Protecting Users From "Themselves"

William Enck, Sandra Rueda, Joshua Schiffman, Yogesh Sreenivasan,
Luke St. Clair, Trent Jaeger, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{enck,ruedarod,jschiffm,sreeniva,lstclair,tjaeger,mcdaniel}@cse.psu.edu

ABSTRACT

Computer usage and threat models have changed drastically since the advent of access control systems in the 1960s. Instead of multiple users sharing a single file system, each user has many devices with their own storage. Thus, a user's fear has shifted away from other users' impact on the same system to the threat of malice in the software they intentionally or even inadvertently run. As a result, we propose a new vision for access control: one where individual users are isolated by default and where the access of individual user applications is carefully managed. A key question is how much user administration effort would be required if a system implementing this vision were constructed. In this paper, we outline our work on just such a system, called PinUP, which manages file access on a per application basis for each user. We use historical data from our lab's users to explore how much user and system administration effort is required. Since administration is required for user sharing in PinUP, we find that sharing via mail and file repositories requires a modest amount of administrative effort, a system policy change every couple of days and a small number of user administrative operations a day. We are encouraged that practical administration on such a scale is possible given an appropriate and secure user approach.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection —*Access Controls*

General Terms

Security

Keywords

Access Control, Policy

1. INTRODUCTION

When access control was invented, computers were expensive and limited resources. Each computer supported several users who shared not only the CPU, but also the storage of these machines. Early access control systems were designed to protect the secrecy and integrity of each user's files from all the other users' processes on a single computer [16, 17]. The main concerns at this time were that: (1) a user's buggy program may modify the files of another user and (2) a nosy user may be able to browse another user's secrets by scanning her files.

The world of computing is very different now. Two major differences are: (1) the increased variety and lower cost of computing devices[1] and (2) the increased variety of threats against our computing devices. First, the advent of many inexpensive devices has created a situation where each user owns multiple devices, so there is no other user to restrict. Second, new threats have emerged due to the increased connectivity and ease of appropriating software that results from that connectivity. Now, users must be more concerned with the threat that their own processes may be malicious or have a vulnerability that a remote attacker can leverage. For example, a web browser client executes a variety of programs (e.g., plugins) to process browser content, but all these programs run with the full rights of the user (i.e., as users "themselves"). Some of these programs may be malicious, some may have vulnerabilities, but the user must trust all these programs with all their data.

We claim that the access control problem of the early days of computing has morphed into a new problem. In the current environment, users are isolated from one another by default and the main challenge is to manage the access of each user's applications. Sharing among users does occur, of course, but we claim that sharing can be modeled by a small number of mechanisms: email, web, and version control repositories. Thus, we believe that future access control models should leverage such natural isolation of users to simplify policy, provide a reliable control of user's data based on applications, and enable limited sharing without complicating policy significantly. Towards this end, we have developed the PinUP access control system [7], a Linux Security Module that binds permissions to applications, provides a rule language for expressing how files are shared among applications, and treating inter-user sharing as an exceptional case.

In the future, we envision that user administration should more closely mirror sharing among isolated users. Our access control infrastructure should be setup such that normal, predictable operation is handled by system policy (i.e., policy specified by system administrators and/or general-purpose policy rules). System administrators may have to make some changes to system policy to support variations in behavior, but these should be quite infre-

CSAW'07, November 2, 2007, Fairfax, Virginia, USA.

[1]The notion of a computing device is much broader than that of a computer of the 1960's and 1970's. We consider any device that may be programmed or whose software may be reconfigured, including cell phones and PDAs, as a computing device.

quent. In this vision, users will have to administer exceptional sharing, but this sharing is limited to a few, well-defined mechanisms: email, web, and version control repositories. Only when users apply these mechanisms do they need to consider the sharing implications. Otherwise, user files are isolated from other users. The PinUP system supports default isolation policies, so it is an ideal candidate to implement this vision as we discuss.

The approach above raises the following question, *"In a world in which we approximate acceptable system behavior through user isolation, how many exceptions to that approximation will occur in practice and how difficult will it be to correct the policy given that approximation?"* This question highlights the key tradeoff in the PinUP system. Inasmuch as the system can predict all uses of a file, no user or system interactions are necessary. Where such approximations are insufficient, the user is required to administer the policies (e.g., using PinUP -supplied tools) that diverge from the norm. Consider for example a user that creates a to-do list from using an ASCII editor such as `vi`. Later, the user may want to share that list with another user by emailing it to her or placing it in a shared file repository. Such behaviors are not predictable in any practical sense, and *must* be driven by user-specified interactions with the policy system. Just how much interaction the user has with the system is key to the usability of the system and access model.

