Reducing Unauthorized Modification of Digital Objects

Paul C. van Oorschot, Glenn Wurster

Abstract—We consider the problem of malicious modification of digital objects. We present a protection mechanism designed to protect against unauthorized replacement or modification of digital objects while still allowing authorized updates transparently. We use digital signatures without requiring any centralized public key infrastructure. To explore the viability of our proposal, we apply the approach to file-system binaries, implementing a prototype in Linux which protects operating system and application binaries on disk. To test the prototype and related kernel modifications, we show that it protects against various rootkits currently available while incurring minimal overhead costs. The general approach can be used to restrict updates to general digital objects.

Index Terms—protection mechanisms, software release management and delivery, system integration and implementation, access controls, file organization, operating systems.

1 Introduction and Overview

In current computing environments, most users are not security experts. Most computer users are instead tasked with using a computer in order to get their job done. Such a scenario results in an environment where security is often a tertiary goal [2]. In spite of this, many access control systems, when deployed on end-user desktops, place users in charge of granting permission to update files [3], [4]. By putting the user in charge, the implicit assumption is made that the user is capable of managing permissions in a way that will not compromise the security of the system. This assumption is dangerous, as it can lead to situations in which the user makes (or allows) modifications detrimental to system security. The assumption is also dangerous because passwords remain the dominant mechanism for authorizing software updates, and security and usability issues related to passwords are well-known [5].

In this paper, we re-examine the problem of how to authorize the modification of digital objects. Instead of relying on the user to properly control updates to a digital object, we focus on allowing the creator of the object to limit modifications to the object, through a well-planned use of digital signatures and verification public keys. Our approach does not rely on authenticating the end-user of the system on which the object to be updated resides, focusing instead on verifying that the creator of the updated object is authorized by the creator of the original object. We focus exclusively on the action of replacing a digital object with a new version of that object (i.e., whether object A_{k+i} is allowed to replace object A_k). We do not require a knowledge of *who* created the updated digital object, but instead ask *was this individual authorized to create an updated version of this object?*

The approach we take is to associate with each digital object a digital signature of the object. This signature is checked by the enforcement mechanism when performing an update to the object. In essence, the object is self-signed; no centralized (or other) public key infrastructure is involved. The core technology is a simple variation of self-signed executables [1]. We use the term *key-locking* to refer to our proposal, to avoid confusion with other schemes designed to limit the installation of digital objects based on the identity of the signing party. The proposed system allows objects to be easily upgraded, under the control of the (trusted) enforcement mechanism.

Section 2 first outlines the generic key-locking approach, along with related benefits and drawbacks. We then present several scenarios in which key-locking is beneficial, and in Section 3 expand on one of these approaches – restricting the modification of application binaries on a modern desktop. Section 4 discusses a prototype implementation of this application of key-locking, discussing its performance and ability to protect against known rootkit-related attacks in a Linux desktop. Section 5 discusses similar proposals designed to limit binary modification and related work. We summarize in Section 6.

2 Generic Proposal: Restricting Updates by Key-Locking

At the core of our proposal is a simple but carefully supported use of digital signatures, designed to protect a digital object against unauthorized modifications. By *digital object*, we mean digital content which is available

- *Carleton Computer Security Lab, School of Computer Science, Carleton University, Canada*
- *Version: August 30, 2010. This paper expands on a preliminary short paper [1]. Contact author: gwurster@scs.carleton.ca*

as a single file identified by a file name. Examples include: program binaries (including both executables and libraries), images, videos, application install packages, and compressed archives. Each digital object protected by key-locking contains both $m \geq 1$ digital signatures and $n \geq 1$ verification public keys. Both m and n are determined by the object author. To restrict who can modify (or replace) an object, one simple protection rule is enforced: *An old key-locked digital object may be replaced by a new one iff at least some number $k \geq 1$ (set in the enforcement policy) of unique digital signatures in the new object validate correctly using public keys in the old object*. This basic idea, as applied to binaries, was first presented in a short paper [1]. To be useful for replacing old objects on a system using a given k, it is necessary that $m \geq k$. In general, m and n can be chosen independently, m_{old} does not constrain m_{new}, and n_{old} does not constrain n_{new}[1] (e.g., possibly $(m_{\text{old}}, n_{\text{old}}, m_{\text{new}}, n_{\text{new}}) = (3, 2, 2, 5)$ or $(3, 4, 5, 1)$).

We suggest supporting a few standardized digital signature algorithms, where for each signature in the object, the chosen scheme is specified alongside and protected by the digital signature (i.e., different types of signatures are supported within the object, allowing for a transition away from a particular signature scheme should it become undesirable).

For currently installed digital objects containing a signature (in the "key-locking section"), an enforcement mechanism (e.g., the kernel in our prototype implementation) prevents replacement unless the new object contains k digital signatures (among its set of m_{new} signatures) which can be verified using k of the n_{old} keys in the old version of the object. Otherwise, the original object remains unmodified. Deployment is incremental – digital objects not key-locked can be replaced without restriction (which is how most systems currently operate). Once an object is key-locked however, it must be replaced by an object with at least k verifiable digital signatures. We discuss the choice of k in §2.2.

Each digital signature is computed over the contents of the entire digital object, with any area reserved for storing a digital signature zeroed out prior to computing the digital signature. The areas of the digital object used for storing verification public keys, as well as the key prefixes (as explained later) used in locating the public key when verifying the signature, are within the scope of the digital signature. If any of the public keys are modified, the digital object will fail to validate unless it is re-signed. Likewise, if any digital signature metadata (e.g., length, any key prefix, or count of digital signatures) is modified. No public key or signature can be replaced (or inserted/deleted) in a multiply-signed key-locked object without regenerating all signatures. This approach of signing everything (except the digital signatures) is necessary unless $k = n_{\text{new}}$ and public keys are tied directly to signatures (see §5.1). Otherwise, if a key-locked object could be signed with additional signatures while still being legitimate for those keys already embedded, an attacker could take control of the object by appending new verification public keys to an already key-locked object.

