Towards Reducing Unauthorized Modification of Binary Files*

Glenn Wurster Paul C. van Oorschot
Carleton Computer Security Lab
School of Computer Science
Carleton University, Canada

Abstract

We consider the problem of operating system and application binaries on disk being modified by malware. We present a file-system protection mechanism designed to protect the replacement and modification of binaries on disk while still allowing authorized upgrades. We use a combination of digital signatures and kernel modifications to restrict replacement without requiring any centralized public key infrastructure. To explore the viability of our approach, we implement a prototype in Linux, test it against various rootkits, and use it for everyday activities. The system is capable of protecting against rootkits currently available while incurring minimal overhead costs. Our design motivates general recommendations for kernel design to improve security, including restricting currently exported kernel interfaces, and conditions related to the granting of privileges for configuration activities. We do not protect configuration files, instead focusing on establishing a beachhead through protecting binaries the user does not modify.

1 Introduction and Overview

In current computing environments, applications written by many different authors all coexist on disk, being installed at various times by the user. Each application normally includes a number of program binaries along with some associated libraries. While the installation of a new application will normally not overwrite previously installed binaries, permission to modify all binaries is common. Indeed, this raises a problem: *Any application installer (or even application) running with sufficient privileges can modify any other application on disk.* Applications installers are routinely given these privileges during software upgrade or install (e.g. almost all installers run as administrator or root, giving complete access to the system). Some applications even run with administrator privileges during normal operation due to a variety of reasons despite the best efforts and countless recommendations against this practise over the years. The common running of applications (including their installers) as administrator leads to a situation in which a single application can modify any other application binary on disk. Normally, applications do not use this privilege to modify the binaries belonging to other applications. Malware, however, does not traditionally respect the customs of normal software, and uses the ability to modify other binaries as a convenient installation vector. Already in 1986, the Virdem virus [41] was infecting executables in order to spread itself; more recently, rootkits [13] have used binary modification in an attempt to hide.

In this paper, we describe a file-system protection mechanism designed to limit modifications to binaries on disk. We focus on establishing a beachhead by protecting binary files – a type of file very rarely modified by a user. Our proposal, however, can be extended to other types of files not modified by the user of a system.

Our approach is to associate with each library file and executable a digital signature of the file which is checked by the kernel when performing an update to the binary. In essence, the binary is self-signed; no centralized (or other) public key infrastructure is involved. We use additional kernel protections to restrict modification of these signed binaries on a system. The core technology is based on a simple application of code-signing or self-signed executables [50]. We use the term *bin-locking* (short for binary-locking) to refer to our proposal, to avoid confusion with other code signing mechanisms. The proposed system allows binaries to be easily upgraded, facilitating the

*Version: September 14, 2009. This paper expands on a preliminary short paper [50].

application of security patches.

In addition to introducing the bin-locking system, we also discuss the surrounding architecture required to make the system secure; for example, we require trust in the kernel. We design and implement a prototype bin-locking system, using it as the underlying platform to perform common activities (e.g. watch videos, listen to music, browse the web, read e-mail) as well as write this paper; the additional overhead imposed is imperceptible to end-users. While the prototype is developed in Linux, the design itself is generic – we see no reason why it could not be implemented on Windows or MacOS. The proposal is not intended for use in all environments, for example, it does not co-exist well with some developer features for reasons discussed in §3.4. Its main target audience is the vast majority of common end-users, who do not use development environments or related tools and are not security experts.

The proposed system reduces the ability of malware (including rootkits) to hide on a system; system binaries can no longer be modified to hide the presence of malware. Anti-virus system files can similarly be protected against modification by malware. We stop short of attempting to prevent malware from being installed or run on a computer, concentrating as a first step, on the isolated problem of preventing the modification of binary files. While configuration files remain unprotected by the core bin-locking system discussed in this paper, the beachhead established by bin-locking may make future protection of configuration files easier.

Our design and implementation leads to important conclusions. Two that we discuss are (1) the current lack of separation between the kernel and user space is a legacy feature which requires very serious reconsideration from a security perspective; and (2) bin-locking has the potential to improve the usability of schemes which segregate configuration operations.

In Section 2, we introduce the bin-locking system. Section 3 discusses a prototype implementation and its evaluation, including how it successfully blocked the install of a variety of Linux rootkits. Section 4 provides further discussion and general recommendations for kernel design. Section 5 discusses similar proposals and related work. We summarize our results in Section 6.

2 Main Proposal: Restricting Updates by Bin-Locking

At the core of our proposal is a simple but well-planned use of digital signatures, designed to protect a binary against unauthorized modifications. To restrict who can modify (or replace) a binary on disk, we enforce one simple protection rule: *A library file or executable on disk can only be replaced by a library or executable containing a digital signature verifiable using a public key in the previously installed binary having the same file name.* This basic idea was presented in a 5-page workshop paper [50]. Binaries protected by the bin-locking proposal have embedded within them a set of digital signatures along with a set of corresponding public keys.[1] We propose supporting a few standardized digital signature algorithms (the particular choice is specified alongside and protected by the digital signature). The kernel, upon finding a digital signature (in the bin-locking section of the binary), will restrict replacement of the binary to those new binaries containing a valid digital signature which can be verified using a key in the currently installed binary. If the signature can be verified (or the currently installed binary is not bin-locked), then the replacement is allowed. Otherwise, the replacement is denied by the operating system and the original binary remains unmodified on disk. Deployment is incremental – binaries not bin-locked can be replaced without restriction (which is how most systems currently operate), but once a binary is bin-locked it must be replaced by a bin-locked binary. The proposed method is quite different than SDSI/SPKI [15, 37]; we trust keys only in a very limited setting (replacing a binary which was signed with the same key). We do not use names or certificates.

Over time, inevitably, signing keys used for bin-locking will be lost or compromised. Both situations can be ameliorated by allowing, as an option, the embedding of multiple verification keys in a binary file. If one key is lost, the other key(s) can be used to sign a subsequent version of the file (which can also introduce new keys). We do not specify any conditions on who or what controls the private keys corresponding to these additional verification public keys, but many options exist including community trusted organizations or trusted friends who function as back-

[1] The portion of the file holding the digital signature itself is not included in the range of the signature, to prevent a recursive definition. The public key, however, is integrity-protected by the signature.

ups. While we mandate no specific infrastructure for key revocation, pro-actively installing a new version of a file which does not allow future versions signed with the previous key (i.e. which excludes the old verification public key(s) from those embedded in the new version) prevents a compromised key from being a threat indefinitely. Because each file can be signed with a different key, the effect of a compromised key can be limited.

2.1 Trust Model

In the proposed system, it is assumed that a malware author does not have physical access to the machine and that malware does not have access to the private signing keys, nor have kernel level control of the system being infected.[2] While the first two assumptions are fairly standard, the third assumption may currently raise alarm bells amongst many readers. On current systems, any application running with root (or administrator) access can modify the running kernel. Such an application can also bypass the file-system interfaces exported by the kernel by writing directly to the raw hard-drive. To justify the assumption of being able to trust the kernel, we both discuss (in §2.3) and implement (in §3.2) several restrictions which lock down the interface between a running kernel and the superuser. We believe that protecting the kernel in this way is beneficial, independent of bin-locking. Malware exploitation of the kernel is a growing trend [4] and others in the security community have already made an effort to protect the kernel [33, 48, 26, 17, 34]).

