on commodity hardware: paged memory, and per-page executable and writable bits. These features are present on most major processor architectures, including Intel and AMD processors, for which per-page executable bits have recently been added. Manitou verifies each code page of a program individually instead of verifying the entire executable file. This means that Manitou does not have to be aware of the semantics of the file system the OS is using. As a result, Manitou can be implemented in a hypervisor, which is a software layer implemented between the operating system and the processor hardware. Hypervisors have long been an attractive tool for implementing system security because of the narrow interface they export and their simple implementation, which makes them less likely to contain vulnerabilities [4]. Because Manitou operates below the OS, its operation is independent of the OS, eliminating the need for OS support or modifications. The hypervisor also isolates Manitou from the OS, making it impervious to OS bugs and misconfigurations, and providing it with secure storage that is not accessible to the OS.

Manitou retains control of the page tables used by the hardware to translate virtual addresses generated by the OS and programs into physical addresses. Per-page executable bits in these page tables are set by Manitou to indicate to the processor which pages in memory may contain executable code. If any program (including the OS) tries to execute code from a page that Manitou has not marked executable, the processor will notify Manitou with a page fault. At this point, Manitou computes a hash of the page that caused the fault and searches for it in its list of authorized hashes. If the hash is found, the executable bit for the page is set, and the code is allowed to execute. Otherwise, an illegal-instruction exception is forwarded to the OS, causing it to terminate the unauthorized program.

To make sure that code is not modified once it has been allowed to run, Manitou enforces a simple rule across all virtual mappings of a physical page: a physical page can either be executable or writable, but never both. To do this, Manitou clears the per-page writable bits on all executable pages to detect when they are being modified. If the OS or some application tries to modify a page that contains executable code, Manitou removes the executable-bit from any page table entry that maps this physical page, effectively revoking execute permissions for that page, and then sets the writable bit on the page. If the program later tries to execute code on the modified page, Manitou will compute a hash of the new contents of the page, and revalidate it against its list of hashes. This enables Manitou to support instances of self-modifying code where the modified version of the program's code can be known a priori. For example, the Linux 2.6 kernel optimizes specific sections of the kernel code depending on the type of underlying processor. Manitou can handle this special case correctly because it is aware of both the original and optimized code page contents. Note that this technique is only applicable when hashes of modified pages can be pre-computed.

Since Manitou overrides the OS's page access bits, Manitou must determine on every page access violation whether

[1]Note that this problem in its most general form is equivalent to the halting problem.

the violation was of Manitou's page access policy, or whether the violation was of the OS's page access policy. To do this, Manitou maintains shadow bits that hold the policy that the OS intended for every physical page. Manitou's policy is always more restrictive than the OS's policy, so Manitou will only handle faults that would have normally passed under the OS's policy, but occurred because of restrictions it introduced. Manitou forwards all other faults to be handled by the OS, thus giving the OS the illusion that its page access policy is in place.

A clever attacker may attempt to modify executable code by asking Manitou to map a physical page as writable at one virtual address, and as executable at a different virtual address. To prevent such a situation from occurring, Manitou needs to ensure that no writable mappings of a physical page exist before creating an executable mapping of that page. Similarly, it needs to ensure that no executable mapping of a physical page exists whenever it allows a writable mapping.

A naïve solution would be to examine all page table entries every time a page becomes executable or an executable page becomes writable. However, this will make changing page executable bits an expensive operation as Manitou would have to traverse the page tables of every process in the system each time the executable bit is modified. Instead, our implementation maintains a *frame map*. The frame map consists of one entry per physical page. Each entry records the *type* of the page, as well as the number of mappings of that page with that type. The type of a physical page can be either "Writable", "Executable", or "None" and these types are mutually exclusive. The frame map contains enough information to know whether another writable or executable mapping of a page exists. Therefore, Manitou can avoid the cost of validating the page tables of all processes in the common case and only performs this examination when another mapping exists.