The process of signing a digital object for key-locking involves three steps, performed by an application designed to key-lock objects: 1) Embed verification public keys into the object – the creator of the object is responsible for choosing which public keys to embed and passing them to the key-locking application. 2) For each digital signature the object creator chooses to embed, the key-locking application creates a corresponding record in the digital object, filling in all fields except for the actual signature. A corresponding key prefix record is also created, which contains a prefix of the public key which can be used to verify the signature contained in the record. The key prefix record is used during signature verification to quickly find related public keys. 3) For each digital signature, the key-locking application uses the corresponding private key to sign the digital object. Although the object does not have to be signed with all private keys at the same time (i.e., the object can be passed around between private key holders involved in the signing process), the set of public keys (step 1) and signatures (step 2) to embed into the digital object must be agreed upon before any signatures for the object can be created.

Which software (or other) component enforces key-locking varies depending on the type of digital object being protected (see §2.7). For our prototype, which protected application binaries, enforcement was built into the Linux kernel. For application packages, enforcement should be built into the package manager. For files hosted on a web-site, enforcement would be built into the web server software.

2.1 Key Evolution

Over time, inevitably, signing keys used for key-locking will be lost, compromised, or become outdated. These situations can be ameliorated by using multiple verification keys in an object, and setting k such that $1 < k < n_{\text{old}}$. If one key is lost, the other key(s) can be used to sign a subsequent version of the object (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 backups. While we mandate no specific infrastructure for key revocation, pro-actively installing a new version of a binary 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 an outdated key from being used indefinitely, "revoking" the key. At somewhat greater key management expense, each object can be signed with a different key to limit the

1. m_{old} is the value of m for an old object. Similarly for n_{old}, etc.

effect of a compromised key. In the absence of versioning (§2.6.1), we suggest replacing verification public keys in the object with every major release to protect against downgrade attacks. Objects must continue to be signed with old keys in order to provide a smooth upgrade path from any previous version, but we do not suggest embedding in an object public keys corresponding to outdated signing keys (i.e., m may grow over time as verification public keys are replaced).

2.2 Using *k* of *n* public keys for signature verification

Each digital object being protected by key-locking will contain $n \geq 1$ distinct verification public keys. In determining whether a new version of the digital object can replace the current one, at least k of the n_{old} public keys in the current object must result in successfully verified digital signatures contained in the new object.

Parameter k is set by policy, being chosen during creation and/or configuring of the enforcement mechanism and enforced during every digital object replacement check. We do not mandate any specific choice of k, but do note some of the trade-offs:

1) The smaller k is, the greater the risk that k parties holding signing keys for a digital object may "go rogue." For example, if $k = 1$ then any single party holding a private signing key can distribute an updated version of the object without cooperation of any other original party, i.e., containing (or excluding) any verification public keys they wish. Choosing a very small k also increases the impact of key compromise, as an attacker needs only k compromised keys to provide "authorized" updates.
2) If $k = n_{\text{old}}$ for a particular object, then the misplacement of even a single signing private key prevents future updates to that object (since the digital signature corresponding to the lost private signing key cannot be created). To accommodate for lost private signing keys, k should satisfy $k < n_{\text{old}}$.

The security policy may set k dynamically as a function of m and n (e.g., $k = \lceil 0.5 n_{\text{old}} \rceil$, $k = m_{\text{new}}$, or $k = n_{\text{old}}$ [6, §5.3.3]), or as a rule-determined fixed value (e.g., $k = 2$ if $n_{\text{old}} > 2$ else $k = 1$).

2.3 Trust Model Assumptions (Generic)

In using key-locking, it is assumed that an attacker does not have access to the private signing keys, nor have control of the key-locking enforcement mechanism. Private signing keys need not be shared amongst parties involved in the creation of key-locked digital objects, and it is assumed that a private signing key cannot be derived from observation of the verification public key. What must be protected against is an attacker obtaining a private signing key through a compromise of the machine where the private signing key is stored. We assume that old verification public keys are not included in newer objects, and new keys are introduced frequently if a downgrade attack (i.e., replacing a more recent digital object with an older version) is a concern.

2.4 Beneficial Characteristics

Key-locking has the following beneficial properties.

1. No Central Key Repository or Infrastructure. The proposed system differs from many other code-signing systems in that it does not attempt to tie the signature to an entity. It can verify that the new version of a digital object is associated with the same author (or organization) as the old version without knowing that party's identity. Because the signature on a to-be-installed object file is verified using the public key embedded in the previous version of the object, 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.* Any new author can create a signing key-pair and begin using it immediately; creation of new digital objects remains unrestricted. We make no effort to restrict what objects can be created. If desired, different keys can be used for each object created to limit 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-object cost for replacing protected objects). Other object signing schemes have relied on a trusted central authority [7], [8].

2. Incremental Deployability with Incremental Benefit. Systems which do not yet support the key-locking mechanism see key-locked digital objects as if they were normal, being unaffected by the existence of the extra data fields associated with key-locking. Similarly, digital objects not key-locked may be allowed on a system which supports key-locking (in contrast to many proposed code signing schemes [8]). Key-locking can be first enabled in either the enforcement mechanism or digital objects without an adverse affect on non-supporting systems or software.

3. Low Overhead. As discussed in §4.8, key-locking can be deployed with imperceptible overhead to the end-user.

4. Simplicity. The key-locking concept is relatively simple to understand, and does not require any additional hardware or co-processors.

2.5 Limitations

We now discuss some limitations related to key-locking.