2.2 Benefits of Bin-Locking

Previous proposals which attempt to limit changes to binaries on disk all apparently fall short for one of three reasons: they either detect changes after they have happened [22, 28, 2, 46] (making recovery hard), rely on the user to correctly validate every file modification, or still allow applications to modify any file at all during install/upgrade [46]. Bin-locking addresses these three points and yields additional benefits as follows.

1. No Central Key Repository or infrastructure. The proposed system differs from many other code-signing systems currently in use in that it does not attempt to tie the signature to an entity. It can verify that the new version of an application binary was created by the same author (or organization) as the old version without knowing who the author (organization) is. Because the signature on a to-be-installed binary file is verified using the public key embedded in the previous version of the same file (by file name) installed on the system, there is no need to centrally register a key or involve any central repository. Thus, *no central certification authority or public key infrastructure (PKI) is required.* An application author can create a signing key-pair and begin using it immediately. Furthermore, if desired, different keys can be used for each file on a system, limiting the impact of a key compromise (as long as all private keys are not stored in one place the attacker gains access to, the attacker incurs a per-executable cost for replacing protected binaries). Because there is no dependence on a centralized trusted authority, development of new software remains unrestricted. We make no effort to restrict the software which can be installed on a system, as long as that software does not modify already-installed binaries. Other digital signature schemes proposed in the past have relied on a trusted central authority [2, 46].

2. Trusted Software Base Even After Compromise. The operating system and core applications are normally installed before malware has infected a machine. We exploit this temporal property. Typical malware, because it is installed after the operating system (including core libraries and programs), is not capable (if the proposed system is in place for operating system files) of changing any of the operating system files. The operating system files, therefore, can be trusted to be unmodified. If an anti-virus system is installed before any malware, the anti-virus binaries can also be automatically protected using the same mechanism. Core binaries on a compromised system can therefore be trusted, allowing much greater control over an infected system without requiring a reboot to clean media. Furthermore, this ability to reliably trust binaries on the system can make recovery easier [19]. Using the proposed system, anti-virus software can be protected against modification and system binaries can be relied upon with confidence in their integrity, restricting the ability of malware to hide. In the case of forensic analysis, while administrators may still choose to reboot to known-clean media once they discover malware, the inability for malware to hide is

[2]We define the kernel (and hence kernel level control) to include only those aspects running with elevated CPU privileges (ring 0 privileges on x86) – this definition of kernel does not include core system libraries installed alongside the operating system but run in user space.

likely to result in the administrator becoming aware of the problem earlier.

3. Incremental Deployability with Incremental Benefit. Kernels which do not yet support the bin-locking mechanism treat signed binary files as normal binary files, and are unaffected by the existence of a digital signature. Similarly, binaries without bin-locking digital signatures are allowed on a kernel which supports the proposal (in contrast to most proposed code signing schemes [46]). Either the kernel or libraries can be modified first without an adverse affect on non-supporting systems.

4. Low Overhead. As discussed in §3.7, the prototype has imperceptible overhead to the end-user.

5. Simplicity. The bin-locking proposal is relatively simple to understand, and does not require any additional hardware or co-processors.

2.3 Kernel Modifications

To ensure that bin-locked files remain visible, we must ensure that a new file-system is not mounted over top of bin-locked files, and that a file-system containing bin-locked files is not unmounted unexpectedly. We first recognize that the mounting and unmounting of file-systems is not commonly performed (or at least does not affect core system directories) after system boot. We therefore extend the kernel to prevent mounting and unmounting of file-systems on specific paths. In the prototype, the list of such paths to protect is fully customizable by the user or machine administrator, being set as part of the boot process (however once a path is specified, it cannot be removed from the list of specified paths). We discuss the specifics of the prototype more in §3.2.5.

To ensure that the proposed protection mechanism cannot be subverted, we must also disable raw disk access to those partitions containing bin-locked binary files. Raw disk access allows privileged processes to write directly to the drive, bypassing the restrictions associated with files on that drive. We disable raw disk access in a way similar to mounting as discussed above. The kernel accepts a list of devices to which writes should not be allowed. By specifying the partitions containing the core system libraries, we force any updates to the files to go through the proposed bin-locking system. We discuss the implementation details of this in §3.2.4. The disabling of the bin-locking system by modifying the running kernel is discussed in §3.2.3.

The ability to restrict updates to only those files which are bin-locked with a verifiable key also presents another opportunity. On many systems, the user of the system is not the same individual as the administrator (especially in business and educational environments). By allowing updates to bin-locked applications, we can safely allow users to install patches without granting them administrator access. Bin-locking provides the mechanism to ensure a user only updates bin-locked binaries with new valid bin-locked binaries. Of course, granting this ability is not appropriate in all environments, and hence we propose still using traditional file access control policies to dictate which users have permission to update a file in the first place. In the prototype, we continued to enforce traditional access control policies in addition to those of the bin-locking system.

If deleting application binaries were still allowed, the bin-locking system would be rendered ineffective; the attacker could simply delete the binary and then install a new one. To delete bin-locked files in the proposed system, the bin-locking protections must be disabled. In the prototype, this requires a reboot into a kernel which does not enforce the protection mechanism (see §3.6). Previous work on providing a trusted interface (e.g., see [52]) – one which cannot be subverted by malware – between the kernel and the user may help to eliminate the reboot requirement. One solution (not implemented in the prototype) is to tie enforcement of bin-locking to whether or not a hardware token is inserted (similar to that presented in [8]) – as long as the hardware token is inserted, unauthenticated moves and deletes would be allowed.

2.4 Limitations

We now discuss some limitations related to deploying the proposed system.

1. Denial of Service. Any attempt to install an application after malware is already on the system and has written bin-locked binaries (with different signatures) to the file names required by the application will result in a failed install. We actually view this 'limitation' as an advantage from a security viewpoint, although not all users will find that this improves usability. Currently, many users continue to use a computer after it has been infected with malware. They either are not aware that malware is there, or are not aware of the full implications of malware being on the system. Malware initiating such a denial of service attack will illicit a forced user response when an application install is prevented. At this point in time, the desire of the user to install an application is a security benefit in encouraging

them to clean their system and remove the malware, including the files which resulted in the denial of service. We note that in order to take advantage of their desire to install an application, the process of removing the malware (including the reboot required in the prototype system) must be as simple as possible; the usability impact of this makes the proposal more suitable for some user environments than others.

The same user response is forced by malware performing a denial of service through filling the hard drive with bin-locked files which cannot be deleted (without a reboot). Both malware actions result in a state where the user (or their technical support team) is aware and forced to take action on installed malware. As long as malware continues to attempt to hide, it seems unlikely that either denial of service will be actively exploited.