The following sections attempt to answer this question by looking at historical data in our laboratory to assess the number of interactions administrators and users would have with the policy system, and at some level attempt to understand the viability of the administrative model. By looking at the use of file repositories and email behavior, we collect the number of operations that would require unpredictable sharing with other users, thus requiring exceptional policy changes. A central distinction that we make between *system* and *host* policy is that the system policy requires the efforts of a system administrator (i.e., someone not directly involved in the application), whereas individual users change the host policy. In the examination of email and repository modifications, we find that the number of policy changes required of users and administrators is not large: administrators must perform a policy change every 2.6 days for a lab of 65 people and users must update their policies every 8.6 days on average. This data motivates optimism that an access control system with the isolation of users at its core may become a practical approach enabling effort to focus on how users manage their own data among applications.

2. RELATED WORK

Recent operating systems provide Mandatory Access Control via SELinux [14], AppArmor [15] and TrustedBSD [1]. However, writing policies to describe the files and objects accessible from a domain requires a deep understanding of access control and operating systems, a task far beyond the abilities of most users. Moreover, such systems are oriented toward protecting mainly system files and not user's files.

Specifying which files an application may access, known as sandboxing, is a common approach to isolate applications [5, 15, 9, 3, 12]. However, sandboxing techniques operate solely on filesystem abstractions like files and directory paths. Ko et al. [10] attempt to overcome such limitation by specifying policies of expected application behavior. However, this approach is more appropriate for developers, as the policies are slightly complex and require knowledge about the expected behavior of the application. Similar to the PinUP model, LIDS [12] allows files to be bound to specific applications, but the interface is restricted to system administrators. Contrary to these systems, the goal of PinUP is specifically to offer a user-oriented platform.

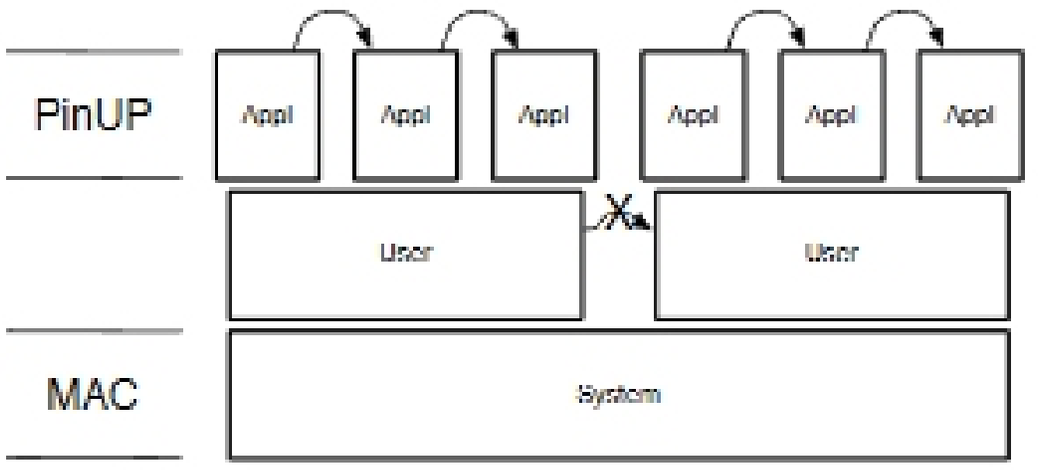

Figure 1: PinUP provides control over user applications, assuming users are isolated by default and a mandatory access control approach protects the system.

Lai and Gray [11] propose a mechanism to allow the user to specify the list of files an application may access. Unfortunately, the list must be specified each time an application is started. TRON [2] offers a similar approach, where a user's initial shell has capabilities for the entire home directory, and each user must explicitly create child processes with less capabilities. A major drawback to both Lai and Gray and TRON is that the user is by default responsible for setting up the environment in a proper way. Additionally, these operations must be performed every time an application is run. The RBAC model was also applied to allow users to run application inside defined subdomains [8]. While this approach allows users to specify the rights of each one of their subdomains only once, the user is still responsible for the initial setting and then for switching to the proper subdomain every time an application is run.

More recently, Polaris [18] extends Microsoft Windows allowing users to sandbox an application by indicating the set of files that will be accessed. Polaris uses "installation endowments" to provide applications capabilities to access system files, thereby focusing policy specification on user files. In an effort to provide greater flexibility, Polaris allows user to specify multiple sandboxes for different instances (pets) of the same application. Usability studies indicate that this option leads to user error where the wrong pet is selected, thereby compromising file security [6].