1. Deletion of Digital Objects. Because the digital objects themselves contain the list of public keys used to authorize replacement by a new version of the object, the deletion of a key-locked object also removes any requirements of specific digital signatures being present in any future version of the object installed onto that system, thereby allowing arbitrary replacement. If key-locked digital objects could be arbitrarily deleted, a new version not containing any authorizing signatures could

be installed by first removing the old version of the digital object (since then there will be no previous version to extract verification public keys from when the new digital object is installed). The deletion of key-locked digital objects is therefore disallowed, except by superusers. In the case where the name of the digital object is to be protected (e.g., for the bin-locking application in §3), the renaming of key-locked objects must also be restricted.

2. Denial of Service. Any attempt to replace a key-locked object with another key-locked object for which the signatures do not verify is denied. This creates a race condition in claiming object file names, with "name-squatting" applications potentially precluding the installation of legitimate digital objects through the creation of large numbers of dummy key-locked objects (much as domain name squatters tie up domain names). Depending on the environment in which key-locking is deployed, it may be desirable to name-space restrict what key-locked objects can be created by which entities. The exact environment-specific restrictions are beyond the scope of this paper.

2.6 Extensions to Key-Locking

Assuming a basic capability of verifying that digital object updates are authorized, extended functionality may be worth considering. The extensions discussed here would require additional support from the key-locking enforcement mechanism.

2.6.1 Versioning

As one possible extension, version numbers can be embedded in both the old and new digital objects. If the enforcement mechanism is expanded to control replacement based on version number, the same set of signature key pairs can be used over an extended period of time without the risk of a downgrade attack. While object authors (or organizations) can achieve the same effect by "revoking" keys (as discussed in §2.1), versioning allows the author of the object to reduce the number of signatures necessary in any new version of the object while still ensuring that the object can replace many previous versions. Note that legitimate software rollback (the process of reverting to a previous version) may not be possible while key-locking enforcement is active on a system (since key-locking, as outlined in this paper, is designed to also prevent downgrade attacks).

2.6.2 Sub-Keying

In the core idea, any digital object which can be validated using k keys in an installed object can replace the installed object. To prevent one object from replacing another semantically unrelated object from the same organization, the organization can use a non-overlapping set of keys for each object. As an extension to basic key-locking, an organization could embed an index number into each object they sign. While new versions of the same object would have the same index number, different objects associated with the same organization would have different index numbers. If the enforcement mechanism dictates that the index number between the old and new object must match, an organization could use the same private signing key for all their digital objects, without allowing semantically unrelated objects to be switched on a system. As an example in our prototype implementation (§3 below), sub-keying could be used to prevent `rm` from replacing `ls` while allowing the two binaries to be signed with the same key.

2.7 Applications of Key-Locking

Examples of possible applications of key-locking include (i) protecting against arbitrary modification of application binaries (as discussed in §3); (ii) restricting package updates (as discussed in §5.1); and (iii) for authorized updates, in place of procedural controls that typically accompany the use of usernames and passwords when pushing new versions of objects to a central server hosting digital objects created by users. A typical course of action in developing a website hosting content provided by multiple parties is to implement an authentication scheme, use it to authenticate valid users of the website, and then restrict the digital objects that can be modified by the authenticated user to some subset of objects on the site. Using key-locking, the initial version of a digital object identified by a URI is provided by a user and uploaded to the site through some alternate method (as specified by the system administrator and not discussed in this paper). Subsequent versions can be uploaded directly by the user pursuant to the constraint that the key-locking verification procedure passes. This provides a type of digital rights management functionality, in the form of exclusive control of updates. Such an approach has several advantages:

1) No state need be maintained between the authentication of a user and the uploading of the corresponding content by the user. Session hijacking attacks are therefore eliminated. The verification of authorization is performed through the verification of key-locking signatures embedded in the uploaded object.
2) Because the cryptographic verification of the digital signature is performed on the server, the digital content is protected against undetected and unauthorized modification while in transit (even without SSL if privacy is not required).

Such an approach to uploading digital content could be useful for servers hosting software repositories (e.g., for open-source package repositories).

3 Exemplar of Key-Locking: Bin-Locking

We first review how software installation is performed in most current computing environments. Applications

created by many different authors all coexist on disk, being installed at various times by the user. Each 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 do so is common. Indeed, to clearly state the problem: *Any application installer (or even application) running with sufficient privileges can modify any other application on disk.* Application installers are routinely given these privileges during software upgrade or install (e.g., almost all installers run as administrator or root, with 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 abuse this privilege to modify the binaries belonging to other applications. Malware, however, exploits the ability to modify other binaries as a convenient installation vector. Already in 1986, the Virdem virus [9] was infecting executables in order to spread itself; more recently, rootkits [10] have used binary modification in an attempt to hide.

The above motivates our application of key-locking to protect against arbitrary modification of application binaries, and the design of a surrounding architecture required to provide enforcement. We use the term *bin-locking*, to distinguish this application of key-locking from the general approach. We design, implement, and test a prototype for bin-locking in Linux. The design itself is generic – we see no reason why it could not be implemented on Windows or MacOS.

Bin-locking is not intended for use in all environments, for example, it does not co-exist well with some developer features for reasons discussed in §4.4. 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.

Bin-locking reduces the ability of malware 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. Bin-locking does not prevent all malware from being installed or run on a computer, but as a first step, prevents the modification of designated binary files – files created by developers and rarely (if ever) modified by the user. While configuration files and scripts remain unprotected by bin-locking, applying key-locking at the application package level (as is done in Android [11] – see §5.1), provides protection of such files.

3.1 Trust Model Assumptions (Bin-Locking)

Our prototype relies on several trust assumptions beyond those of the generic key-locking approach (§2.3). We implemented bin-locking enforcement in the kernel, and assume the attacker does not have kernel level control of the system.[2] On current systems, any application running with root (or administrator) access can modify the running kernel. Such an application can also bypass 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 discuss (in §3.4) and implement (in §4.2) several restrictions which lock down the interface between a running kernel and root. We note that protecting the kernel in this way is beneficial, independent of bin-locking. Malware exploitation of the kernel is a growing trend [12] and others in the security community have numerous proposals to protect the kernel [13], [14], [15], [16], [17], [18].