2. Signed Binary File Deletes/Moves. With the bin-locking system in place, file deletion and movement become much more complex. We cannot allow a bin-locked file to be easily moved or deleted since this would open up a method for allowing file replacement. We note, however, that application (and operating system) binaries are rarely moved or deleted on a system. Furthermore, we believe (and personal experience seems to support) that users rarely uninstall applications. For those times where an uninstall or move is required, a reboot into a different kernel would allow the operations to be performed. Again, we acknowledge that the usability impact of this makes the proposal in its present form more suitable for some user environments than others. We discuss reboots as they relate to the prototype implementation in §3.6.

3. Kernel Interface Lock-down. In order for bin-locking to successfully resist attacks, the kernel needs to have several dangerous interfaces locked down (as discussed in §2.3). In the prototype implementation, the new kernel restriction which influenced applications the most was disabling raw disk writes. Applications affected by locking down this kernel interface include file-system repair utilities, disk partitioning, and disk formatting utilities. In the discussion, we concentrate on when these tools are used to *write* to partitions containing bin-locked files – the only partitions protected by the prototype (as discussed in §2.3). File-system repair utilities are only run on these partitions during boot, being accounted for in the prototype by disabling raw writes later during the boot process. Partitioning and formatting utilities are very destructive in nature and hence arguably should not be allowed to make changes to core partitions during normal operation of a system. We believe blocking the destructive power of file-system utilities during normal system operation to be an acceptable (even beneficial) security practise. When file-system utilities need to modify core partitions, a reboot into a kernel which does not enforce bin-locking is sufficient to enable access. We note that the prototype implementation did not restrict file-system operations on removable media (e.g. USB flash keys).

During development and use of the prototype (including watching videos, listening to music, browsing the web, reading e-mail, and writing this paper), we did not encounter any programs that were blocked by the other kernel interface restrictions implemented as part of the prototype. To our knowledge, no program attempted to either mount or unmount a file-system over core directories. We also did not encounter any program (other than malware) which attempted to write directly to swap or kernel memory. Note that Windows Vista already implements many of the kernel access protections discussed here, including restrictions on write access to raw drive partitions, swap, and kernel memory [30, 29, 40].

4. Aliases. The proposed protection mechanism only protects binaries on disk. If malware can prevent the correct binary on disk from being invoked, then it retains a measure of control over previously installed programs. As an example, running the `ps` command from the prompt without a pre-pended path (i.e., fully qualified file name) will cause the first copy of `ps` found to be run (even though it may not be the `/bin/ps` binary). While the bin-locking scheme is designed primarily to protect binaries against modification, these binaries are of no use if they are not used. We must ensure therefore on an infected system that the legitimate binary can be easily run instead of one found at a location of the attacker's choice. Additional copies of binaries installed by malware can be avoided by running applications from the trusted base directly, avoiding the environment. There are a number of methods for accomplishing this, including calling the kernel directly (e.g. using the `execve` system call) to run a program. Because much of the aliasing functionality is implemented by libraries likely to be protected by the bin-locking scheme, some aliasing vulnerabilities can be avoided (e.g., by restricting `PATH` to include only core system directories when running as root).

2.5 Extensions to Bin-Locking

Assuming a kernel has the basic capability of verifying that binary updates are authorized, other extended functionality may be worth considering. While we did not implement these in the prototype, we believe them to be useful extensions to the core idea; those discussed here would require additional support from the kernel.

2.5.1 Versioning

As one possible extension to the system, version numbers could be embedded in both the old and new binaries. If the kernel limits replacement based on version number, the same key could be used over an extended period of time without the risk of a downgrade attack (i.e. replacing a more recent binary with an older version containing a vulnerability). While authors (or organizations) can achieve the same effect by 'revoking' keys (as discussed at the top of §2), versioning allows the software author to minimize the number of keys which must be contained in any new version of the binary while still ensuring that the binary can replace many previous versions. We acknowledge that rollback (the process of reverting software to a previous version) may not be possible while bin-locking enforcement is active on a system (since bin-locking, as outlined in this paper, is designed to also prevent downgrade attacks).

2.5.2 Sub-Keying

In the core idea, any binary which can be validated using keys in the installed binary can replace the installed binary. To prevent one binary from replacing another totally different binary by the same organization (preventing their software from working properly), the organization could use a non-overlapping set of keys for each binary. As an extension to the basic bin-locking idea, an organization could embed an index number into each binary they sign. While new versions of the same binary would have the same index number, different binaries by the same organization would have different index numbers. If the kernel enforces that the index number between the old and new binary match, an organization could use the same private signing key for all their binaries (without allowing their binaries to be switched on a system). As an example, sub-keying could be used to prevent `rm` from replacing `ls` while allowing the two binaries to be signed with the same key.

3 Prototype Implementation and Evaluation

To verify the viability of the bin-locking proposal, we modified a system to implement bin-locking, including the kernel interface restrictions. The prototype implementation is composed of a number of different pieces which all work together to protect the system. We wrote a binary signing utility which is used along with associated custom scripts to sign the binaries in the Debian software archive (for Debian 4.0), creating a new local mirror. We then installed these binaries on a test system using the Debian package manager which we modified to support bin-locked binaries. The Linux kernel (version 2.6.25) on the test system was modified to enforce the proposed protection mechanisms (which include restrictions on bin-locked binaries as well as access to the kernel and file-systems). The boot process was modified on the test system to initialize kernel data structures which limit raw writes and mounting. We now discuss each of these elements in detail.

3.1 Extensions to the ELF file format

Executable files for a particular operating system normally follow a standard structure. Most Unix distributions (including Linux) use the binary format file ELF (Executable and Linkable Format). The basic ELF file is represented in Figure 1. Except for the ELF file header, all other elements are free to be arranged as desired. We modified ELF files (our approach could be adapted to other types of files not modified by the user – e.g. Windows executables, Windows libraries, or application data files), creating a new type of section for the purposes of storing bin-locking related data. The ELF binary file format was designed such that applications could create new sections and many other applications (e.g. GCC and bsign) take advantage of this flexibility.

The bin-locking section of the ELF file is made up of one or more records (the section table contains a field specifying the number of records), with each record containing a type of digital signature (e.g., all elements related to the DSA algorithm would be in one record). Each record specifies a signature, prefix of the key used to generate the digital signature, and zero or more keys which can be used to verify digital signatures of the same type in subsequent versions of the binary. The key prefix record contains the first four bytes of the public verification key related to the

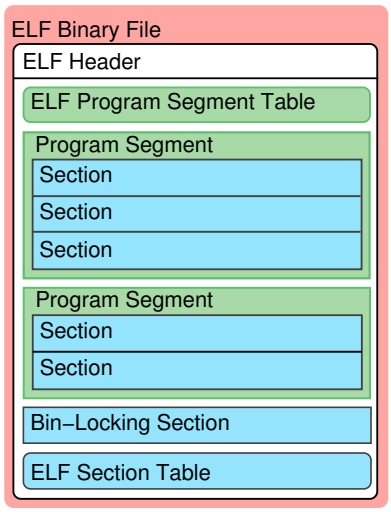

Figure 1: Basic ELF Layout including Bin-Locking section.

digital signature and is used for quickly determining what verification key in a previous version of the binary should be used for verifying the digital signature (if multiple keys share the first four bytes, the kernel will attempt to verify with each). We illustrate the layout of the bin-locking section in Figure 2.

