Towards Protecting Sensitive Files in a Compromised System

Xin Zhao Kevin Borders Atul Prakash
University of Michigan, 1301 Beal Ave, Ann Arbor, MI, 48109, USA
{zhaoxin,kborders,aprakash}@eecs.umich.edu

Abstract

Protecting sensitive files from a compromised system helps administrator to thwart many attacks, discover intrusion trails, and fast restore the system to a safe state. However, most existing file protection mechanisms can be turned off after an attacker manages to exploit a vulnerability to gain privileged access. In this paper we propose SVFS, a Secure Virtual File System that uses virtual machine technology to store sensitive files in a virtual machine that is dedicated to providing secure data storage, and run applications in one or more guest virtual machines. Accesses to sensitive files must go through SVFS and are subject to access control policies. Because the access control policies are enforced independently in an isolated virtual machine, intruders cannot bypass file protection by compromising a guest VM. In addition, SVFS introduces a *Virtual Remote Procedure Call* mechanism as a substitute of standard RPC to deliver better performance in data exchanging across virtual machine boundaries. We implemented SVFS and tested it against attacks on a guest operating system using several available rootkits. SVFS was able to prevent most of the rootkits from being installed, and prevent all of them from persisting past reboot. We also compared the performance of SVFS to the native Ext3 file system and found that performance cost was reasonable considering the security benefits of SVFS. Our experimental results also show VRPC does improve the filesystem performance.

1 Introduction

Modern operating systems are very complex. Despite of best effort of system researchers, it is very difficult to prevent all computer security breaches. Successful intruders often attempt to disrupt computer systems or make profit by accessing sensitive files. For example, intruders may steal confidential information (e.g., passwords), taint system logs to cover up intrusion trails, or create back doors and Trojan horses to hide their presence and retain access to the machine [2] [6].

While it is very difficult to completely prevent computer intrusions, if sensitive system files can be protected from a compromised system, administrators will be able to largely restrict intrusion activities, determine the attacking source, and quickly restore the system to a safe state. For example, one can thwart many viruses from infecting system executables by preventing modification on these files. The trails of many intrusions can also be discovered from the system logs, if the log file is protected from being scrubbed.

However, protecting sensitive files in a compromised system is not easy. Most existing file protection mechanisms reside in the same space as the victim OS. If the operating system is compromised, any file protection mechanisms running in the victim operating system becomes vulnerable and can be bypassed or disabled. Therefore, for a viable file protection mechanism that is designed to continue to function in an untrusted environment, the most important challenge is how to prevent itself from being bypassed or disabled.

In this paper, we present **SVFS**, a Secure Virtual File System to leverage virtual machine technology to protect sensitive files from a compromised operating system. SVFS uses multiple virtual machines. One virtual machine, called *Data Virtual machine (DVM)*, is dedicated to providing the single file system that protects sensitive files, such as system binaries, shared libraries, and privileged service configuration files. Other "guest" virtual machines contain a user's normal operating system and applications. When accessing sensitive files from a guest VM, requests must go through the SVFS layer, which is transparent to the guest operating system. SVFS only accepts requests that satisfy the access control policies.

The main contribution of SVFS is threefold. First, by storing sensitive files in an isolated virtual machine, SVFS can independently regulate all access requests to these sensitive files, preventing intruders in a compromised guest VM from turning off or bypassing the file protection. Second, SVFS provides an interface to allow guest virtual machines to access the sensitive files stored in DVM transparently, as if these file are stored in those guest VM's local filesystem, making SVFS convenient to use. Finally, by introducing a VM based communication mechanism, called *virtual remote procedure call (VRPC)*, SVFS is able to support fast data exchange across virtual machine boundary, which greatly improves the file system performance.

The remainder of this paper is organized as follows. Section 2 describes our assumptions and the threat model. Section 3 presents the design of SVFS. Sections 4 illustrates a key technique used in SVFS: VRPC mechanism. Section 5 discusses the security properties SVFS must possess to effectively protect sensitive files from a compromised virtual machine. It also analyzes how SVFS meet these security requirements. Section 6 compares SVFS with several potential file protecting mechanisms. Section 7 presents experimental results from evaluating security offered by SVFS against several rootkits running in privileged mode. Section 8 evaluates SVFS performance on a workload similar to that of the Andrew benchmark. Finally, Section 9 presents conclusions and directions for future work.

2 Threat model and assumptions

SVFS focuses on protecting sensitive files from a compromised operating system. We define "compromised" to mean that the attacker subverted the victim host's software system and gained OS-level privileges. After successfully compromising a machine, many intruders wish to modify critical system files to steal confidential information (e.g., passwords), taint system logs to hide intrusion trails, or create back doors and Trojan horses to hide their presence and retain access to the machine. System administrators, on the other hand, wish to prevent intruders from undetectably tampering with, deleting sensitive files, or adding malicious codes that can run automatically during the system reboot.

The following types of files and directories are regarded as sensitive and protected by SVFS:

- System configuration files such as `/etc/rc.d/rc.sysinit`;
- System executables
- Shared libraries
- Critical system directories like `/etc/rcX.d`, where malicious scripts can be placed that will executed during system boot.
- System log files that an intruder may want to modify to cover up the trail of an intrusion.
- Other files and directories specified as sensitive by an administrator.

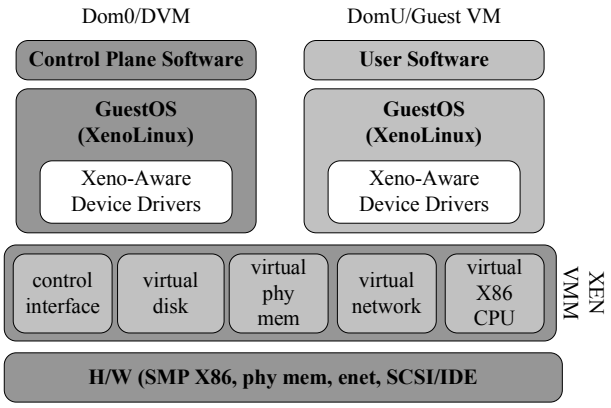

Figure 1: Virtual Machine Architecture

SVFS is not designed specifically to stop attacks in a compromised system. It does not detect and prevent attacks that do not involve any file modification. However, if properly configured, SVFS can prevent malicious codes from persisting across system reboot.