3.2 Additional Benefits for Key-Locking of Binaries

Bin-locking has benefits in addition to those discussed in §2.4. Previous proposals which attempt to limit changes to binaries on disk all apparently fall short for one of three reasons: they either detect changes a posteriori [19], [20], [7], [8] (making recovery hard), rely on the user to correctly validate every file modification, or still allow applications to modify arbitrary files during install/upgrade [8]. Deploying bin-locking addresses these three points with the additional benefit of helping to preserve trust in known parts of the software base even after infection by user-level malware.

The operating system and core applications are normally installed before malware has infected user-space. Bin-locking exploits this temporal property. Typical malware installed after the operating system (including core libraries and programs) cannot modify bin-locked operating system files. The integrity of the operating system files can therefore be trusted. 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 system infected by user-level malware can therefore be trusted, allowing partial control over an infected system even without requiring a reboot to clean media. This can also make recovery easier [21]. Using bin-locking, the integrity of anti-virus software and system binaries can be relied upon, 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 likely to result in earlier awareness of the malware.

3.3 Limitations for Key-Locking of Binaries

Bin-locking has two limitations beyond those in §2.5.

1. Kernel Interface Lock-down. In order for bin-locking to successfully resist attacks, several dangerous

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

kernel interfaces must be locked down (as discussed in §3.4). In the prototype, 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 (see §3.4). File-system repair utilities are only run on these partitions during boot. This was addressed 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 suggest that blocking the destructive power of file-system utilities during normal system operation is an acceptable (even beneficial) security practise. When file-system utilities must modify core partitions, a reboot into a kernel which does not enforce bin-locking can enable access.

During development and use of the prototype, 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 tested 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 [22], [23], [24].

2. Aliases. The bin-locking mechanism only protects binaries on disk. If malware can prevent the user-intended 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, this protection is of limited use if the authentic binaries are bypassed. We must ensure therefore on an infected system that the legitimate binary can be easily run instead of a binary at a location of the attacker's choice. Additional binaries of the same file name installed by malware can be avoided by running applications from the trusted base directly (e.g., during forensic analysis), avoiding the environment. Methods for accomplishing this include 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 bin-locking, some aliasing vulnerabilities can be addressed by controls in the binaries themselves (e.g., by the shell restricting `PATH` to include only designated core system directories when running as root). While it may be possible to restrict updates to symbolic links based on signatures in the objects they reference, this approach was not explored in the prototype.

3.4 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 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 prototype details more in §4.2.5.

To ensure that the bin-locking 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 any restrictions associated with files on that drive. We disable raw disk access in a way similar to mounting as discussed above. The modified kernel accepts a list of devices to which writes should not be allowed. Specifying the partitions containing the core system libraries forces any file updates to be processed through the bin-locking system. We discuss the implementation details of this in §4.2.4.

If deleting application binaries were still allowed, the bin-locking system would be rendered ineffective; an attacker could simply delete the binary and then install a new one having the same file name. To delete bin-locked files, 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 §4.7).[3] Previous work on providing a trusted interface (e.g., see [25]) – 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 by Butler et al. [26]) – as long as the appropriate hardware token is inserted, the deletion of bin-locked objects would be allowed.

Disabling bin-locking by modifying the running kernel, and how to prevent this, are discussed in §4.2.3.

4 Bin-Locking Prototype Implementation and Evaluation

To verify the viability of the bin-locking exemplar of key-locking, we modified a system to implement bin-locking,

3. Because all application installs are performed as root in Debian, restricting the deletion of bin-locked binaries to the root userid is not sufficient for protecting the system.

including the kernel interface restrictions. The prototype implementation is composed of a number of different pieces which work together to protect the system. We did not implement the §2.6 extensions to key-locking. 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 Linux boot process was modified on the test system to limit raw writes and mounting (see §3.4). We used a constant value $k = 1$ (m and n both varied between 1 and 3). We now discuss each modified element of the prototype Debian system in detail.

4.1 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 purpose of storing bin-locking related data. ELF was designed such that applications could create new sections and many other applications (e.g., GCC and bsign [27]) take advantage of this flexibility.

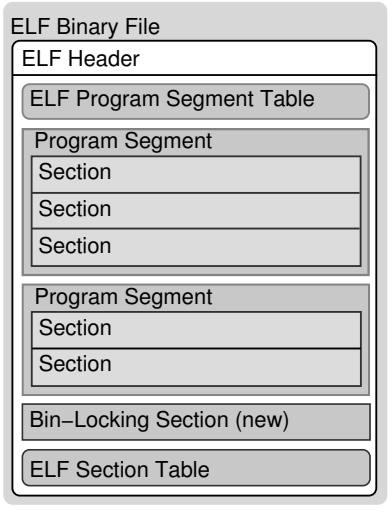

Fig. 1. Basic ELF Layout including Bin-Locking section.

The bin-locking section of the ELF file consists 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 public key corresponding to the private key used to generate the 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 key related to the signature and is used for quickly determining what verification key in a previous version of the binary should be used for verifying the signature (if multiple keys share the first four bytes, the kernel will attempt to verify with each). Figure 2 illustrates the layout.

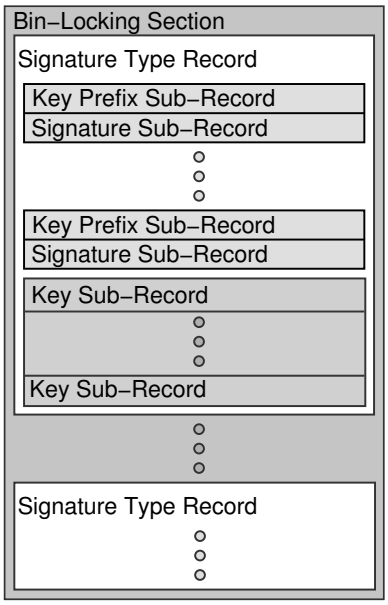

Fig. 2. Bin-Locking File Section Layout

To allow for future signature schemes, we included several 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 modified kernel to skip over unrecognized signature types. The sub-record header, in addition, contains a flag which notifies the modified kernel that the record contents should be zeroed before hashing the file as part of the signature creation or verification – signatures are stored in sub-records with this flag set. We discuss the kernel verification of digital signatures more in §4.2.2. The layout of a sub-record is illustrated in Figure 3.