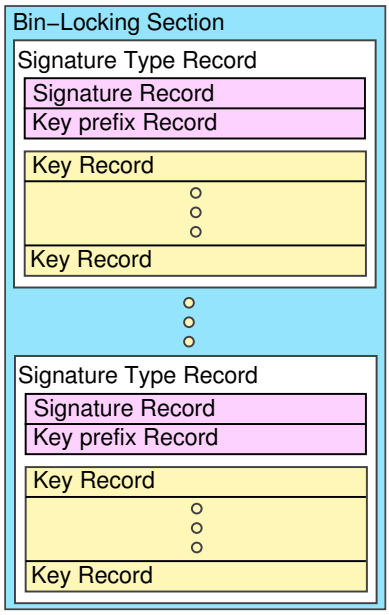

Figure 2: Bin-Locking File Section Layout

To allow for future signature schemes, we included several key components in bin-locking section headers. The header for records and sub-records was specified to include both a length and type field, allowing the kernel to skip over unrecognized signature types. The sub-record header, in addition, contained a flag which allowed the kernel to skip its contents when hashing the file (even if the kernel did not recognize the record type) – signatures were stored in sub-records with this flag set. We discuss the kernel verification of digital signatures more in §3.2.2.

3.2 Kernel Modifications

The kernel was modified to enforce bin-locking as discussed in §2. Signed binary files could not be deleted, moved, or opened for writing. They could only be replaced with new binary files which contained a signature verifiable using keys in the old binary. On a replacement request (which is initiated through a move system call involving a bin-locked binary), the kernel attempts to extract the keys from the old binary and use them to verify the validity of the new binary. If the signature in the new binary successfully verifies, the kernel moves the new binary over top of the old binary. Using the move system call, however, presents a dilemma. The new binary must not be bin-locked (or the move will be denied), but at the same time it must be bin-locked (to replace the old binary). We deal with this by ignoring the first eight bytes of the new binary when performing movement validation checks (a feature used by the modified Debian package manager discussed in §3.3). The first eight bytes change the file such that the kernel does not recognize it as bin-locked (for the purposes of preventing its movement and deletion) but still recognizes it as bin-locked (allowing it to replace another bin-locked file if the signature verifies). During the replacement, the kernel will first verify the signature on the new binary (ignoring the first eight bytes). If signature validation passes, the kernel will move the new binary over top of the old one, removing the initial eight bytes during the move (the file is locked to prevent it from being modified between signature validation and the move operation). The resulting bin-locked binary after the move is protected by the kernel. Because binaries currently being run cannot be modified on disk (in any Linux system), a move must be used (instead of a copy or other update mechanism) to replace bin-locked binaries. The kernel retains backwards compatibility with binary files that are not bin-locked, not restricting their replacement or removal. Currently, the kernel can verify application binaries signed using the Digital Signature

Algorithm (DSA).

We chose not to rely on a user space helper in designing the prototype system in order to simplify the implementation. While we believe a user space helper could be implemented securely, it would need to be bin-locked itself, and the interfaces it uses to talk to the kernel would have to be designed very carefully to avoid them being taken over by malware.

3.2.1 Detecting Bin-Locked files

To detect whether or not a file was bin-locked, the modified kernel examined very specific elements in the file. It verified that the file was 1) an ELF file [14], 2) contained a bin-locking section, and 3) the bin-locking section contained a digital signature in a format known to the kernel (currently, only DSA signatures are supported). To determine if a file is bin-locked, we needed to read the file header (the first 52 bytes of the file on a 32bit x86 platform) as well as the section table (40 bytes per section). If any element in either the file header or section header was considered invalid according to the ELF specification [14], the file was treated as not bin-locked. An attacker is not able to turn a bin-locked file into one not bin-locked because the kernel does not allow a properly bin-locked file to be altered (except by replacing it with another properly bin-locked file). The eight byte header described above results in a file which is not recognized as a valid ELF file and hence the modified kernel allows it to be removed, modified, and moved. We assume individuals attempting to bin-lock their own binaries will not purposely create invalid binaries (as that would negate the effort of bin-locking the binary in the first place). As part of the prototype, we created a tool to test that ELF files are recognized as valid and correctly signed. The overhead of enforcing bin-locking, while imperceptible to end-users of the running system (a Pentium 4 at 2.8GHz with 1G of RAM), is discussed more in §3.7.

3.2.2 Verifying Digital Signatures

To verify that a new binary can be installed, the modified kernel first extracts out a list of verification keys from the old binary (which share the same prefix as the key prefix stored in the new version of the binary alongside the digital signature). For each verification key that matches the key prefix, the kernel attempts to verify the digital signature using the verification key. If the digital signature check passes, the replacement is allowed. If the binary contains two different types of digital signatures, the modified kernel will allow replacement if any one of them contains a digital signature which validates.

While the prototype used an older and less efficient implementation (space-wise) to store and verify digital signatures than described herein, the method proposed in this sub-section is preferred.

3.2.3 Avenues for Kernel Modification

To protect the bin-locking system itself, the kernel was also modified to remove functionality which could be used to attack the system. The main such attacks are modifying the kernel to disable the protection scheme, editing the bin-locked binaries directly on disk, and hiding bin-locked binaries (by either mounting or unmounting the partitions they reside on). To prevent disabling the scheme, `/dev/kmem` was disabled and `/dev/mem` was restricted to only allow access to non-RAM physical addresses (in parallel and independent of our proposal, this protection was introduced in version 2.6.26 of the Linux kernel). `/dev/mem` cannot be disabled entirely as X (the graphical display manager) uses it to communicate with the video card (by restricting access to `/dev/mem` instead of modifying X, we do not require X to be trusted alongside the kernel). While we disabled module loading entirely, a better option is to deploy module-signing (as discussed in [25]). We also limited raw disk access and drive mounting (as discussed below). While certain hardware configurations may provide additional methods of gaining write access to kernel memory in the prototype, we believe kernel device drivers can be modified to remove vulnerabilities caused by specific hardware; this requires further exploration beyond that done in the prototype implementation.

