Protecting User Files by Reducing Application Access

ABSTRACT

Traditional discretionary access control mechanisms do not differentiate between a user's running applications–hence they provide no means of preventing one application from exploiting another's data. Commercial mandatory access control mechanisms such as SELinux and AppArmor aim to protect system files, but do little to prevent similar misuse of user data. This paper presents the PinUP access control overlay. PinUP extends filesystem protections to explicitly identify the set of applications that may access each user's sensitive files. This reflects users' intuition about access: that files should only be accessed by the applications that own them. This approach reduces the often esoteric task of access control policy specification to a significantly simpler declaration of the relationship between sensitive user files and applications. In so doing, we reduce the significant gap between existing access control and least privilege frequently exploited by malware such as viruses, worms, and spyware. We describe our model, architecture, and Linux implementation, evaluate run-time costs, and detail use-cases illustrating the power and utility of the augmented policy. Our performance experiments show that all costs are nominal, with a maximum observed delay of 40 milliseconds occurring at application startup and a few tens of microseconds at each access check. In this, we provide an efficient and intuitive means of pushing access controls provided to users ever closer to the ideal of least privilege.

1. INTRODUCTION

Files are the object of user risk. They can contain the sensitive artifacts of a user's job, finances, and personal life. Files are modified by diverse applications implementing complex tasks and workflows. Further, the access control mechanisms of current operating systems make no distinction between the vast array of applications that can modify user files. As a result, user applications can corrupt or leak user files with impunity. This leads to an unfortunate reality: *a user's files are only as protected as the least trustworthy application accomodates*. This lack of *least privilege* is frequently exploited by every kind of malware: Trojans, viruses, spyware, and worms can change user startup files and configurations, leak application caches, export user databases, and even modify files containing other user applications. Whether to steal information, violate privacy, install bots, or for other nefarious reasons, they all exploit the broad permissions given to applications on behalf of users.

Commercial operating systems now implement a form of mandatory access control (MAC) that addresses a closely related problem: how to protect *system* files in the presence of broad discretionary file system controls. Systems such as SELinux [?] and AppArmor [?] place tight controls on how processes can modify sensitive system objects. However, such solutions are not appropriate for users for at least two reasons. First, every user has a different set of applications and files related to their data. Hence, specifying policy for these artifacts using largely foreign concepts such as groups, roles, and attributes is difficult and probably impractical. Second, history has shown that users cannot and will not manage complex policy correctly. For this reason, even the simplest of existing system level policy specifications is beyond the grasp of the vast majority of users.

We hypothesize that even sophisticated users can only successfully create policy for their data when the specification process is: *a*) simple, *b*) uses language and concepts that native to their understanding, and *c*) the implications of policy decisions are simple and obvious. This work promotes and access control extension that attempts to meet these ambitious goals. In pursing this, we reflected on the nature of the user experience with commodity operating system. In essence, users create and manipulate each file with a single or series of well-known applications. Users necessarily must understand the nature of the relationship between sensitive files and the applications that use them. Importantly, user knowledge about these associations is authoritative (the user is often the only in a position make decisions sensitive data) and complete.

This introspection led to the following deceptively simple PinUP protection model: each user-specific policy specifies the set of applications that may access a user's sensitive files (known as *high-value* files throughout). For example, a user may indicate financial data (in a `*.qdf` file) be restricted to the Quicken accounting application. Of course, users cannot be reasonably expected to specify policy for every sensitive file. We extend the model to embrace PinUP policies that are automatically applied to files as they are created by applications. For example, it may be highly desirable to restrict the files `.xls` files created by Excel to only that application. These application-specific policies are largely uniform across users and environments, and thus can be provided to the system during application installation. Such a system meets our goals above: the policy is simple, reflects users' conceptual model of the system, and whose enforcement is observable and intuitive, i.e., respects the *principal of least astonishment* [].

Note that PinUP is not intended replace existing protection systems, but to augment them—any existing access controls will be enforced *in addition* to the user-specified PinUP policy. Architecturally, we overlay MAC controls with a secondary access control module that limits which applications can access which user files. Like stacking a second module in the Linux Security Modules framework [?], our access control module "overlays" MAC system access by consulting the additional user file controls only if the underlying file system protections permit the operation.