SVFS assumes that attackers may have software control over the protected host, but does not have physical access to the machine. SVFS does not address insider attacks, in which intruders often have physical access to the protected machine. A user on the console is considered trusted. Also, we assume that the administrative channel of SVFS is secure.

3 Design of SVFS

Because we assume that intruders can compromise the host's software system and gain OS-level privileges, any data protection mechanisms residing in the same space of the victim operating system are subject to attacks and can be disabled, including those in-kernel security policy enforcers. In order for a file protection to continue to function in a compromised environment, it must be isolated from the attacker. Virtual machines can help in achieving this goal because they are good at providing separation between virtual hosts on the same physical machine.

In this section, we first give an brief overview of virtual machine technology. Next, we present the architecture of SVFS.

3.1 Overview of VM technology

Virtual machines have existed for thirty years, and used to be deployed on mainframes. Recently, virtual machines have been growing in popularity, and are now widely used on personal computers and servers. Examples of modern virtual machine systems include VMWare [29], UML [13], FAUmachine [17], Denali [31], Disco [9] and Xen [5]. Some of these systems have led to successful businesses and are currently used by many large corporations [4].

The most important component in a virtual machine system is its virtual machine monitor (VMM). Figure 1 illustrates the Xen virtual machine monitor as an example. A VMM is a thin software layer that runs directly on a physical machine's hardware. On top of the virtual machine monitor, there can be one or more virtual machines. The VMM provides each virtual machine with a set of virtual interfaces that resemble direct interfaces to the underlying hardware. Different virtual machines can run different operating systems, which are referred as the *guest OS*. Applications in guest OS, often referred as *guest applications*, can run without modification as if they were on a

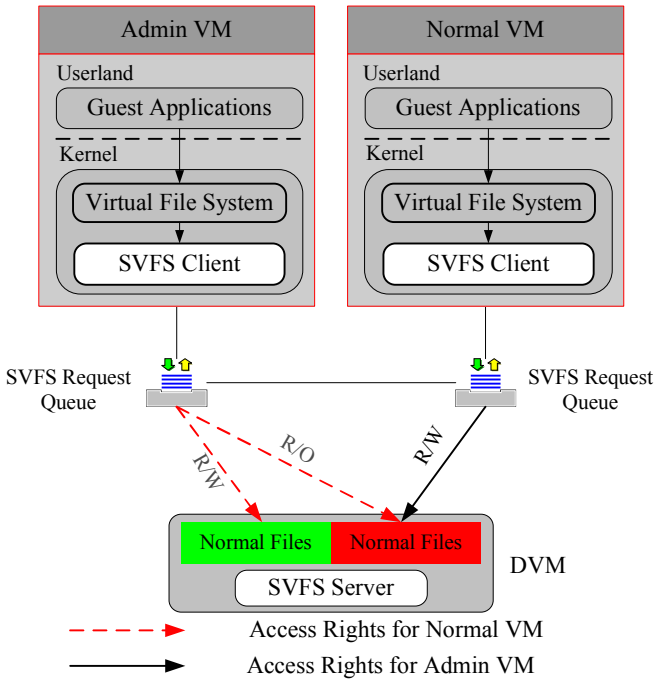

Figure 2: System Model

dedicated physical machine. The VMM also allows multiple virtual machines to be running at the same time and transparently multiplexes resources between them [16]. The VMM isolates the virtual machines from one another, preventing them from accessing each other's resources.

SVFS is developed on the Xen architecture, whose system model is slightly different from traditional VM systems like VMWare in I/O device virtualization. In particular, Xen deploys xen-aware I/O device drivers in each guest VM. These xen-aware I/O device drivers are responsible for forwarding virtual I/O commands of guest virtual machines to a special privileged virtual machine, called *Domain0* in Xen terminology. Domain0 is then able to serve the virtual I/O commands for I/O device virtualization. This technique is called paravirtualization and helps improve the performance of virtualized I/O devices. Domain0 also works as an administrative VM to allow administrators to communicate with Xen and manage other guest VMs. Further details can be found in [5]. Currently, SVFS uses Domain0 as DVM to store sensitive files for other guest virtual machines.

3.2 SVFS Architecture

To protect sensitive files, one can enforce security policies over sensitive file requests at the VMM layer, since VMM is responsible for disk virtualization and is able to see all disk I/O requests coming from guest virtual machines, including file requests. Unfortunately, the disk I/O requests that VMM can see only contain block-level parameters such as cylinder and sector numbers [29]. To enforce file-level security policies, VMM would have to translate the block-level operations to file-level operations. This would require VMM to understand the structure of every filesystem used by the guest virtual machines, and likely to be difficult and impractical to implement [11].

Instead, as Figure 2 shows, SVFS proposes to deploy multiple virtual machines in a physical machine. For simplicity of description, we assume that all virtual machines run Linux as their operating system. One virtual machine, called *Data Virtual Machine (DVM)* is dedicated to running

SVFS, a new file system, to store and protect sensitive files for other VMs. In other guest virtual machines, users run their standard applications. When one of these applications needs to access a sensitive file, it must go through the SVFS file system, which happens transparently.

A SVFS file system is composed of two parts: *SVFS server* that runs in DVM, and *SVFS clients* that run in each guest VM. An SVFS client runs at the kernel level of each guest virtual machine. It registers the SVFS file system and is hooked to the virtual file system (VFS) layer [8]. An guest application can access sensitive files by issuing system calls, such as `open`, `read` and `write`. The file accessing calls are processed by the Linux VFS layer. The Linux VFS layer determines that the requested file object belongs to SVFS and call the corresponding SVFS function registered by the SVFS client. The SVFS client then packs the file request parameters and put the packed request into the request queue by issuing a virtual remote procedure call (VRPC) to the SVFS server. (The VRPC mechanism is described in detail in section 4.) After the virtual remote procedure call returns, SVFS client unpacks the result and sends it back to Linux VFS layer, giving guest applications an illusion that the request is processed locally.