3.2.4 Disabling Raw Disk Access

To protect bin-locked binaries against modification, one must also disable raw writes for partitions that contain bin-locked files. We did this by exporting a syscontrol from the kernel, allowing a user space process to set which partitions should prevent (disable) raw disk writes. A syscontrol is a single pseudo-file (a file which does not exist on disk) which exposes kernel configuration to user space. In this case, the list of protected partitions can be read, and a new partition can be appended to the list by writing to the pseudo-file. Because the syscontrol only supports appending to the list maintained by the kernel, the only way

to remove a partition from the list is to reboot the system, clearing the list. As part of the boot-up process, the list of partitions for which raw disk access is disabled is written back into the syscontrol (after the initial `fsck`/file-system check). This list of partitions we wrote to the syscontrol included the swap partition (to prevent attacks against kernel memory [40]). If any partition on a disk is protected, the prototype kernel disables raw writes to the file representing the entire drive. In order for malware to enable raw disk writes, it must modify the start-up process to disable initialization of the syscontrol and reboot the system. One solution to prevent this is to initialize the syscontrol in `init` (the first binary run). Binaries involved in the start-up process (including `init`) can be bin-locked, preventing modification.

In the implementation, the restriction on raw disk writes was implemented as a user-specified list because the kernel could not determine quickly what partitions contain bin-locked files. As a usability improvement, the file-system could be modified to include a flag indicating the presence of bin-locked files on that partition. If bin-locked files are present, then raw writes to the partition could be automatically disabled without the kernel needing a list. By avoiding file-system modifications, the prototype was able to operate at the security module layer [49], not depending on a particular file-system. By leaving the file-system unmodified, backward compatibility with systems not aware of bin-locking is also maintained.

3.2.5 Restricting Mounting

To prevent bin-locked files from becoming inaccessible, we restricted the locations where file-systems can be both mounted and unmounted using the same approach of a syscontrol which supports both read and write operations. By writing "`< /usr/lib`" to a syscontrol created by the prototype, the modified kernel enforces that no file-system can be mounted or unmounted at `/usr/lib`, `/usr`, or `/`, meaning that all files in `/usr/lib` continue to be accessible until the system is rebooted. By writing "`> /usr/lib`", no file-system can be mounted on any sub-directory or parent directory of `/usr/lib`. File-system root rotations are also not permitted by the prototype if the syscontrol restricting mounts has been written to. Although both the new syscontrols support write operations, all writes to these syscontrols are converted to appends by the modified kernel and hence cannot be used to modify previous entries written to the bin-locking related syscontrols. Remounting partitions to enable and disable write access must be allowed, as this functionality is used during the shutdown process to avoid file-system corruption.

We currently see no easy method of avoiding the list of mount location restrictions. Administrators may require unmounting on devices containing bin-locked files (e.g., unmounting removable media). While it is possible to prevent mounting a new file-system over bin-locked files using a file-system flag (as discussed above), whether to prevent file-system unmounts depends on the environment.

3.3 Modifications to Executable Files

To bin-lock binary files, we used binary rewriting. We created an application which would use one or more signing keys to sign an existing binary, injecting into the binary both the signatures and all the verification public keys related to the signature (we chose not to use bsign [7], preferring to keep the kernel code as simple as possible). ELF files are used for both program executables and shared libraries – the bin-locking approach covers both. Currently, the signing application signs binaries using the Digital Signature Algorithm [16], although this can be extended to other signature formats. The modification of executables is backwards compatible. Signed (i.e., bin-locked) executables can be used seamlessly on a system which does not understand bin-locking.

We modified the Debian package manager [12] to not write out bin-locked binaries to temporary files during the installation of the system (since once written into a temporary file, the modified kernel will not allow the file to be moved or deleted). Instead, the package manager writes out an eight byte prefix (we used the prefix `CODESIGN`) followed by the signed file (which is recognized as such by the modified kernel during replacement of the original signed file, as mentioned in §3.2). The binary rewriting application was used along with several additional scripts to create a local Debian 4.0 mirror [9] where every application binary and library was bin-locked.

One element in the standard Debian boot process initially posed an issue for our bin-locking process. During the boot process, the temporary initial RAM disk (a file-system within RAM which stores files used early in the boot process) is deleted because it is no longer necessary. If this initial RAM disk contains binaries which are bin-locked, the new kernel prevents the delete. To overcome this, bin-locking is disabled on drives not associated with a physical device.

As partial evidence that the modified kernel and

signed executables are viable, this paper was written on a test system (which used the prototype implementation). On the system, all binaries and libraries were bin-locked and kernel restrictions were active. The test system, while running KDE (the graphical based K Desktop Environment) was also used to browse the web, write e-mail, listen to music, and view video – all with no noticed differences from an ordinary system (with the exception of differences caused by bugs in early versions of the prototype implementation). This confirmed that it is possible to lock down the kernel interface, as well as use bin-locking on a deployed system.

3.4 Bin-Locking on Developer Systems

While developers are not the target audience of the bin-locking system, we nevertheless discuss several developer features which currently remain enabled on all systems and undermine the security of bin-locking. We propose disabling these features on systems using bin-locking (e.g., end-user desktops) in order to increase security. We believe that typical end-users (as opposed to developers) will not be bothered by the following restrictions.

1. Ptrace hooks are used by developers to debug a running application. By allowing reads and writes to a process memory space, arbitrary changes to both data and code within the running application can be made. In order to ensure bin-locked binaries are run unmodified, ptrace access needs to be disabled for bin-locked binaries.

2. The use of custom builds by third parties is made much more difficult by bin-locking. The modification of binaries by third parties, however, is exactly the type of attack that bin-locking aims to prevent. Bin-locking ensures the software run by end-users is never modified by anyone other than those who developed the software. Developers wishing to switch between original software and custom builds (e.g., during debugging) will not be able to take advantage of the benefits of bin-locking for those binaries.

3. Preloaders such as `LD_PRELOAD` allow additional libraries not specified in an executable to be linked in at run-time. The use of pre-loaders provides a method for modifying a binary at run-time. There are two ways of preventing this from being exploited by attackers. The first (and easiest) is to disable `LD_PRELOAD` on non-developer machines (e.g., by shipping a bin-locked `/lib/ld-linux.so.2` not implementing the feature). The second defence comes through recognizing that `LD_PRELOAD` must be processed (indirectly) by the binary during start-up. While most applications leave the functionality intact (i.e., by calling the default `/lib/ld-linux.so.2`), the application developer can disable the functionality on binaries they intend to bin-lock.

3.5 Protection Against Current Rootkits

To verify that the bin-locking system was able to defend against rootkit malware, we attempted to install several Linux rootkits.[3] Linux rootkits can be grouped into two categories. The first category is those that use some method to gain access to kernel memory, installing themselves in the running kernel. These rootkits then operate at kernel level, hiding their actions from even root processes. The second category consists of rootkits that replace core system binaries. These binaries are often used by the root user in examining a system. Both classes of rootkits work to hide nefarious activities and processes on a compromised system.

We selected six representative Linux rootkits, two that modify the kernel and four that replace system binaries. Both kernel-based rootkits (`suckit2` and `mood-nt`) failed to install because of disabled write access to `/dev/kmem`. The `mood-nt` kernel based rootkit which we tested also attempted and failed to replace `/bin/init` (in order to re-initialize itself on system boot); this replacement was denied by the modified kernel. The four binary replacement rootkits (`ARK 1.0.1`, `cb-r00tkit`, `dica`, and Linux Rootkit 5) were all denied when attempting to replace core system programs (e.g. `ls`, `netstat`, `top`, and `ps`). The bin-locking proposal provided protection against the modification of both application and system binaries. The fact that no rootkit was able to install is supporting evidence of expected functionality of the bin-locking system.