This paper introduces the PinUP[1] access control overlay system. We begin by considering the limitations of current system protections and more fully present the PinUP model. PinUP administrative operations and tools are detailed and our implementation is described and evaluated empirically. A performance analysis shows that application load times incurred in the worst case a 40 millisecond delay and all access check costs were nominal. We conclude by

[1]Our model conceptually "pins" files to specific applications, hence the name PinUP.

illustrating a number of motivating use cases and discuss the complexities and administrative overheads of implementing PinUP.

The remainder of this paper considers the philosophy, design, implementation of the PinUP system. This work, on some levels, parallels recent attempts at achieving system level least privilege. We refocus these efforts on the users themselves, beginning in the following section by considering the motivation and requirements such a model and system.

2. PROTECTION APPROACH

Traditional DAC systems authorize file access by checking whether the user identity associated with the requesting process has the necessary permissions. Because every process run by a user has the same identity, all processes operate with the same permissions–leaving all user data vulnerable to any single malicious process. For example, email client attachments can be downloaded and executed with access to any user data, enabling theft of financial information (e.g., Quicken files), the modification of important documents (e.g., corporate documents), and the modification of executables (especially if the user runs with administrative privileges enabled).

Emerging approaches, such as SELinux [?] and AppArmor [?], enforces MAC policy over file access. Unlike traditional discretionary systems, MAC systems associate permissions with process labels, typically defined by the application being run and not the user[2]. MAC is well suited to protect system files, but generally not for protecting user files. For example, the SELinux reference policy uses a label `user_t` to express user access policy. Not only does this approach fail to restrict access based on application identity; it does not even isolate one user from another.

Observe that three types of files exist in current systems: (1) system files; (2) high-value user files; and (3) other user files. We believe that different protection approaches are necessary for each. The integrity and secrecy of system files is of paramount importance, and is necessarily protected by MAC systems such as AppArmor or SELinux. Conversely, the huge body of non-sensitive user files is of little or lesser value to the adversary, thus traditional protections seem adequate. What remains–(2) the user's high-value files–calls for additional protections such as PinUP.

We now illustrate PinUP via the example in Figure 1. Assume that the user `jdoe` creates a file `x.c` using the the editor `vi`. The identity of the user, creating application, and other attributes such as file extension are used to identify the PinUP policy to be applied to the file, e.g., `jdoe`, `vi`, and `.c`. The policy explicitly states which applications will be allowed to subsequently access that file. In this case, `vi` is given subsequent read and write access. `x.c` is a source file, so the `jdoe` wants to make it read-only accessible to a compiler through a command line tool. Note that such a policy need not be manual–an alternate reasonable policy would dictate that all `.c` files created by `vi` would made available to the `gcc` compiler automatically. The compiler can then create a new object file `x.o` that it can write and that can also be read by the linker. Similar policy enforcement will occur as applications cascade over data to consume and create protected files.

Also illustrated in Figure 1, PinUP presents a number of other interesting design and implementation challenges. For example, how one securely identifies applications and propagates their identity across updates is key to ensuring correct policy enforcement. The issue of attaching, tracking, and propagating PinUP polices associated with user files is also daunting. This latter process is closely related to label management in mandatory access control systems. However, because of policy semantics and form, PinUP policy tracking requires different machinery.

The central challenge of PinUP is to develop a system that protects high-value user files[3] by limiting the applications that can access those files. The PinUP system consists of: (1) an authorization mechanism that can enforce such access and (2) administrative functions that enable evolution of such rights. This system ensures that when a user creates files from an authorized binary, subsequent accesses to this file will be limited to applications specified by a combination of system rules and authenticated user assignments. We begin this exploration in the next section.

3. SYSTEM DESIGN

The key design challenge is to enable effective administration of the access of authorized applications to high-value user files. Figure 1 presents different types of administrative operations that must be supported to enable application-level control of user files. Each operation presents a unique set of challenges:

Creating a File: When a new file is created, we need to determine how it can be accessed. We define our representation of file access rights, and how such rights may be computed at create time. Our model requires explicit specification of applications; only applications in a file's access list may gain access to that file. Further, we must not allow access by maliciously modified applications, so we grant access to application binaries.