SVFS server runs in DVM and uses the Ext3 filesystem [30] as its local filesystem to store files for guest virtual machines. It is responsible for checking file access requests coming from other guest virtual machines against security policies. Only valid requests are accepted. Unlike many file systems that employ user based access control mechanism, SVFS uses a VM based access control model. For an SVFS file system object such as file, directory and symlink, an administrator can specify different access permissions for different guest virtual machines. The supported access permissions will be discussed in the coming section 3.3. Upon receiving a file access request, SVFS server identifies the source VM that making this request by the communication channel over which the request is sent. With the source VM identity, SVFS server verifies compliance of sensitive file access requests with security policies accordingly. Only valid requests will be served.

The advantage of developing a new file system to store sensitive files and putting the policy enforcer in DVM instead of VMM is that the security policy enforcer can see file request arguments at the file system level instead of disk block level, making the file system policy enforcement much easier. Because sensitive files are stored in DVM, access requests for those files must go through SVFS server to get served, making the security policy enforcement on sensitive files non-bypassable. Moreover, as SVFS server enforces security policies independently in DVM, which is isolated from other guest virtual machines, the file protection cannot be disabled by another virtual machine, regardless whether that guest VM is compromised or not. This guarantees that SVFS file protection can continue to function even if a guest VM is compromised.

Our prototype of SVFS is able to use NFSv2 protocol [27] to achieve transparent file access across virtual machine boundary. The same architecture also works for other distributed filesystem protocols such as CIFS [20] and Samba [10]. NFSv2 was chosen over other protocols because it is simple and is better suited for developing and testing a new mechanism.

Figure 2 also illustrates an example of how SVFS can be used to protect sensitive system files such as `/etc/passwd` and `/usr/sbin/sshd`. In this example, there is a ***normal virtual machine*** that runs standard applications such as a word processor, an e-mail client, or a web browser. Sensitive system files such as `/etc/passwd` and `/usr/sbin/sshd` are seldom changed by those applications; they are thus set to *read-only* for the normal VM. For the purpose of system administration, an ***administrative virtual machine*** is also deployed. The administrative virtual machine is allowed to update sensitive files, but network access to it should be restricted.

Upon receiving a request for a sensitive file from one of the guest VMs, SVFS server can identify the source of the request, and enforce the appropriate security policy. Using this approach, the normal virtual machine is able to run less secure applications, but is not allowed to modify sensitive files. However, if the user needs to update a sensitive file, that can be done by simply switching

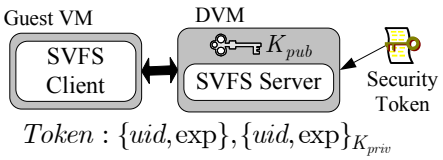

Figure 3: Security Token based File Access

to the administrative VM, which is less vulnerable to attack due to its limited network access and constrained set of applications.

While the above scenario is a simple example with only two guest virtual machines, SVFS can be configured in variety of ways depending on the desirable security guarantees. For example, a corporate administrator can restrict access to confidential files so that they can only be read by high-security VMs. These VMs may only be connected to the company's private network, preventing attackers from accessing sensitive files and preventing malicious insiders from sending them out over the internet.

3.3 Access Control Mechanisms

In SVFS, each guest VM is assigned a virtual machine ID (VMID), which is used to specify the access control policies for sensitive files at the per-VM-per-file granularity. There are two mechanisms that an administrator can use to govern access to these files.

The first way for an administrator to set a file's access control policy is to use standard file permissions (such as read, write, and execute), with one addition —"`append-only`" attribute. A file with the append-only attribute set can only be opened for writing at the end of the file. Overwriting or truncating an append-only file is not allowed. This attribute helps to protect the integrity of important data such as system logs while allowing appropriate modification.

The second way for an administrator to restrict access to a file is to specify extended security conditions to be checked when this file is to be accessed. Currently, SVFS supports checking of *security token* as an extended security condition. As figure 3 shows, a security token is a file that can be stored in a stand-alone physical device such as USB key or smartcard for extra security. The security token must be directly presented to DVM via a channel outside guest virtual machines. The content of security token is transparent to guest virtual machines. VMM is responsible for preventing other guest VMs from accessing the security token presented to the DVM. The content of the security token file can be represented as $\{uid, exp\}, \{uid, exp\}_{K_{priv}}$. Here, uid is the token owner's user ID, exp is the token expiration time, and K_{priv} is SVFS administrator's private key. The security token contains the plain text of the token owner's identity and token expiration, which are signed with SVFS administrator's private key. This private key is stored in a secure location outside of the machine and the corresponding public key K_{pub} is held by SVFS. With the public key K_{pub}, SVFS can decrypt the $\{uid, exp\}_{K_{priv}}$ part in the token. By comparing the decrypted $\{uid, exp\}$ with the associated plain text $\{uid, exp\}$, SVFS can check the validity of the token.

SVFS uses the Ext3 filesystem to store sensitive files in DVM. The VM based access permissions are saved as extended attributes and stored in the extended attribute block as part of Ext3 filesystem objects. An extended attribute is a (name, value) pair associated with inodes (files, directories, symlinks, etc). An extended attribute name is a simple NULL-terminated string. The name includes a namespace prefix - there may be several, disjoint namespaces associated with an individual inode. For example, the "system.posix_acl_access" is in the "system" namespace and used to store the access control list. The value of an extended attribute is a chunk of arbitrary textual

or binary data of specified length. SVFS stores the access permissions as extended attributes in the "user" namespace. Specifically, the traditional UNIX style permissions such as read, write, execute and append-only are stored as ("user.svfs_std.$VMID", "rwxa") pairs, which means that virtual machine with $\$VMID$ has the access permission "rwxa" to the file associated with this attribute. Here, the combination of "rwxa" stands for read,write, execute and append-only permissions of the associated file. The security token setting is set as ("user.svfs_tok.$VMID", "$UID:rwxa") pairs, which has similar semantics as the attributes representing UNIX style permissions. The only difference is $\$UID$, which is used to specify the owners of acceptable tokens. Currently, only one user ID can be specified in the $\$UID$ field of a security token attribute. We plan to extend it to support more flexible combination of users in the future. For example, an administrator can specify a confidential file to be readable/writable only if two authorized users present their security tokens, which could be useful in protecting top secrets.