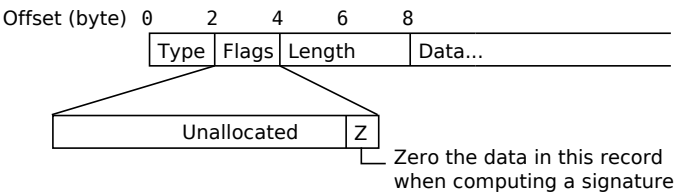

Fig. 3. Layout of a sub-record in the bin-locking section (records are similar).

4.2 Kernel Modifications

The kernel was modified to enforce bin-locking as discussed in §3. We used the MPI library ported to the Linux kernel for cryptographic primitives [28]. Signed binary files could no longer be deleted, moved, or opened for writing. They could only be replaced with new binary files which contained a signature verifiable using a key in the corresponding old binary. On a replacement request (which is initiated through a `move` system call involving a bin-locked binary), the kernel attempts to extract and use the keys from the old binary to verify the authorization 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. Figure 4 summarizes the sequence of decisions the modified kernel makes in replacing a bin-locked binary through the `move` call. Figure 4 illustrates one technique by which a kernel can update a bin-locked file. The kernel retains backwards compatibility with binary files that are not bin-locked, not restricting their replacement or removal.

Note 1 to Figure 4: recall from §2.5 that renaming bin-locked files would allow arbitrary replacement. Renaming is therefore not directly allowed by the prototype kernel; hence the use of an 8-byte `BIN-LOCK` prefix. The `BIN-LOCK` prefix serves as a special marker to the kernel, to temporarily allow deletion and movement of the file prior to it replacing an existing bin-locked file, since normally bin-locked files may not be deleted or moved.

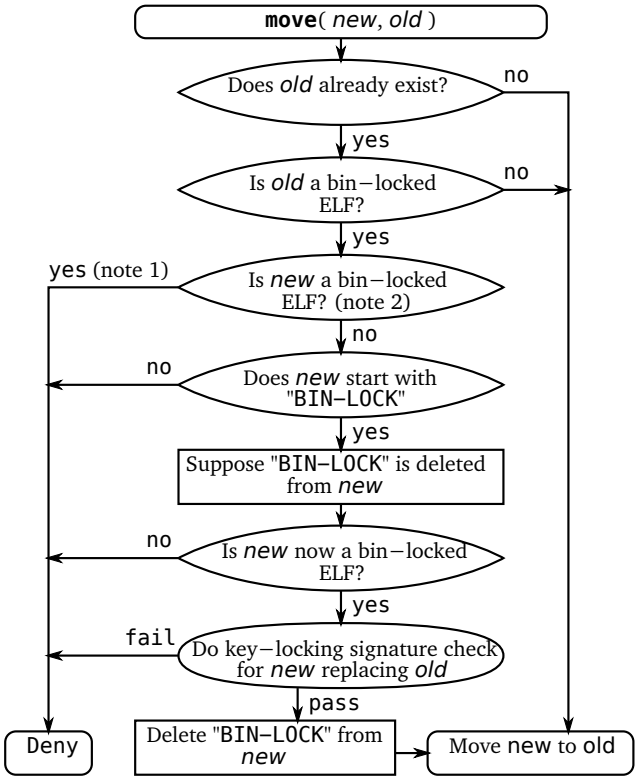

Fig. 4. Kernel flow chart for replacing bin-locked files. See text for explanation of note 1 and 2.

To simplify the implementation, we chose not to rely on a user mode helper (e.g., by using the `call_usermodehelper` function in the kernel) in designing the prototype system. While we believe a user mode 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 being taken over by malware.

4.2.1 Detecting Bin-Locked files

To detect whether or not a file was bin-locked (at note 2 in Figure 4), the modified kernel examined very specific file elements. It verified that the file was 1) an ELF file, 2) contained a bin-locking section, and 3) the bin-locking section contained a digital signature in a format known to the kernel (e.g., DSA). To determine if a file is bin-locked, the kernel 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 [29], 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 in §4.2 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 bin-locked and correctly signed. The overhead of enforcing bin-locking is discussed in §4.8.

4.2.2 Verifying Digital Signatures

To verify that a new binary is authorized to 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). With each verification key that matches the key prefix, the kernel attempts to verify the signature. If the signature check passes, the replacement is allowed (i.e., $k = 1$).

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.

4.2.3 Avenues for Kernel Modification

To protect the bin-locking system itself, the kernel was also modified to remove the following functionality which could otherwise be used to attack the system: (1) modifying the kernel to disable the protection scheme; (2) editing bin-locked binaries directly on disk; and (3) hiding bin-locked binaries (by either mounting or unmounting the partitions they reside on). To prevent (1),

/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 [30], [31]. /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). Red Hat has restricted access to /dev/kmem and /dev/mem for several years without problems [30], [31]. While we disabled module loading entirely, a better option is to deploy module-signing (as discussed by Kroah-Hartman [32]). While the above protections were sufficient for testing the prototype, we refer the reader to mechanisms and proposals by others for complete kernel protection (see §3.1)

To address (2) and (3), 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.

4.2.4 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, which resets the list to empty. 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 [24]). If any partition on a disk is designated as 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 prototype, 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 [33], 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.