Granting Access: File creation may not determine the entire set of applications that will access a file over its lifetime, so the user may need to add new applications to the access list manually. To ensure that a real user is behind a permission update, an authenticated channel between the user and the tool performing the permission change must be established.

Upgrading Applications: Applications will be upgraded over the lifetime of the system, so we must also update access policy accordingly. Since we use binary values as application identifiers, an application's access would be denied whenever its binary executable file is changed. To maintain the system's security, we must distinguish between a malicious binary modification and one resulting from a legitimate upgrade. Furthermore, performing this update must not require extensive operations such as traversing every file in the file system.

File System Manipulation: Users occasionally rearrange and delete files. File system utilities, e.g., `cp`, `mv`, `rm`, cannot be added to application access lists. If these utilities are added, malicious applications can execute them to circumvent file protection. Therefore, we must provide an authenticated channel between the user and such operations.

3.1 Creating a File

A file's lifetime begins with its creation. Providing application-level control of user files requires the specification of an initial file access policy. In this section, we define the representation of access policy and describe a means for computing this policy automatically.

Specifying a file access policy requires identifying specific application binary instances. The enforcement mechanism must differentiate an authentic application from one modified by a virus;

[2]Other forms of MAC, such as multilevel security [?], focus on the secrecy of the objects rather than the users or applications, but these approaches are too restrictive for the protection that we have in mind.

[3]From this point, we will use the terms *user files* or *user data* to mean high-value user files/data unless it is ambiguous.

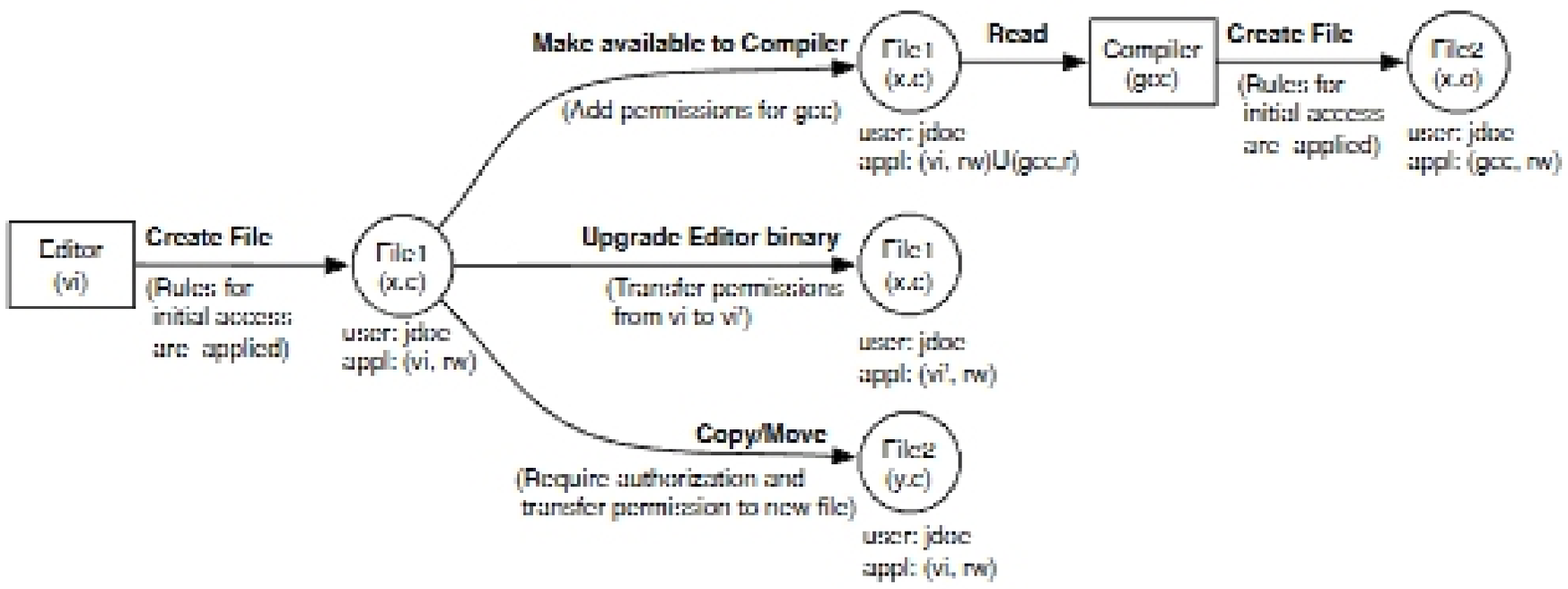

Figure 1: A summary of the administrative tasks for managing application access to files.