In addition to the VM based access control mechanism, to avoid disrupting users' working pattern, SVFS also supports traditional UNIX style access control, which can prevent a user's file from being maliciously accessed by other users before the guest OS is compromised. However, because SVFS does not check the internal states in guest virtual machines, the user ID has to be reported by the guest OS. If the guest OS is compromised, a successful intruder can impersonate any user in the victim OS. Under such condition, the sensitive files will be protected by the VM based access control mechanism.

4 Virtual Remote Procedure Call

As sensitive files reside in DVM, they can be accessed from other guest virtual machines only through inter-VM communication. In order to improve the performance of SVFS file system, it is desired to have a mechanism that can efficiently exchange data between SVFS client and server.

Remote procedure call (RPC) is traditionally a convenient way to exchange data over network. We initially used RPC for communication between the SVFS client in the guest VM and the SVFS server in DVM. However, we noticed that standard RPC mechanism hurts SVFS performance, because it involves expensive operations, such as XDR encoding/decoding, packet packing/unpacking and connections establishing. Those operations are unnecessary for SVFS system as the client and server side resides on the same physical host.

To improve the performance of RPC communication on the same machine, Brian Bershad et al proposed to use thread tunnelling technique [7]. However, this technique is designed to improve performance of communication between two domains in `ONE` machine. Although a Xen server runs multiple virtual machines on the same physical host, these virtual machines are completely isolated as if they are physically separated machines, which prevents thread tunnelling. Moreover, this method is only effective when the RPC argument is small. As a filesystem, SVFS often need to transmit bulk of data.

We therefore designed *virtual remote procedure call (VRPC)*, a modified RPC mechanism based on Xen, to support fast data exchange between SVFS client and server running on the same physical machine. VRPC shares the same protocol and working flow with standard RPC, but differs in that VRPC does not rely on network to transmit data and is more efficient than standard RPC.

VRPC improves the data exchange performance by leveraging Xen's memory re-mapping support. In a Xen server, a virtual machine can allocate a memory region and report the starting address and size of this region to another virtual machine, which can then map that memory region into its own memory space and access data stored in that region directly. VRPC leverages this memory re-mapping support to setup a shared memory space between DVM and guest VM. These

two virtual machines can then exchange data using this memory space, which avoids expensive network transmission and reduces the number of memory copy needed for data exchange to zero, because the data stored in the shared memory can be seen directly by two virtual machines. In contrast, network based RPC needs at least two memory copies: the sender copying data to network stack from data buffer and the receiver copying data from network stack to data buffer. In addition, as VRPC exchanges data between virtual machines on the same physical host, both sides have the same endian order. Therefore, XDR encoding and decoding is unnecessary for VRPC and can be removed, which further reduces the VRPC overheads.

Different SVFS requests involve different amount of data exchange. For example, function *lookup()* searches certain directory for a specified file. The input parameters are file name and parent directory inode. The output values are file attributes such as file size, modification time and access rights. This function only involves small amount of data exchange. On the other hand, function *read()* or *write()* can involve data of arbitrary length.

VRPC implements an hybrid memory re-mapping mechanism to handle different types of data requests. For functions that need to exchange data of variable length or more than 4K bytes of data transfer, VRPC allows SVFS client to dynamically allocate memory and map it into DVM for data sharing. In SVFS, dynamic memory sharing is used in the following functions: `read()`, `write()`, `readlink()`, and `readdir()`. All these functions involve variable-length data exchange.

Other file system requests and results can be packed into relatively small buffers (less than 4K bytes). At initialization stage, SVFS client allocates a memory region that is sufficient for such requests. The memory region is mapped into Dom0 memory space and shared by SVFS client and server. This static memory sharing needs only one Xen call to remap memory at the initialization stage, and is thus very cheap. On the other hand, the dynamic memory sharing mechanism is more expensive as it needs a Xen call for each memory mapping, but it allows arbitrary length of data to be transferred in a VRPC call.

A *communication ring*, consisting of statically allocated shared memory, is used as the SVFS request queue to exchange request parameters and responses between a SVFS client and the SVFS server. The communication ring is divided into multiple 4K byte slots, with each request and response using one slot. SVFS client puts file requests and fetches responses. SVFS server fetches the requests and put the responses back to the ring.

When a new request or response is put into the shared ring, its producer has to notify the other party that new data is available. VRPC use the event channel mechanism provided by Xen to achieve notification. Event channels are an asynchronous inter-VM notification mechanism. At a guest VM's bootup stage, DVM instantiates an event channel between this guest VM and itself. Both virtual machines can request that a virtual IRQ be attached to notifications on a given channel. The result of this is that one virtual machine can send a notification on an event channel, resulting in an interrupt in the other virtual machine. By installing an IRQ handler, SVFS server and client can detect new request or response that is put into the communication ring by the other party. They can then checks the shared communication ring to get new request or response. The event channel can only transmit a one bit message, so it is not appropriate to transfer bulk data, but it is reasonable for event notification.

An example helps illustrate the working of a VRPC. Suppose a guest virtual machine wants to read 8k bytes of data from a sensitive file f stored in DVM. The SVFS client first allocates a buffer of size 8k. The SVFS client then puts a request in the shared ring that includes the read function code, the file handle, offset, starting physical address x of the buffer for receiving the results, and the buffer size. It then notifies the SVFS server of this new request via the event notification mechanism. Upon receiving this request, the SVFS server maps the memory buffer of length 8K bytes that starts at physical address x into its own memory space and executes the

read operation, resulting in the file data being placed in the mapped memory buffer. After the operation is completed, DVM notifies the guest virtual machine. The SVFS client receives an interrupt, and returns the data that is in the mapped memory region. No extra memory copy or network transmission is required.