3.6 Reboots

Because the prototype requires a reboot to delete or move bin-locked binary files, the process of rebooting into a kernel which does not enforce bin-locking must be as usable as possible. We used the standard GRUB [31] boot loader to provide an option to the user as to

[3] All Linux rootkits tested were from `http://packetstormsecurity.org/UNIX/penetration/rootkits/`

whether or not to use the bin-locking enabled kernel; the user simply selects one of the non-enforcing kernels from the menu during boot. Once booted into an alternate kernel, the user may delete and move bin-locked files, including those installed by malware (as discussed in §2.4). An open problem is how to persuade users to normally use the kernel which enforces bin-locking. Creating a trusted interface between the kernel and user (e.g., see [52]) – one that cannot be subverted by malware – may help eliminate the reboot requirement; as discussed in §2.3, a hardware key is one such option.

3.7 Performance

Any performance impact of the proposed system was imperceptible to the end-user during the writing of this paper (and indeed while performing other common activities such as web browsing, video watching, and image editing). For more precise measurement, we ran benchmark tests to quantify the overhead of the system. Using the Perl benchmark library, we measured the average increase in kernel time required to perform a delete and move operation on both non-ELF and unsigned ELF files with an ext3 file-system. Over 25000 test runs (10000 with a small file, 10000 with a medium one, and 5000 with a large file), the average increase in time to delete or move a non-ELF file was 15.59% or 6.032μs when the file was cached. The time required to open a cached non-ELF file for writing similarly went up by 34.84% or 3.86μs. For unsigned ELF files, the overhead of deleting or moving the file increased by 26.77% or 13.3μs. The overhead of opening an unsigned ELF file for writing increased by 66.65% or 7.73μs. While these percentage increases are high for opening a file, the amount of physical time required to open a file remains small. In the interest of retaining file-system compatibility with kernels not enabling bin-locking, we chose not to optimize the overhead of moving, deleting, and opening bin-locked files. By reserving one bit per file on the file-system for indicating whether a file is signed or not, this overhead could be brought down to essentially 0%.

When the file was not cached, the time required to perform an update, move, or open varies with the speed of the hard drive. Determining if a file is bin-locked requires three additional hard drive accesses. The first disk read is to bring in the data block pointers [42], the second is to read the ELF header, and the third is to read the section table. With a delete, the data block pointers need to be read from disk anyway, resulting in bin-locking requiring an overhead of just two additional disk reads. During testing, the average increase in time required to both delete and move a non-cached and unsigned ELF file was 28ms. This overhead can also be brought down by tracking whether files are bin-locked on the file-system.

The cost of replacing a bin-locked file (i.e., the cost of validating the signature on a signed binary) is $O(n)$ in the proposed system (where n is the size of the file), an increase from $O(1)$ in a system not enforcing the protection mechanism. This overhead translates to 111.8ms on the test system for a 1M binary (with disk caching disabled[4] and using the older implementation as mentioned in §3.2.2). The overhead is apparently unavoidable, since the entire file must be hashed to verify a digital signature. We emphasize that this cost is only occurred during the install or upgrade of a bin-locked binary, not while performing normal tasks (i.e., executing an application).

4 Discussion and Recommendations for Secure Kernel Design

Here we discuss a few observations and insights that emerged over the course of designing and implementing the bin-locking system. While one would hope that these observations have been made before, we believe they merit broader discussion (and/or renewed discussion, as the case may be).

4.1 Kernel Separation and Kernel Interfaces Exported to Root Users

Traditionally, the distinction between the kernel and user space has been fluid (especially in Linux). Any user with root privileges has been able to modify kernel memory. There is, however, still a distinction between the privileges bestowed on the kernel vs. a user running as root. It is the kernel's job to provide an environment for the processes which run on the system (including those processes run as root). The kernel mediates memory and disk access for all processes. The Linux kernel has traditionally allowed root users to modify it arbitrarily through several exported interfaces – blurring the line between the

[4]Disabling disk caching involves writing other information to memory, causing the cached file to expire and be removed from disk cache.

privileges given to root users and those given to the kernel.

We believe that the current approach of allowing superusers to modify the kernel arbitrarily is a legacy ability which for security purposes requires very serious reconsideration. Closing this front door (so to speak) which allows root users to modify the kernel would provide security defenders with a better chance of addressing remaining kernel security violations and vulnerabilities through approaches such as introspection, integrity self-checking, and host-based intrusion detection [26, 17, 34, 4, 33, 3]. We believe, however, that the simplest method of restricting kernel exploitation is to remove the interfaces used to gain access to the kernel. Detection tools can then concentrate on those exploits that take advantage of kernel security vulnerabilities.

4.2 Separating Configuration Mode Privileges from Regular Use Privileges

The prototype bin-locking system imposes additional restrictions on applications, restricting their ability to modify the system. We are not the first to reduce an application's ability to configure the system. Most of the time, the system is not being configured but instead being used to accomplish some task. Many have recognized this distinction and attempted to differentiate between the two. Hardware keys [8], boot-time options, and separation of privileges [6, 35, 51] all attempt to protect system configuration from modification while performing standard tasks.

The separation of privileges, between system configuration and standard use, allows the lock-down of elements which should never need to be modified under normal system operation. Bin-locking has the potential to improve usability of systems which separate configuration privileges by allowing a safe subset of binary update operations without requiring user interaction.

5 Comparison with Related Approaches

We first compare bin-locking with several closely related proposals and then discuss other related work in §5.4. Table 1 focuses on Google Android [1], rootkit-resistant disks [8], Tripwire [21, 22], and using read-only media [23], comparing the approaches on whether they are proactive (prevent vs. detect modifications), accommodate upgrades (without extra end-user effort), the types of files they protect, and the granularity of protection. While only bin-locking and Google Android accommodate program upgrades, the table is not the full story; we now discuss differences in more detail.

5.1 Google Android

In parallel to and independent of our work (but subsequent to publication of our preliminary design [50]), Google introduced a signing approach [18] in the Android platform [1] which closely parallels bin-locking. An Android application is packaged and signed with a private key created by the developer. As with bin-locking, there is no requirement for a public key infrastructure. Application updates are allowed if the same private key was used to sign both versions. While bin-locking signs individual binaries, Android signs application packages. Each application is copied into its own separate directory during install by the Android OS. The OS keeps track of application signatures and prevents applications from overwriting files outside their install directory. The Android approach protects all types of files, not just application binaries. Backwards compatibility requirements preclude bin-locking from assuming that all data associated with an application is installed into the same directory (e.g., configuration files are commonly all stored in `/etc` and binaries stored in `/usr/bin` on Linux [39]). The Android signing approach is a customized solution for the platform because of the constraints it puts on how and where applications are installed. Bin-locking comes as close as possible to a general solution while preserving backward compatibility with current file-system layouts.