even the slightest variation in the application should deny access. Therefore, applications are identified by the cryptographic digest of their executable file[4]. This ensures that only authentic applications will gain access.

Our policy model associates a file with a user and a set of application binary identifiers and their rights (e.g., read, write, and execute) to the file in an access list. Such access lists are stored in the file's inode as is typical in MAC systems (e.g., via the ext3 extended attributes as used by SELinux). This ensures that the file object accessed is the one authorized.

Determining the actual policy to assign to a new file is non-trivial, but there are a number of hints that we can leverage. For example, the state of file creation, including the creating application, the type of file created, and the location of the new file, may be used to determine the correct set of applications and their rights to the file. On file creation, such rules are consulted and the desired access policy is derived.

3.1.1 Access Automation Rules

It is neither desirable nor practical to expect that a user can determine all the applications and the rights that they can use to access any high-value file. Some applications create transient but important files as a matter of course. Other applications create many thousands of potentially important files as they run. For example, consider again the use of `gcc` in Section 2. `gcc` creates object files as it compiles files, which are then used (often immediately) by a linker to create an executable. A common "make" may indirectly compile tens or hundreds of object files as the result of a single user action. For well-known applications with potentially high value files, such environments mandate automated tools for granting access to these files.

We have created an initial facility to automate this process. The system is configured with a set of *access automation rules* that indicate how files' access lists should be automatically defined as they are generated by an application. In the simplest instantiation, the operating system monitors all file creation, and new applications are added to the PinUP policy as directed in those rules. Because access list entries are added without user intervention, this can greatly reduce the burden of managing transient files or prolific applications.

An example of an access automation policy for our initial implementation is illustrated in Figure 2. The semantic of this policy is as defined above: applications are configured to automatically grant access (*add*) to some an application when a particular class of files is created. For example, the LaTeX application grants read and write access to Ghostview to all postscript and PDF files it creates. Note that this policy illustrates one of potentially many automated policy specification operations, e.g., one may want to revoke an applications access to a file once it is closed. We consider these additional policies briefly in Section 5.3, but defer deeper analysis of policy automation to future work.

3.2 Granting Access

Unfortunately, a file's set of authorized applications cannot always be determined at the file creation time. Therefore, the system requires a mechanism that allows the user to add and remove applications from a file's access policy. The key requirement of this mechanism is that the system must establish an authenticated channel with the user to ensure the request was not made by a malicious application.

In traditional discretionary access control systems (e.g., UNIX and Windows), users specify the usernames and rights for file access. Manually adding applications to application-level access lists presents a similar experience.

Assigning binaries to file access lists is abstracted to application identifiers and groups of applications. To abstract the binary identity from each file for user assignment, the user is presented the application name with the binary. The system must further justify the integrity of the binary file (e.g., via a signature from the code distributor at a particular date).

If a user frequently specifies multiple applications for several files, redundant operations can be reduced by creating predefined sets of applications. These sets of applications are mapped to *file types*, referenced by file type identifers, or `ftid`'s. This allows the user to simply specify an `ftid` and associated rights. To ease specification, each `ftid` has a corresponding human readable name. Allowing users to specify both applications and file types provides a flexible interface for users to influence file access policy.

Users must define file types before specifying them in access lists[5]. Additionally, the file type definitions must be stored in a location accessible for access control enforcement. File type definitions are stored in the file system volume metadata. This allows portable file systems to securely transfer between systems (given access lists support applications on multiple systems) and user home directories (created on separate partitions) to persist across multiple operating system installations (however, application upgrades must be resolved, see below).

[4]Cryptographic digests are a common method of binary code measurement [7].

[5]The system can predefine a set of known and useful file types.