A malicious guest VM may try to request excessive memory sharing request, hoping to exhaust all memory of DVM as an attempt of denial-of-service attack. However, SVFS server never allocate memory in DVM for sharing with SVFS clients. Instead, it only maps the memory allocated by a SVFS client in a guest VM. Therefore, this kind of DoS attack is impossible.

Because the starting address and the size of the memory region is reported by SVFS client from the guest VM, intruders in a compromised VM may try to report false address or memory size for sharing, hoping to disrupting SVFS server or inject malicious data into DVM. However, this attack is unlikely to succeed. While SVFS server has no way to tell the validity of the reported starting memory address and size, false starting address only causes memory mapping failure (when the memory space is not belonged to the guest VM) or SVFS server to access data at the wrong place in the guest VM, which will only disrupt the malicious guest VM. Moreover, this method is unlikely to be used to attack guest OS either. Because only SVFS client that runs in the guest OS kernel has the privilege to request SVFS server to dynamically remap a memory region, if the SVFS client is manipulated to sent false memory information, the intruder must already get the system privilege on the guest VM. Reporting false information will not further escalate the intruder's privilege in the compromised VM.

5 SVFS Security Discussion

In this section, we first discuss the essential security properties that SVFS must possess to effectively protect sensitive files from a compromised guest OS. Next, we will analyze how SVFS meets those requirements.

5.1 Essential Security Properties of SVFS

According to the SVFS architecture, in order to continue to function in a compromised environment, SVFS must possess the following security properties:

1. *Trusted computing base (TCB) is difficult for an attacker to compromise.* The trusted computing base of SVFS is the combination of VMM and DVM. The SVFS security argument rests on the assumption that TCB is hard to compromise and will work properly. A compromise of SVFS's TCB is a catastrophic failure in a SVFS system.

2. *DVM is securely isolated from other guest virtual machines.* SVFS can continue to function and independently enforce security policies in a compromised environment only if a compromise of other guest virtual machines does not result in immediate access to resources of DVM, where sensitive files and SVFS server reside. Therefore, DVM must be securely isolated from other VMs regardless of whether those virtual machines are compromised.

3. *Communication between DVM and guest VMs is secure.* As sensitive files reside in DVM, they can be accessed from other guest virtual machines only through inter-VM communication. For a file access request, SVFS server enforces security policies according to the source VM that making this request. The source VM is identified by the communication channel over which the request is sent. Therefore, the communication channel must be secure; otherwise,

a malicious VM can disrupt SVFS by impersonating DVM or another guest VM, or injecting false data into other guest virtual machines' communication channels.

4. *Administrative Environment is secure.* An administrative VM is often deployed to configure SVFS system and other VM components, as shown in the example in section 3.2. It is usually authorized to modify sensitive files and other system settings. Intruders can therefore bypass any SVFS protection if they can compromise the administrative virtual machine.

Note that SVFS does not rely on the security of guest virtual machines, so the integrity of guest VMs is not included as an essential security requirement. A malicious VM can send any invalid file requests, but all requests will be checked independently by SVFS server in DVM. A false file request from a malicious VM will only result in wrong data or access deny, and will not disrupt the normal running of SVFS.

5.2 SVFS Assurance

5.2.1 TCB Security

The SVFS security argument rests on the assumption that TCB is difficult for an attacker to compromise. We believe this assumption is reasonable for two reasons. First, compared to a full operating system that typically has several million lines of code and runs variety of services, VMM and DVM are much simpler and more likely to be implemented correctly. Specifically, VMM is primarily responsible for virtualizing the hardware of a single physical machine and partitioning it into logically separate virtual machines. A VMM is usually built on an order of tenth of thousands lines of code. For example, Disco [9] and Denali [31] have around 30K lines of codes [14], and Xen has around 60K lines of codes. Although DVM also runs an operating system, it is dedicated to running SVFS service only. All irrelevant components are removed. Second, the interfaces to VMM and DVM are much simpler, more constrained and well specified than a standard operating system, which reduces the risk of being attacked. For example, Xen virtual machine monitor can be accessed only through 28 predefined hypervisor calls. In addition, Xen needs one communication channel for each virtualized I/O device of a guest VM to exchange I/O commands and results [5]. Moreover, DVM needs to expose one communication channel to each guest VM for sensitive file accessing. In contrast, modern operating systems expose much more interfaces. For example, a standard Linux operating system can be accessed via a lot of system calls (i.e. kernel 2.6.11 exposes 289 system calls), special devices such as /dev/kmem, kernel modules, plenty of privileged programs (sendmail and sshd, for example), and variety of network services provided by the OS or other software vendors.

5.2.2 Isolation between DVM and other guest VMs

As described in Section 3.1, the underlying VMM is responsible for isolating virtual machines from one another. This guarantees that SVFS server running in DVM cannot be disabled by another virtual machine, regardless whether that guest VM is compromised or not. Similarly, isolation also prevents a malicious guest VM from accessing virtual disks of DVM directly, which guarantees that accesses to sensitive files must go through SVFS server so that security checking cannot be bypassed.

5.2.3 Secure communication between two virtual machines in a physical host

To make communication channel between two virtual machines secure, two conditions must be satisfied. First, the sources of communication packets should not be spoofed, otherwise, a malicious VM can impersonate another authorized VM to access sensitive files or impersonate DVM to provide false data to guest virtual machines. Second, the communication data should not be modified during transmission. The underlying VMM help SVFS satisfy these two conditions. SVFS uses the VRPC mechanism as the communication channel to exchange file requests and responses between guest VMs and DVM. VRPC is essentially a memory region allocated in the guest VM and mapped into DVM's memory space for sharing. VMM ensures that this memory is accessible for the owner VM and DVM only. Thus, both the owner VM and DVM can identify each other by this shared memory region. A third party cannot impersonate them since it cannot access the shared memory, which prevents source spoofing. For the same reason, the data exchange via the shared memory cannot be changed by a third party since it cannot access this memory region, which guarantees that communication data is not changed during transmission.