5.2 Rootkit-Resistant Disks

Rootkit-resistant disks by Butler et al. [8] relies on the user inserting a hardware token every time an area of the disk protected by that token is updated. New changes written to disk with the token inserted are marked as requiring the presence of that token in order to be modified. While rootkit-resistant disks protect a much larger range of file types, a knowledgeable user is required to insert the hardware token whenever any write operation is performed to the protected files (including updates). To protect every application separately, a different hardware token would need to be used for every application in-

Scheme	Proactive	Upgrades	File Types	Granularity
Bin-Locking (present paper)	Yes	Yes	Program Binaries	File
Google Android [1]	Yes	Yes	All	Application Package
Rootkit-Resistant Disks [8]	Yes	No	All	Hardware Token
Tripwire [21, 22]	No	No	All	File
Read-Only Media [23]	Yes	No	All	System

Table 1: Comparison of related file-system protection mechanisms. The granularity indicates to what extent the principle of least privilege is applied when modifying files.

stalled on the system. If only one token is used, then any application can modify any other application arbitrarily as long as the hardware token is inserted. A single-token system fails entirely if the user is ever tricked into running malware during the time the token inserted (including if the token is inserted after malware has started running).

5.3 Tripwire and Read-Only Media

Tripwire [21, 22] records cryptographic checksums for all files on a system to detect what files are changed by malware (by comparing against the current checksum). Read-only media [23] prevents any change from being made to the drive while the system is running, allowing the user to revert to a known-good state by simply rebooting the system. While Tripwire and read-only media differ from each other in their ability to prevent changes to the file-system, they share many characteristics, the most prominent being the way they deal with software installs and upgrades. With read-only media, the install or upgrade must be made on a system with writable media and then a new version of the read-only media is created – a potentially time-consuming process. With Tripwire, all changes to the file-system are flagged as potentially bad and the user must verify that each file modification is indeed acceptable (also a time consuming process). In both cases, security patches become troublesome to install. With Tripwire, the user has the option of verifying that an application does not overwrite core system binaries during install or upgrade – the same is not the case when updating read-only media. Tripwire does not prevent the modification of a file; it only detects these modifications.

5.4 Other Related Work

Related to work by Butler et al. [8], SVFS [53] also protects files on disk at the cost of running everything in a virtual machine. Software updates and installs are not addressed by SVFS. Strunk et al. [43] proposed logging all file modifications for a period of time to assist in the recovery after malware infection. Their approach does not prevent binaries from being modified in the first place. By combining their approach with bin-locking, logging can be restricted to configuration file changes – decreasing disk space requirements.

There have been many attempts at detecting modifications to binaries (in addition to Tripwire, discussed above). Windows file protection (WFP) [28, 10] maintains a database of specific files which are protected, along with signatures of them. The list of files protected by WFP is specified by Microsoft and focuses on core system files. WFP is designed to protect against a non-malicious end-user, preventing only accidental system modification. Pennington et al. [32] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. All these attempts rely on detecting modifications after the fact. While WFP is capable of handling updates, the other solutions do not appear to directly support binary updates.

Apvrille et al. [2] presented *DigSig*, an approach which also uses signed binaries in protecting the system. They modified the Linux kernel to prevent binaries with invalid signatures from being run (as opposed to the bin-locking approach of preventing the modification). Under Digsig, all binaries installed must be signed with the same key. While the use of a single key may work for corporate environments deploying DigSig, it does not seem well suited to decentralized environments. DigSig also relies on a knowledgeable user to verify all updates to binary files (similar to Tripwire) before signing them with the central key.

The approach to bin-locking also differs from that of van Doorn et al. [46] (and indeed many other signed-executable systems such as [36] and [11]). In these systems, the installation (or running) of binaries is restricted by whether or not the application is signed with a trusted key. In contrast, bin-locking does not restrict the addition of new executa-

bles (those with new file names) onto the system and does not rely on any specific root signature key being used, or external notions of trusted keys; it does not rely on any centralized PKI.

With all the approaches described (with the exception of that by Butler et al. [8] when using multiple tokens), it seems one common pitfall is that any application performing an update or install will have permissions sufficient to modify any other binary on the system. Some proposed systems attempt to mitigate this threat by assuming a vigilant and knowledgeable user will verify all changes to binaries. They rely on this user to never be tricked into installing a Trojan application. We believe that by differentiating between files originating from different developers or organizations, bin-locking can rely less on vigilant and knowledgeable users to protect some parts of the system. All approaches except bin-locking treat upgrades the same as new application installs.

While policy systems such as SELinux [27, 20] have the capability to restrict configuration abilities, the overhead of correctly configuring a policy for every application (including every installer) makes this approach unrealistic in many environments. Bin-locking allows binaries to be protected based on who *developed* (or *created*) them, a property not easily translated into frameworks such as SELinux. While projects such as DTE-enhanced UNIX [47] and XENIX [44] restrict the privileges of root (reducing the risk of system binaries being overwritten), installers (and even upgrades) are still given full access to all binaries on disk.

The OpenBSD `schg` [24] and ext2 immutable [45] flags are similar to bin-locking in that they prevent files from being changed, moved, or deleted. These flags, however, do not allow an application binary to be updated, resulting in a system more akin to read-only media (see §5.3).

Code-signing involves verifying the author (code source) before software is run [38]. Code-signing approaches generally do not restrict what the software can do while running. When applied to installers, code-signing allows a user to verify the source of the software they are about to install (and that the software has not been modified since the vendor signed it) – the same is true for package managers [9]. In common usage, in both cases the signature applies to the entire package (not to the individual binaries) and does not end up restricting which binaries either the installer application or installed program can modify. While some systems may maintain a cryptographic hash for files installed, these hashes are more akin to those used by Tripwire (see §5.3). Hashes alone are insufficient for tying two versions of a binary to the same source. While the bin-locking approach can prevent binaries modified during distribution from being installed as an upgrade, we don't focus specifically on this problem as do Bellissimo et al. [5].

Related to the approach herein of locking down the kernel interfaces which can be abused by malware to infect the kernel, several proposals have been designed to detect and prevent kernel exploitation as discussed in §4.1 [3, 4, 33, 26, 34].

6 Summary

When binaries are being installed, the current (almost universal) situation is that the installer has write access to essentially the entire file-system – far too coarse a granularity. To address this, we presented a bin-locking (i.e., digital signing) system including kernel support capable of protecting binaries on disk against modification by unauthorized software. We also discussed a prototype implementation, its strengths, weaknesses, and issues encountered and addressed during implementation. While bin-locking is not designed to protect all files or address all malware-related problems (indeed, a single solution to all such problems is unlikely to ever be found), we believe the beachhead created by bin-locking may serve as an important general technique, and one step in restricting the abilities of malware. One of the key features not widely addressed by previous file protection schemes (to our knowledge) is the ability to transparently handle software application upgrades. With many applications now receiving regular patches, dealing with upgrades in a smooth and non-intrusive manner is important. The proposed system enforces a separation between binary files belonging to different applications; even with privileges sufficient to install an application, binary files belonging to one application cannot be modified by an application originating from a different source. Bin-locking provides a start at addressing the problem of privileges given to installers, one which under common circumstances (e.g., auto-update of software), provides protection from malware with no additional burden on the end-user, through a software-only mechanism. While we do not restrict the ability to bin-lock binaries to certain vendors, we suspect that the vendors most interested in the capabilities offered by bin-locking may be those who develop or

provide system monitoring utilities and crucial services.