4.2.5 Restricting Mounting

To prevent bin-locked files from becoming inaccessible, the prototype restricts 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 kernel 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. Unmounting may (or may not) be required 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.

4.3 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 (per the format of Figure 3). We chose not to use bsign [27], 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. The prototype signing application signs binaries using the Digital Signature Algorithm (DSA) [34], 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 [35] to not write out bin-locked binaries to temporary files during the installation of the prototype system (since the temporary bin-locked ELF file would be protected by the modified kernel). Instead, the package manager writes out the `BIN-LOCK` prefix followed by the signed file (see §4.2). The binary rewriting application was used along with several additional scripts to create a local Debian 4.0 mirror [36] where every application binary and library was bin-locked. Binaries created by install scripts through install-time compilation or optimization were not bin-locked in the system.

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, the modified kernel does not enforce bin-locking on drives not associated with a physical device.

4.4 Bin-Locking on Developer Systems

Developers are not the target audience of bin-locking. Here we discuss several developer features which currently remain enabled on all systems and undermine the security of bin-locking. We propose limiting 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) do not typically use these features.

1. UNIX `ptrace` hooks [37] 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. To ensure bin-locked binaries are run unmodified, `ptrace` access must be disabled for bin-locked binaries.

2. The use of custom builds by third parties is limited (by bin-locking) to those verifiable using public keys in the original binary. 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` on Linux allow additional libraries not specified in an executable to be linked in at run-time. Pre-loaders provide a method for modifying a binary at run-time. There are two alternatives for preventing this from being exploited by attackers. The first (and easiest) is to globally 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 executable 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 ship bin-locked binaries to end-users with the feature disabled.

4.5 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.[4] Linux rootkits can be grouped into two categories. The first 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 administrator in examining a system. Both classes of rootkits attempt 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 none of the six rootkits were able to install is supporting evidence of expected functionality of the bin-locking system.

4.6 Effect on User Tasks

As partial evidence that the modified kernel and signed executables are viable, all installed binaries and libraries were bin-locked and kernel interfaces were locked down (as discussed in §4.2.3) on a test system. The test system, a desktop install running KDE (the graphical based K Desktop Environment) was used to browse the web, write e-mail, listen to music, and view video – all with no noticed differences from an ordinary system. In using the test system over the course of two weeks, no application appeared to be broken by the enforcement of bin-locking, and we found no problems. From the nature of the modifications, we have no reason to anticipate that major problems would be found through more thorough testing. We did not encounter any scripts that attempted to remove or optimize pre-existing binaries as part of the package install process. While the home directory of the test system was mounted over the network and worked

4. All Linux rootkits tested were from http://packetstormsecurity.org/UNIX/penetration/rootkits/

without difficulty, we did not explicitly test bin-locking enforcement on network-based file-systems.

4.7 Reboots

Because the prototype requires a reboot to delete or move bin-locked files, the process of rebooting into a kernel which does not enforce bin-locking protections must be as user-friendly as possible. We used the standard GRUB [38] boot loader to provide an option to the user as to whether or not to use the bin-locking enabled kernel; the user must select one of the non-enforcing kernels from the menu during boot. Once booted into an alternate kernel, the user may delete and move any bin-locked file. An open problem is how to persuade users to normally use the kernel which enforces bin-locking. We discuss an option for removing the reboot requirement in §3.4.

4.8 Performance

While any performance impact of the proposed system was imperceptible to the end-user, for precise performance measurements we ran benchmark tests to quantify the overhead of bin-locking on a Pentium 4 at 2.8GHz with 1G of RAM. First, consider all non-locked files. Using the Perl benchmark library, we measured the average increase in kernel time required to perform a open, 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 (1.2K) file, 10000 with a medium (32K) one, and 5000 with a large (5.3M) file – the average increase in time to open for writing, and move/delete both non-ELF and unsigned ELF files is listed in Table 1. There was a 0.05% overhead in opening any file for reading.

	Operation	Average	Min	Max
non-ELF	Move/Delete	15.59%	13.83%	17.52%
		6.0μs	3.0μs	7.0μs
	Open for Write	34.84%	25.00%	38.24%
		3.9μs	2.6μs	4.3μs
unsigned ELF	Move/Delete	29.15%	24.36%	41.91%
		11.2μs	9.3μs	15.4μs
	Open for Write	66.65%	57.14%	77.78%
		7.7μs	6.7μs	9.4μs

TABLE 1
Overhead of file operations with bin-locking enabled.

While the percentage increases are high for opening a file, the absolute 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 bin-locked or not, this overhead could be brought down to essentially 0%.

Second, the cost of replacing a bin-locked file is increased by the time to perform a cryptographic hash (thus linear in its length) and one DSA verification. We emphasize that this cost is only occurred during the install or upgrade of a bin-locked binary, not while performing normal tasks (e.g., executing an application).

5 Comparison with Related Approaches

We first compare bin-locking with Google Android v2.0 [11], and then discuss work related specifically to bin-locking (in §5.2) and key-locking (in §5.3).

5.1 Google Android

In parallel to our work (but subsequent to publication of the preliminary design [1]), Google introduced a signing approach [39] in the Android platform [11] with similarities to key-locking. An application developed for Android v2.0 is packaged and signed with a private key created by the developer. As with key-locking, there is no requirement for a public key infrastructure. Application updates under Android are allowed if *all* verification public keys in the new version are also in the installed version of the package, and each verification public key in the new version can be used to verify a digital signature in the new version (i.e., $m = n$, $n_{\text{new}} \leq n_{\text{old}}$, k is set dynamically such that $k = m_{\text{new}}$, and the set of keys in the new package is a subset of keys in the old). The approach of checking all verification public keys as used in Android is necessary because of its use of `jarsigner` [40] to perform the signing operation – `jarsigner` does not include verification public keys in the data protected by a digital signature (recall from §2 paragraph 4 that k is constrained such that $k = n_{\text{new}}$ if public keys are not included in signed data). Another artifact of using `jarsigner` is that public keys cannot be added to a package without the corresponding digital signature of the package also being added. In contrast to key-locking, the Android scheme precludes new public keys being added during upgrade. While the bin-locking implementation of key-locking signs individual binaries, Android signs application packages. Each application must be installed as a package, and is assigned into its own separate directory by the Android OS. The OS keeps track of application signatures, preventing applications from overwriting files outside the assigned 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 [41]). The Android signing approach is a customized solution for that platform but not suitable in other environments because of the constraints it puts on how and where applications are installed. Bin-locking provides a more generic, configurable solution while preserving backward compatibility with current file-system layouts.