5.2.4 Secure administrative environment

There is a risk that intruders can compromise an administrative VM to change SVFS permissions or modify sensitive files. However, this could be very difficult if the administrative VM is isolated from the outside network and other guest VMs, as is recommended when deploying SVFS. An administrative virtual machine should be set up for system maintenance only and not have any extra services running that may be vulnerable to attack. An administrative VM should also run a firewall that limits unnecessary network access to it; ideally only local console access should be allowed. If all the above precautions are taken, then it would be exceedingly difficult for a remote attacker to compromise an administrative virtual machine.

6 Related Work

Diverse paradigms have previously been developed to protect sensitive files.

To detect modification of sensitive data, one can deploy an integrity checker such as Tripwire [18] that periodically verifies cryptographic hashes of specified files by comparing them to trusted values. However, Tripwire is a passive data protection mechanism that does not actively prevent malicious accesses to sensitive data. Using such a system, serious damage may have already occurred before an alarm is raised. In addition, an intruder could potentially install a kernel module that feeds false data to Tripwire so that it is not detected [2] [6] [3] [33].

One can also mount the directories that contain sensitive files as read-only filesystem. However, after getting system privileges, a hacker can easily re-mount the filesystem and make it writable with commands such as "mount -o remount,rw /dev/sda*".

Another approach for protecting sensitive files is to use a mandatory access control system, e.g., SELinux [21], LIDS [32], and Systrace [25]. These systems associate immutable attributes with subjects and objects and perform authorization checks based upon those attributes. Sandboxing mechanism proposed in *Janus* [15] and *Chakravyuha* [12] tries to build isolation upon an operating system to prevent an untrusted entity from seeing the whole filesystem. The mandatory access control or sandboxing policy is often enforced at the OS kernel level to make it hard to be disabled. Those approaches, however, all suffer from the same problem: intruders can turn off the data protection if they manage to compromise the operating system kernel by exploiting kernel vulnerabilities [1], which are hard to avoid completely in a complex modern operating system. More-

over, intruders can read and write sensitive files by accessing raw disks (using system call *sys_iopl*, for example), rendering those kernel based monitors ineffective. This attack can be prevented by selectively allowing specified specified program to access raw devices, but doing this requires high expertise in system security and a minor mis-configuration can disrupt normal applications that use raw disk access for better performance, such as databases.

A network file system, such as NFS [23], could be configured to protect sensitive files by giving read-only access to some machines and read-write to others. SVFS is similar to this solution, but has the advantage of running on a local machine and allows SVFS to work without network connectivity and external server. This makes system management more convenient. SVFS also supports a more extensive set of file permissions, including token based access control. In addition, the performance of SVFS is much higher than that of NFS file system even if both NFS client and server is deployed on the same physical host, which makes SVFS more practical for protecting frequently accessed files.

It is also possible to protect sensitive files with security co-processor or other dedicated hardware. These hardware rely on the embeded firmware to work and are also hard to be disabled. However, to provide effective file protection, these hardware would have to understand the structure of every filesystem the operating system may use, which is likely to be difficult and impractical for hardware firmware.

The *VMI-IDS* system, originally proposed by Garfinkel and Rosenblum, periodically scans a virtual machine's file system from the virtual machine monitor to search for rootkits and viruses [14]. This approach is similar to that of Tripwire, but can detect unauthorized file modification even if a guest OS is compromised. In contrast, SVFS is a file protection service that prevents file from being tampered with in the first place. SVFS also differs in that it does not need to be run at the VMM layer. We chose to place SVFS outside of the VMM so that it did not have to map disk-level I/O commands to file system operations in order to check access rights, which greatly reduces the complexity of implementation and makes the system more practical.

Another potential solution for protecting files is to use a log-based versioning file system [28]. If a sensitive file is compromised, then the file system is able to recover a previous version. This solution does not prevent malicious file accesses at the first place, but offers a recovery mechanism. It can be used as a supplement to SVFS to backup files updated by authorized VMs, which can further protect those files even if an authorized VM is compromised. SVFS also differs from the log-based versioning file system in that it provide higher performance with the VRPC mechanism while the log-based versioning file system would need to use network to transmit data.

Pennington et al. proposed a storage-based intrusion detection system to monitor access to sensitive files [24]. It provides better isolation than a host-based IDS because the detection system is on a remote machine. Like other intrusion detection systems, however, it does not actively prevent access to sensitive files.

7 Security Evaluation

In this section, we first present the security evaluation results of the SVFS prototype. The evaluation consisted of testing known rootkits and backdoors to see if they are blocked by SVFS. Then, we take a look at limitations of SVFS.

7.1 Testing Against Rootkits and Backdoors

For this part of the evaluation, we attempted to install nine known rootkits and backdoors on a guest virtual machine and recorded the results. The rootkits were obtained from public sources

Type	Name	Prevented from installing by SVFS	Blocked from persisting across reboot by SVFS
User-level rootkits	t0rn	X	X
	Dica	X	X
	Lrk5	X	X
	Flea	X	X
	SAdoor		X
login/PAM backdoors	ulogin	X	X
	pam_backdoor	X	X
Kernel rootkits	Adore-0.31		X
	Knark-0.59		X

Table 1: Common Attacks Evaluation Results

with minor modification for running in our system[1]. These rootkits were classified as user-level if they did not modify the kernel, otherwise they were classified as kernel-level. For this experiment, rootkits were run already having administrative privileges on the guest OS.