While Manitou relies on features of the underlying processor, it is OS-agnostic. As a result, it does not need to rely on assumptions about the OS implementation for correctness and only the hashes in Manitou's list need to be updated when the user upgrades or patches their OS. Because Manitou verifies the authenticity of kernel code just like any other program, malware that tries to hide itself by loading itself into the kernel will be detected by Manitou. Manitou is a small addition to the hypervisor, ensuring that the hypervisor remains lean and simple – our current proof of concept consists of less than 1500 lines of code added to the Xen hypervisor [1].

4. SOFTWARE INSTALLATION AND USAGE

This list of authorized code hashes needs to be managed and protected by Manitou to prevent corruption by an adversary. Because Manitou interposes on all access to hardware resources by the operating system kernel, Manitou is able to protect this list from modification by any malware inside the OS kernel or user programs. On the other hand, users will want the ability to install and run new programs. To allow this, Manitou exposes an interface to modify the list of hashes. When the user installs a new application, hashes of the code pages of that application need to be added to Manitou's list of authorized hashes. We describe two complementary approaches to do this.

The first approach consists of requiring new applications to submit a signed list of code page hashes to Manitou. Manitou maintains a list of valid signing certificates and the list of hashes will only be accepted if the signature is valid. While this would ensure that only authorized code is permitted to run, this approach unfortunately requires the establishment of a Public Key Infrastructure (PKI) to certify and distribute signing keys. Such a centralized method for global software distribution is not acceptable as it would severely hamper small software developers, as well as open-source software that is distributed by source code and compiled individually by users. Thus, we believe the applicability of this approach is limited primarily to environments where a single entity is responsible for determining what software can be installed on all machines, such as organizations with centralized IT management, or to environments allowing automatic software updates from trusted entities to be applied to machines for individual users.

The second approach allows individual users to determine what hashes should be added to Manitou. Hashes are added via a trusted path between the user and Manitou that does not involve the monitored OS. As a result, a malicious OS or application cannot add hashes of its own. Besides, when adding a set of hashes for a new application, the user is asked to provide a name for that set. This name will then help identify the application should the user later decide to neutralize it. The details of how the set of hashes is provided by the user to Manitou are beyond the scope of this work. Our current solution involves mounting the file system of the monitored OS in a distinct virtual machine and generating the hashes from the on-disk binaries of the application.

While this second approach appears to leave Manitou vulnerable to the same social engineering attacks that currently allow attackers to trick users into installing malware, Manitou provides an ability that will help users get rid of installed malware. The key is that Manitou will allow users to give execution privileges to software by adding hashes to its list, but guarantees that users will always be able to revoke those privileges by removing the hashes from the list, should they later find that the application is malicious. While the application's code will remain on the system, revoking its execution right has neutralized it. As a result, any software that does not come signed by an organization the user trusts can have its hashes installed temporarily, much the same way that a user that is not completely trusted may be granted a temporary and limited "guest account" on a system. Manitou is not a jail, as it does not rely on a temporary sandbox. Instead, possibly malicious applications are allowed to run on the machine itself, with Manitou providing the guarantee that such software will always be removable. We feel that this guarantee, combined with a simpler usage model and more compact mechanism are a desirable alternative to complex and heavy-weight jails.

Since Manitou must verify the code pages of any program before it executes, it also provides other useful abilities to the user. Manitou is able to inform the user whenever a program is executing. This can aid a user in identifying malware by informing her whenever a program executes without her knowledge – a typical behavior for many kinds of malware. It is possible that the malware came bundled with a seeemingly innocuous application. Manitou will help the user make the connection between this application and the

OS	Code (MB)	Hashes	Storage (MB)
Win XP a	1466	379,184	13.02
Win XP b	1506	388,496	13.34
FC4 Linux	949	239,408	8.22

Table 1: Hash list size estimation for two popular operating systems. The *Code* column indicates the amount of executable code in megabytes, found in those operating systems. The *Hash* column gives the number of hashes this translates to, and finally the *Storage* column indicates the amount of memory in megabytes required to store the entire hash list, assuming each hash is 256 bits + 32 bits of metadata.