5.2 Bin-Locking Related Work

The rootkit-resistant disks proposal by Butler et al. [26] provides protection for designated disk areas by requiring that the user insert a hardware token every time the corresponding area of the disk protected by that token is to be updated. To protect every application separately, a different hardware token would be used for each application installed. In contrast, bin-locking protects each application separately, allowing binaries to be upgraded without the presence of a physical token.

Tripwire [42], [19] records cryptographic checksums for designated files on a system to detect what files are changed by malware (by comparing against the current checksum). Read-only media [43] 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. In both systems, security patches become troublesome to install (either a new read-only media needs to be created, or all changes need to be verified by the user). 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 detects any modifications afterwards.

Package managers such as `dpkg` allow signing the root file in the package repository [36], which results in all packages in the repository being protected against undetected modifications. The signature is checked before any package install or upgrade by the package manager using public key certificates stored on the client. While public key certificates can be updated by the install of a package, the approach differs from key-locking because a single signature protects the entire repository, where key-locking is potentially much finer-grained. Furthermore, `dpkg` ties public key certificates to entities (typically, the repository maintainers).

Related to the work of Butler et al. [26], SVFS [44] 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. [45] proposed logging all file modifications for a period of time to assist in recovery after malware infection. The approach does not prevent binaries from being modified in the first place. By combining this approach [45] with bin-locking, only modifications to configuration files need to be logged, since binaries can not be modified by user-level malware.

There have been many proposals for detecting modifications to binaries (in addition to Tripwire above). Windows file protection (WFP) [20], [46] maintains a database of specific files which are protected, along with signatures on 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. [47] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications; as with previous approaches [45], [19], their approach relies on detecting modifications after the fact. While WFP is capable of handling updates, the other solutions [26], [44], [47], [45], [19], [43] do not appear to directly support binary updates.

Apvrille et al. [7] presented *DigSig*, another approach using signed binaries in protecting the system. A modified Linux kernel prevents binaries with invalid signatures from being run (in contrast, bin-locking prevents 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 is less well suited to decentralized environments. DigSig relies on a knowledgeable user or administrator to verify all updates to binary files (similar to Tripwire) before signing them with the central key.

The signing approach in bin-locking also differs from that of van Doorn et al. [8] (and indeed many other signed-executable systems such as [48] and [49]). 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 executables (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, or on any centralized PKI.

In all the approaches described above (with the exception of that by Butler et al. [26] 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 assume this user will 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 of the above approaches except bin-locking treat upgrades the same as new application installs.

While policy systems such as SELinux [50], [51] have the capability to restrict system configuration actions, 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 [52] and XENIX [53] 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` [54] and ext2 immutable [55] 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

above).

Sparkle[5] allows developers to create signing keys, using them to sign their applications without relying on a central PKI. The approach authenticates updates to applications by ensuring all updates are signed with the same signing key as the initial version. While the approach is similar to key-locking, Sparkle enforces that only a single fixed signing key can be used per-application (key-locking allows both the embedding of multiple keys and key evolution).

5.3 Other Related Work

Digital Rights Management (DRM) [56] focuses on controlling the distribution and usage of digital assets. Key-locking, in contrast, is not concerned with the distribution of digital objects, but rather the updating of already-existing copies of those digital objects.

SDSI/SPKI [57], [58] simplifies management of public keys by treating the public keys as principals. Each principal can make statements, requests, or even act as a certification authority. In doing this, SDSI/SPKI removes the reliance on a global key repository, but still attempts to associate public keys with certificates (where the DN is a local name [59]). In key-locking, there is no attempt made to tie a verification public key to a digital certificate. Instead, keys are trusted only for the limited scope of replacing a digital object.

Code-signing involves verifying the author (code source) before software is run [60]. Such an approach involves authenticating the source, as well as determining whether authorization to perform the requested actions is given. While code-signing approaches can restrict what the software can do while running (e.g., in Symbian [61] and Blackberry [62]), tying a key to an entity results in an approach distinct from key-locking. 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 [36]. Some systems maintain a cryptographic hash for files installed, akin to those used by Tripwire (see §5.2); but hashes alone are insufficient for tying two versions of a binary to the same source. While the key-locking approach can prevent digital objects modified during distribution from being installed as an upgrade, we don't focus specifically on this problem as do Bellissimo et al. [63].

6 Summary

Key-locking allows fine-grained control over entities allowed to replace a particular object. Key-locking can be used to protect against arbitrary modification of application binaries (as discussed in §3), and restrict package updates (as discussed in §5.1). Key-locking can also be used instead of usernames and passwords when pushing new versions of binary objects to a central server hosting digital objects created by users (as discussed in §2.7).

Our application of key-locking to application binaries addresses a widespread problem: 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 from a security perspective. 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 prototype implementation validates the general approach and provides an important mechanism to help limit the abilities of malware. One aspect not widely addressed in the literature (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. Key-locking provides a mechanism to enforce 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.

The bin-locking exemplar of key-locking 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 create bin-locked binaries, inserting both the digital signature and verification public keys. Our prototype testing included key-locking every binary in the Debian archive, creating and installing the new packages (resulting in every application binary on the system being key-locked).