The prototype consists of a modified Linux kernel which restricts updates to designated binaries. It also restricts writes to raw disk sectors, drive mounting/unmounting, and write access to kernel memory by user space processes. It includes a utility which can bin-lock binaries, inserting both the digital signature and public verification keys. Our test system included bin-locking every binary in the Debian archive, creating and installing the new packages (resulting in every application binary on the system being bin-locked). The system was used for everyday activities and evaluated for performance and security against current Linux rootkit malware. It successfully prevented installation of current rootkit malware while having an imperceptible overhead to the end-user.

We offer several key insights into the configuration of a typical computer system. We advocate creating a stronger distinction between the kernel and root privileged processes and discuss how bin-locking allows a safe-subset of configuration operations to be performed while still limiting the ability to configure a system.

Acknowledgements. We thank many individuals who provided feedback on preliminary drafts of the paper. The second author acknowledges NSERC for an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security. Partial funding from NSERC ISSNet is also acknowledged.

References

[1] Google Android. Web site (viewed 28 Aug 2009). `http://code.google.com/android/`.

[2] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. Digsig: Run-time authentication of binaries at kernel level. In *Proc. LISA '04: Eighteenth Systems Administration Conference*, pages 59–66, 2004.

[3] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Rutgers University Department of Computer Science, January 2006.

[4] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. 2007 IEEE Symposium on Security and Privacy*, pages 246–251, May 2007.

[5] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *Proc. USENIX 2006 Workshop on Hot Topics in Security (HotSec)*, Jul 2006.

[6] M. Bishop. *Computer Security: Art and Science*, pages 381–388. Addison-Wesly, 2003.

[7] bsign. Web site (viewed 22 Jan 2009). `http://packages.debian.org/lenny/bsign`.

[8] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 403–415, Oct 2008.

[9] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 565–574, Oct 2008.

[10] J. Collake. Hacking Windows file protection. Web Page, May 2007. `http://www.bitsum.com/aboutwfp.asp`.

[11] G. Davida, Y. Desmedt, and B. Matt. Defending systems against viruses through cryptographic authentication. In *Proc. 1989 Symposium on Security and Privacy*, pages 312–318, May 1989.

[12] *The Debian GNU/Linux FAQ: Chapter 8 - The Debian Package Management Tools*, Jun 2008. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[13] D. Dittrich. "root kits" and hiding files/directories/processes after a break-in. Web Page, Jan 2002. `http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq`.

[14] *Executable and Linkable Format (ELF)*, version 1.1 edition. `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

[15] C. Ellison. *RFC 2692: SPKI Requirements*, Sep 1999. `http://www.isi.edu/in-notes/rfc2692.txt`.

[16] "Digital Signature Standard", Federal Information Processing Standards Publication 186. Technical report, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Virginia, 1994.

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

[18] Google. Andorid developer guide. Developer Website, Jul 2009. `http://developer.android.com/guide/publishing/app-signing.html`.

[19] J. B. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, May 2006.

[20] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proc. 12th USENIX Security Symposium*, pages 59–74, Aug 2003.

[21] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-93-071, Purdue University, 1993.

[22] 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.

[23] Knoppix Linux. Web Page (accessed 15 Dec 2008). `http://www.knoppix.net`.

[24] Y. Korff, P. Hope, and B. Potter. *Mastering FreeBSD and OpenBSD Security*, chapter 2.1.2. O'Reilly, 2005.

[25] G. Kroah-Hartman. Signed kernel modules. *Linux Journal*, 117:48–53, January 2004.

[26] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[27] P. Loscocco and S. Smalley. Integrating flexible support

for security policies into the Linux operating system. In *Proc. FREENIX '01*, Jun 2001.

[28] Microsoft. Description of the Windows file protection feature. Web Page, May 2007. `http://support.microsoft.com/kb/222193`.

[29] Microsoft. *Digital Signatures for Kernel Modules on Systems Running Windows Vista*, Jul 2007. `http://www.microsoft.com/whdc/winlogo/drvsign/kmsigning.mspx`.

[30] Writefileex function. Web Page, Nov 2008. `http://msdn.microsoft.com/en-us/library/aa365748(VS.85).aspx`.

[31] Y. K. Okuji. GNU GRUB. Web Page, Dec 2008. `http://www.gnu.org/software/grub/`.

[32] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proc. 12th USENIX Security Symposium*, pages 137–151, August 2003.

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

[34] N. L. Petroni Jr., T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symposium*, pages 289–304, August 2006.

[35] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing*, pages 215–219. Prentice Hall, 4th edition, 2007.

[36] M. Pozzo and T. Gray. An approach to containing computer viruses. *Comuters and Security*, 6(4):321–331, Aug 1987.

[37] R. L. Rivest and B. Lampso. A simple distributed security infrastructure, Oct 1996. `http://people.csail.mit.edu/rivest/sdsi11.html`.

[38] A. D. Rubin and D. E. Geer Jr. Mobile code security. *IEEE Internet Computing*, 2(6):30–34, Nov 1998.

[39] R. Russell, D. Quinlan, and C. Yeoh. *Filesystem Hierarchy Standard*. Filesystem Hierarchy Standard Group, 2.3 edition, Jan 2004. `http://www.pathname.com/fhs/`.

[40] J. Rutkowska. Subverting Vista kernel for fun and profit. Blackhat Presentation, August 2006. `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`.

[41] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, 2004.

[42] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, 2001.

[43] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, Oct 2000.

[44] Trusted XENIX version 3.0 final evaluation report. Technical Report CSC-EPL-92-001, National Computer Security Center, Apr 1992.

[45] C. Tyler. *Fedora Linux*, chapter 8.4. O'Reilly, 2007.

[46] L. van Doorn, G. Ballintign, and W. A. Arbaugh. Signed executables for Linux. Technical Report CS-TR-4259, University of Maryland, 2001.

[47] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX Security Symposium*, Jul 1996.

[48] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 368–377, June 2005.

[49] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symposium*, pages 17–31, Aug 2002.

[50] G. Wurster and P. van Oorschot. Self-signed executables: Restricting replacement of program binaries by malware. In *Proc. USENIX 2007 Workshop on Hot Topics in Security (HotSec)*, Aug 2007.

[51] G. Wurster and P. C. van Oorschot. System configuration as a privilege. In *USENIX 2009 Workshop on Hot Topics in Security (HotSec)*, Aug 2009.

[52] Z. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security*, 8 (2):153–186, May 2005.

[53] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *Proc. Third IEEE International Security in Storage Workshop (SISW'05)*, pages 21–28, 2005.