suspicious behavior he observes on his computer. In addition, before removing a piece of software she suspects is malware, a user can monitor when it is being executed to see if revoking its execution rights will break dependencies with other software. In the case where malware came bundled with another application, neutralizing the malware will mean disabling the application as well. Finally, when a user is about to perform a sensitive operation, she can use Manitou to temporarily disable the execution of software that the user does not trust, but does not want to remove permanently from her system. If such software is currently executing, Manitou injects code into a program's instruction flow to make it execute a `sleep` system call, causing the operating system to suspend the process. Later, execution of that program can be resumed and no local program state will have been lost.

5. PRELIMINARY RESULTS

Initially, one of our concerns was that computing hashes of every page of executable code instead of computing a single hash for every executable file would lead to an extremely large list of hashes. The performance of applications may be adversely affected due to the time required to do a lookup into this large list of hashes, particularly if the lookup results in disk accesses because the hash list does not fit in memory. Conventional wisdom dictates that security mechanisms that impact the usability of a system will eventually be disabled by users themselves.

Thus, we began by trying to estimate the size of hash lists for some common commodity OS installations. Table 1 shows the amount of binary code we measured on two Windows XP systems currently being used by the authors and a complete Fedora Core 4 Linux distribution with all packages installed. We also show the number of hashes this results in and the amount of memory required to store the hashes in Manitou. From these results, we realized that the storage overheads for such a hash list would be quite modest. For comparison, we have observed that a common anti-virus utility in use at the University of Toronto typically consumes about 20 MB of memory. Given the large amounts of memory available on current machines, it is realistic to assume that the hash list could be cached completely in memory.

To further investigate the feasibility of Manitou, we have implemented a proof of concept system on top of the Xen hypervisor system running Linux as our commodity OS. Our system runs on an Intel Pentium 4 processor (version 630), which supports per-page executable bits. In our experiments, we found that since code page authentication occurs when a page is executed for the first time, this event always happens right after a page fault to disk because OS kernels lazily load pages from disk when they are accessed. Manitou only causes a small amount of performance degradation since a page authentication operation consists only of a hash calculation and a lookup into the hash list, whose time can be made logarithmic by sorting the hash list. Both the hash calculation and lookup occur in memory and are thus much faster than the disk access required to fault in a page. Once the page has been authenticated by Manitou, it does not have to be checked again, so the authentication cost is only incurred once per page fault. In addition, since code pages are frequently shared read-only across many processes, all pages in those processes only need to be checked once, further reducing the frequency of page authentications. As a result, we find that Manitou adds almost no overhead.

6. LIMITATIONS AND FUTURE WORK

Manitou deals solely with binary code. As such, it cannot control malicious interpreted code, such as Javascripts or shell scripts. While malware can be and has been written in interpreted code, operating system kernels do not include interpreters, so any code running in supervisor mode is binary code. This means that malware that does not contain binary code will not be able to modify the OS kernel itself and can be easily removed by conventional methods. In addition, interpreters could implement mechanisms similar to Manitou to further protect the user. For example, the Java Virtual Machine could hash application bytecode before translating it.

Another current limitation of our system pertains to revoking execution rights for an application. This will result in a set of hashes being removed from the list of code that is allowed to execute and in an illegal instruction error if that code tries to execute again, crashing the application that tried to execute that code. If the malware has modified code crucial to the execution of the system, e.g., by overwriting a core system library, the system will no longer be functional. While Manitou allows the user to identify the culprit, an unusable system is far from ideal. We are currently working on solutions to allow system recovery in the face of insidious malware.

In addition, our current work centers around authenticating dynamically generated code and providing a light-weight trusted path mechanism. Programs supporting virtual execution environments, such as Java just-in-time (JIT) compilers and VMware, dynamically generate and execute code. Because this code is created by a program during run-time, and may even be recompiled and optimized while being executed, the hash of the code page will not be present in Manitou's hash list. As a result, we must enlist the aid of the dynamic code generation program to sandbox the application and inform Manitou of what code it is generating.