We derived SVFS policies from default Tripwire policies that are included in the standard Tripwire package. In general, we set system directories and files within them to be read-only for normal virtual machines. These included: `/etc`, `/sbin`, `/lib`, `/usr/lib`, `/usr/bin`, and `/usr/sbin`. `/dev` directory is managed with udev [19], which allows device files to be generated in memory instead of disk. SVFS can then prevent new files from being created on disk in the `/dev` directory. In addition, some software packages employs variable files in the `/etc` directory. For example, a DHCP client modifies file `/etc/dhclient-eth0.conf` at runtime. We moved these files to a different directory "/var/etc" and left symbolic links in "/etc" so that we can lock down the `/etc` directory and still allow the variable files to be modified at runtime. According to our observation, these variable files are usually used to show system status or record internal states of the owner applications. They are unlikely to be used to attack other system components.

Table 1 summarizes results from our attempts to install nine rootkits and backdoors on a guest OS. SVFS was able to immediately stop six of the malicious programs from installing, and was able to prevent all of them from persisting past reboot. The rootkits that were immediately blocked relied on modification of sensitive system files that were protected by SVFS. Examples of standard system files that rootkits tried to modify include `netstat`, `tcpd`, `ls`, `ps`, `pstree`, `top`, `read`, `write` and `ifconfig`.

The three attacks that were not immediately prevented, SAdoor, Adore, and Knark, did not attempt to modify any sensitive files. SAdoor is a standard backdoor program that runs a server process in the background that accepts connections from remote users and gives control over the machine. SVFS is not able to prevent this type of attack altogether, but it does stop the remote user from tampering with protected files, and keeps SAdoor from starting again automatically after the guest OS reboots.

Adore and Knark inject a malicious module into the kernel image resident in memory. After the kernel module is installed, it redirects system execution system calls to trojan versions of standard binaries. SVFS is unable to prevent this type of attack because it targets the operating system image resident in memory. Standard versions of Adore and Knark rely on adding files to protected system directories in order to run when a system restarts, so SVFS is able to prevent them persisting past reboot.

[1] *http://packetstormsecurity.nl/UNIX/penetration/rootkits/* and *http://www.antiserver.it/backdoor-rootkit/*

In addition, even the rootkits that were able to install successfully in the guest virtual machine's memory were not able to tamper with any of the files protected by SVFS. Also, if a particular guest virtual machine was denied read access to certain files, then rootkits installed on that machine would not have access to them either.

7.2 Limitations

As many other security mechanisms, SVFS has its limitations. First, SVFS does not prevent malicious programs from running in a guest virtual machine's memory. SVFS is able to prevent these programs from running automatically when the guest OS reboots, but it does not stop an attacker from exploiting a vulnerability in the guest OS again after the system has started up.

Another potential attack on a guest OS involves leaving a root-owned malicious executable with setuid bit enabled in a writable directory. If a user could be tricked into running the malicious executable, then it could compromise the guest system. For example, an attacker could put "$PATH = \mathbf{/tmp/MALICIOUS} : \$PATH$" in a valid user's `.bash_profile`. The attacker can then place a trojan "ls" program in the"/tmp/MALICIOUS" directory that starts the rootkit after the user logs in.

One way to mitigate the threat mentioned above would be to add run-time checks in the guest's kernel to ensure that root-owned programs with the setuid bit cannot be run unless they are protected by SVFS.

8 Performance Evaluation

In this section, we evaluate the performance of SVFS by comparing it with a native Ext3 filesystem and a locally run network-based filesystem. The results show that SVFS offers enhanced security features with reasonable performance overhead. The results also show that the VRPC mechanism can significantly improve overall system performance when compared to a network RPC mechanism.

8.1 Experimental Setup

The SVFS prototype runs on a Xen 2.0.6 virtual machine system. DVM is deployed in Xen's privileged domain0, and runs Linux Kernel 2.4.29-xen0. DVM uses the Ext3 filesystem to store sensitive files for other virtual machines. The Ext3 filesystem is mounted with the "data=journal" option and uses a 50 MB ramdisk [22] as the journal device. A guest virtual machine was also used for this experiment, which ran linux kernel 2.4.29-xenU. All experiments were run on a PC with a Pentium IV 3.0MHZ processor and 512 MB of RAM. All virtual machines, were configured to use 128 MB of RAM.

For this experiment, we used a linux kernel build, which emulates the Andrew File System benchmark. This benchmark is not representative of all workloads, but is often used to compare the performance of different filesystems. The kernel build benchmark consists of three phases:

- The *unpack phase* decompresses a tar archive of linux 2.4.27 kernel (The archive is approximately 173 MB). This phase creates many small files and directories.
- The *configure phase* determines the source code dependencies and involves a lot of small file reads.
- The *build phase* compiles and links the kernel source code, then removes temporary files. It is the most CPU intensive, and generates many intermediate object files.

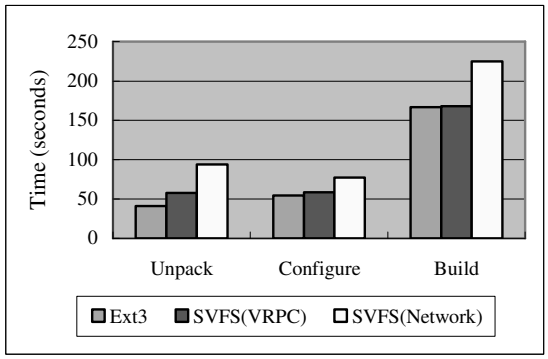

Figure 4: Linux Kernel Build Benchmark

Using the linux kernel build benchmark, we compared three filesystems. Two of them are versions of SVFS, the first version uses VRPCs to communicate between virtual machines, and the second uses network-based RPCs with read and write transfer sizes setting to 8K. These SVFS versions were compared to the native Ext3 filesystem running within a guest operating system. For both SVFS versions, all temporary files were placed on the DVM.

To avoid the effects of a warm buffer cache, all the experiments were conducted right after booting up the guest operating system. For these experiments, none of the requests violated SVFS file permissions. The experimental results are normalized.

8.2 Performance Comparison

Figure 4 shows the difference in performance between the native Ext3 filesystem, SVFS using VRPCs, and SVFS using network-based RPCs. Across all three phases, the Ext3 filesystem was the fastest, and SVFS with VRPCs was 8.3% slower, while SVFS with network RPCs was 51% slower than Ext3. Given these results, it is clear that VRPC is much faster than network based RPCs. Compared to the local filesystem, SVFS offers a higher degree of security without incurring a significant performance penalty.