5. http://sparkle.andymatuschak.org

Acknowledgements

We thank the anonymous referees whose comments helped improve this work, and many individuals who provided feedback on preliminary drafts of this paper. The first author acknowledges NSERC for an NSERC Discovery Grant and his Canada Research Chair in Authentication and Computer Security. Partial funding from NSERC ISSNet is also acknowledged.

References

[1] 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)*.

[2] ——, "The developer is the enemy," in *Proc. 2008 Workshop on New Security Paradigms*, Sep 2008.

[3] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible support for multiple access control policies," *ACM Transactions on Database Systems*, vol. 26, no. 2, pp. 214–260, Jun 2001.

[4] S. Garfinkel, G. Spafford, and A. Schwartz, *Practical Unix & Internet Security*, 3rd ed. O'Reilly Media, Inc., Feb 2003, Chapter 5. Users, Groups, and the Superuser.

[5] D. Florêncio and C. Herley, "A large-scale study of web password habits," in *Proc. 16th International Conference on World Wide Web*, 2007, pp. 657–666.

[6] G. Wurster, "Security mechanisms and policy for mandatory access control in computer systems," Ph.D. dissertation, Carleton University, 2010.

[7] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy, "Digsig: Run-time authentication of binaries at kernel level," in *Proc. LISA '04: 18th Systems Administration Conference*, 2004, pp. 59–66.

[8] L. van Doorn, G. Ballintign, and W. A. Arbaugh, "Signed executables for Linux," Univ. of Maryland, Tech. Rep. CS-TR-4259, 2002.

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

[10] 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.

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

[12] 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*, May 2007, pp. 246–251.

[13] 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*, August 2004, pp. 179–194.

[14] 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)*, June 2005, pp. 368–377.

[15] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proc. 20th Annual Computer Security Applications Conference (ACSAC'04)*. IEEE Computer Society, 2004, pp. 91–100.

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

[17] 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*, August 2006, pp. 289–304.

[18] A. Baliga, X. Chen, and L. Iftode, "Paladin: Automated detection and containment of rootkit attacks," Rutgers Univ. Dpt. of Computer Science, Tech. Rep. DCS-TR-593, January 2006.

[19] 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*, 1994, pp. 18–29.

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

[21] J. B. Grizzard, "Towards self-healing systems: Re-establishing trust in compromised systems," Ph.D. dissertation, Georgia Institute of Technology, May 2006.

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

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

[24] 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.

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

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

[27] "bsign," Web site (viewed 22 Jan 2009), http://packages.debian.org/lenny/bsign.

[28] D. Howells, "Modsign: Kernel module signing," Patchset from Linux Kernel Mailing List (LKML: 14 Feb 2007), Feb 2007, http://lkml.org/lkml/2007/2/14/164.

[29] *Executable and Linkable Format (ELF)*, 1st ed., http://www.skyfree.org/linux/references/ELF_Format.pdf.

[30] A. van de Ven, "make /dev/kmem a config option," GIT Commit, Apr 2008, http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=b781ecb6a379f155568ef7093e38c6c1d857fe53.

[31] ——, "x86: Introduce /dev/mem restrictions with a config option," GIT Commit, Apr 2008, http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=ae531c26c5c2a28ca1b35a75b39b3b256850f2c8.

[32] G. Kroah-Hartman, "Signed kernel modules," *Linux Journal*, vol. 117, pp. 48–53, January 2004.

[33] 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*, Aug 2002, pp. 17–31.

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

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

[36] 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*, Oct 2008, pp. 565–574.

[37] P. Padala, "Playing with `ptrace`, part 1," *Linux Journal*, vol. 103, Nov 2002.

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

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

[40] S. Oaks, *Java Security*, 2nd ed. O'Reilly Media, Inc., May 2001, Chapter 12. Digital Signatures.

[41] R. Russell, D. Quinlan, and C. Yeoh, *Filesystem Hierarchy Standard*, 2nd ed., Filesystem Hierarchy Standard Group, Jan 2004, http://www.pathname.com/fhs/.

[42] G. H. Kim and E. H. Spafford, "Experiences with Tripwire: Using integrity checkers for intrusion detection," Purdue Univ., Tech. Rep. CSD-TR-93-071, 1993.

[43] "Knoppix Linux," Web Page (accessed 15 Dec 2008), http://www.knoppix.net.

[44] 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)*, 2005, pp. 21–28.

[45] 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.

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

[47] 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*, August 2003, pp. 137–151.

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

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

[50] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," in *Proc. FREENIX '01*, Jun 2001.

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

[52] 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.

[53] "Trusted XENIX version 3.0 final evaluation report," National Computer Security Center, Tech. Rep. CSC-EPL-92-001, Apr 1992.

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

[55] C. Tyler, *Fedora Linux*. O'Reilly, 2007, Chapter 8.4.

[56] Q. Liu, R. Safavi-Naini, and N. P. Sheppard, "Digital rights management for content distribution," in *Proc. Australasian Information Security Workshop Conference on ACSW Frontiers*, vol. 21, 2003, pp. 49–58.

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

[58] C. Ellison, *RFC 2692: SPKI Requirements*, Sep 1999, http://www.isi.edu/in-notes/rfc2692.txt.
[59] J. Y. Halpern and R. van der Meyden, "A logical reconstruction of SPKI," *Journal of Computer Security*, vol. 11, no. 4, pp. 581–613, 2003.
[60] A. D. Rubin and D. E. Geer Jr., "Mobile code security," *IEEE Internet Computing*, vol. 2, no. 6, pp. 30–34, Nov 1998.
[61] A. P. Heiner and N. Asokan, "Secure software installation in a mobile environment," in *Poster, 3rd Symposium on Usable Privacy and Security*, 2007, pp. 155–156.
[62] *BlackBerry Java Application: Fundamentals Guide (Version 5.0)*, Research in Motion Limited, Apr 2010.
[63] 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)*.