Finally, we are also working on providing a trusted path between Manitou and the user to make sure that malware cannot spoof interaction with Manitou. While using a separate channel is a viable solution, we think that we can provide a more user-friendly solution by leveraging Manitou's ability to dynamically restrict what code is allowed to run on the machine.

7. RELATED WORK

There exists a number of systems that try to detect illegal modification of binaries. One of the earliest and simplest is Tripwire [5], which maintains cryptographic hashes of executables on the file system, and checks these periodically. More recently, there have been several projects using secure coprocessors to scan parts of kernel memory and ensure that they have not been tampered with [10, 13]. Livewire is another project that implements kernel memory scanning in a Virtual Machine Monitor (VMM) [3]. Manitou provides stronger guarantees than these systems by verifying the integrity of all binary code on the system instead of just the kernel or selected binaries, and is able to do so continuously instead of just checking periodically.

Attestation tries to prove to a remote party what code is executing on a machine, typically by computing hashes of executables loaded in memory and then signing these hashes with a secret signing key. One proposal is to use specialized hardware to compute hashes of the operating system and low-level software, and then have the operating system compute hashes of all applications as they are loaded [11]. The Terra Trusted VMM [2] computes and signs hashes of virtual machine images as they are loaded, and can then attest for them to a remote party. All attestation techniques suffer from a common weakness, which is that they can only make guarantees about the state of the machine when the measurement is taken. Manitou's combination of measurement using hashes with executable bits in the page tables allows Manitou to guarantee that the measurements it takes will be valid at all times.

A plethora of work exists on jailing applications to prevent them from damaging the system. For example, SEE [12] provides one-way isolation, where the application can read data from the system and is given the illusion that it can modify data. Later, the user can choose to commit the changes the application believes it made to the system. The user now faces a complex decision: commit early and risk permanently installing some malware, or commit late and have a hard time reconciling several views of the system. We feel that this "many worlds" paradigm is too complex for the average user. Rather, Manitou sacrifices data integrity but provides a simple, guaranteed way for users to uninstall software, preventing it from further damaging their system. Revocation of execution privileges can be performed even months after the software was installed, and regardless of where the software has hidden itself on the file system.

8. CONCLUSIONS

To combat malware, we must provide users with systems that will prevent the execution of unauthorized code, and let users know exactly what code is executing on their machines. In this paper, we argue that such systems are feasible with today's hardware and operating systems. We outline a system we are currently building called Manitou, which provides users with the ability to assign, as well as revoke execution privileges to programs based on the hashes of their code pages. In addition, users can also be notified whenever software on their system is executed, helping them identify potentially malicious programs and ascertain dependencies between applications. Manitou provides these abilities regardless of the integrity of the OS or the type of OS the user is using. We believe that a simple usage model such as this could be easily adopted by users, and help them combat malware by making it easier to detect and recover from a malware infection of their system.

9. ACKNOWLEDGMENTS

We would like to thank Tom Hart, Jesse Pool, Ian Sin and Richard Ta-Min for their comments, which helped improve this paper. The research presented in this paper was supported in part by an NSERC Discovery Grant and a MITACS Seed Grant.

10. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Oct. 2003.

[2] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Oct. 2003.

[3] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, Feb. 2003.

[4] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[5] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[6] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[7] Microsoft Antimalware Team. The Windows malicious software removal tool: Progress made, trends observed. Technical report, Microsoft, June 2006.

[8] A. Moshchuk, T. Bragin, S. D. Gribble, and H. Levy. A crawler-based study of spyware in the web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006)*, Feb. 2006.

[9] R. Naraine. Microsoft says recovery from malware becoming impossible, 2006. `http://www.eweek.com/article2/0,1895,1945808,00.asp`.

[10] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Aug. 2004.

[11] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, Aug. 2004.

[12] W. Sun, Z. Liang, R. Sekar, and V. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the 12th Annual Symposium on Network and Distributed System Security (NDSS 2005)*, Feb. 2002.

[13] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Sept. 2002.