The *unpack phase* of the benchmark was where VRPC-based SVFS and the native Ext3 filesystem showed the most difference. We believe the reason for this is that VRPC relies on NFS consistency semantics. The NFS consistency semantics require file data to be flushed to disk when closing a file, while Ext3 allows lazy disk updates. The *unpack phase* of the kernel compilation benchmark involves creation, writing, and closing of many small files. This causes file data to be flushed to disk frequently and reduces the effect of disk cache. In contrast, Ext3 file system can take full advantage of disk cache and flush data to disk at appropriate time, which improves the system performance a lot. The NFS consistency semantics are designed to preserve data safety since the file server runs on a remote host. However, SVFS server runs on the same host as the SVFS client. The strict consistency semantics can be loosen to leverage Ext3 style lazy disk updates to expedite small file updates. This will be one potential improvement for our future work.

The current implementation of SVFS is unoptimized. Moving the VRPC implementation from NFS version 2 to version 3 should help improve performance. The use of aggressive meta-data caching and update aggregation mechanisms (used by iSCSI) could further improve performance when creating many small files [26].

9 Conclusion and Future Work

Preventing the modification of sensitive system files is an important operating system security problem. In this paper we proposed SVFS, a secure virtual file system, for managing access to sensitive files. SVFS works by storing sensitive files in a dedicated data virtual machine. Accesses to those files from other guest virtual machines must go through SVFS and are subject to access control policies. As the file protection mechanism resides in an isolated VM, it is hard to be bypassed or disabled, which allows SVFS to continue to work even if a guest VM is compromised.

To evaluate the effectiveness of SVFS, we tested it against several rootkits and backdoors. SVFS was successfully able to prevent six of nine rootkits that we tested from installing, and it was able to prevent the other three from persisting past reboot. Also, we evaluated the performance of two SVFS implementations compared to the native Ext3 filesystem. We found that using an improved virtual remote procedure call mechanism, SFVS was able to run a kernel compilation benchmark with only an 8% performance penalty compared to the local Ext3 filesystem.

SVFS currently enforces security policies on requests according to the guest VM making these requests. One future direction is to integrate the VM states such as keyboard events to help determine whether a request is malicious or not. We also plan to explore an intrusion detection system that can leverage SVFS accessing log to generate alerts when it sees file access patterns that indicate a known or potential attack.

References

[1] Kernel brk() vulnerability. http://seclists.org/lists/bugtraq/2003/Dec/0064.html.

[2] A New Adore Rootkit. *http://lwn.net/Articles/75990*.

[3] Phreak Accident. Playing hide and seek, UNIX style. *Phrack Magazine*, 4(43), 1993.

[4] M. Aslett. A virtual success. In *Computer Business Review Online*. *http://www.cbronline.com/content/COMP/magazine/Articles/Servers_Mainframes/AVirtual-Success.asp*.

[5] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[6] J. Beale. Detecting server compromises. In *Information Security Magazine*. TechTarget, Feb. 2003. *http://infosecuritymag.techtarget.com/2003/feb/linuxguru.shtml*.

[7] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM Press.

[8] D. P. Bovet and M. Cassetti. *Understanding the Linux Kernel (Ed. A. Oram)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[10] Gerald Carter. Samba. In *LISA*, 2004.

[11] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[12] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code. Technical Report 20742, IBM T.J. Watson Research Center, 1997.

[13] J. Dike. A user-mode port of the Linux kernel. In R. Spenneberg, editor, *Proc. of the 4th Annual Linux Showcase & Conference*, Oct. 2000.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, February 2003.

[15] I. Goldberg, D. Wagner, R. Thomas, and E.Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th Usenix Security Symposium*, pages 1–13, San Jose, CA, USA, 1996.

[16] R. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[17] H.-J. Höxer, M. Waitz, and V. Sieh. Advanced virtualization techniques for FAUmachine. In R. Spenneberg, editor, *11th International Linux System Technology Conference, Erlangen, Germany, September 7-10, 2004*, pages 1–12, 2004.

[18] G.H. Kim and E.H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *the 2nd ACM Conference on Computer and communications security*, pages 18–29, 1994.

[19] Greg Kroah-Hartman. udev c a userspace implementation of devfs. In *Proceedings of the Linux Symposium*, 2003.

[20] P. Leach and D. Perry. Cifs: A common internet file system. *Microsoft Interactive Developer*, November 1996.

[21] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proc. of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, 2001.

[22] A. Morton. Using the ext3 filesystem in 2.4 kernels. *http://www.zip.com.au/ akpm/linux/ext3/ext3-usage.html*.

[23] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[24] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *USENIX Security Symposium. USENIX Association*, pages 137–152, Aug. 2003.

[25] N. Provos. Improving host security with system call policies. In *Proc. of the 12th Usenix Security Symposium*, pages 257–272, Aug 2003.

[26] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P.J. Shenoy. A Performance Comparison of NFS and iSCSI for IP-Networked Storage. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 101–114, 2004.

[27] R. Sandberg. The Sun Network Filesystem: Design, Implementation, and Experience. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 300–316. IEEE Computer Society Press, Los Alamos, CA, 1992.

[28] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. of Operating Systems Design and Implementation (OSDI)*, pages 165–180, 2000.

[29] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[30] T.Y. Ts'o and S. Tweedie. Planned extensions to the Linux Ext2/Ext3 filesystem. In *Proc. of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–244, 2002.

[31] A. Whitaker, M. Shaw, and S.D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[32] H. Xie. Build a Secure System with LIDS. *http://www.lids.org/document/build_lids-0.2.html*, 2000.

[33] J. Zhou and L. Qiao. Backdoor and Linux LKM rootkit - smashing the kernel at your own risk. Project report: *http://citeseer.ist.psu.edu/giffin04efficient.html*.