We claim these previous approaches make user administration difficult because users have to work with filesystem abstractions to sandbox applications. Our experience is that users only share files via a small number of mechanisms, such as email and repositories. Our goal is to determine whether user administration can be made tractable if users are isolated by default and can only share files using these mechanisms. We examine the hypothesis using an access control system called PinUP that isolates users by default. In our evaluation we demonstrate the administrative effort system administrators must exert to configure the default, general policies that PinUP provides for isolation and the administrative effort of users for sharing files between users.

3. PINUP

In this section, we examine the PinUP system to make clear the type of policy administration that results from PinUP's approximation of system behavior. Figure 1 shows PinUP's view of access control. PinUP is not another discretionary access control (DAC) approach, but rather, it is an overlay on a mandatory access control (MAC) base. PinUP is designed under two key assumptions: (1) the MAC enforcement, such as SELinux [14] and AppArmor [15], protects the system and (2) interaction between users is not allowed by default. First, since MAC enforcement protects the system, PinUP only protects user data. Second, since users do not interact by default (i.e., user interaction is not the norm), PinUP focuses

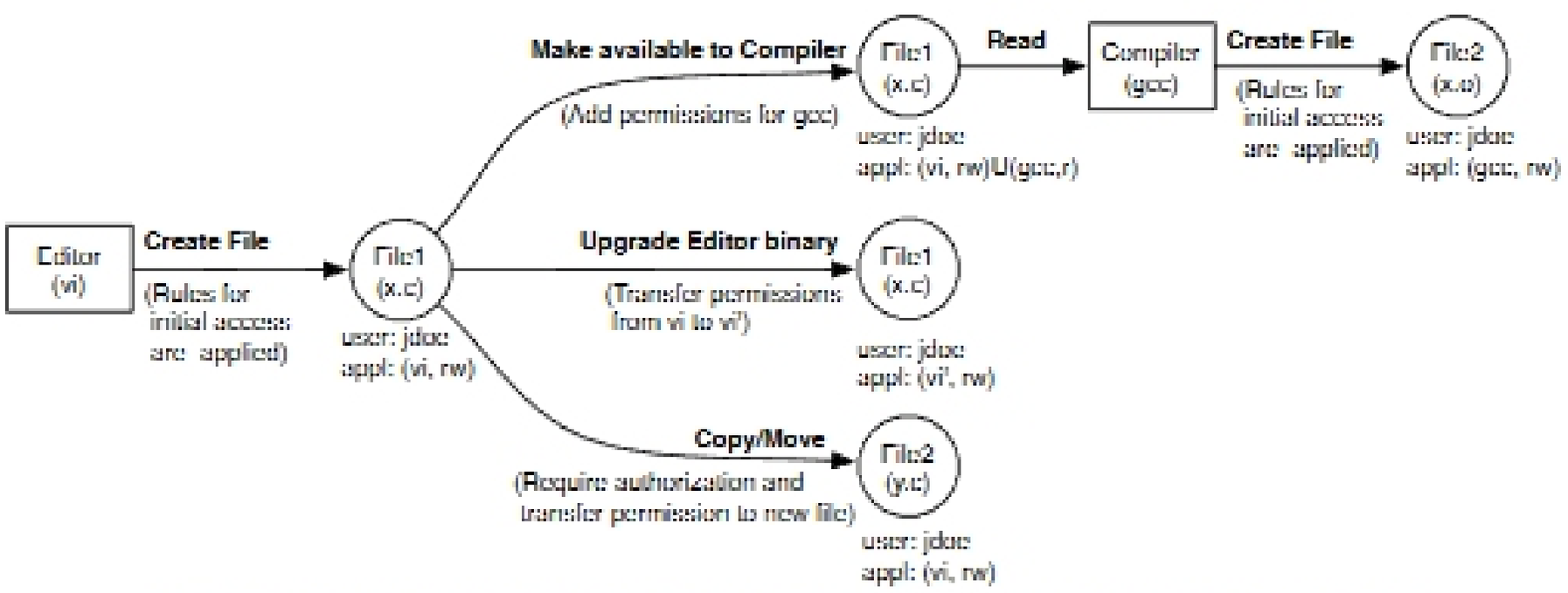

Figure 2: A summary of the administrative tasks for managing application access to files.

on managing access between each user's own applications. Since user interaction is infrequent, PinUP requires that the user or administrators perform manual operations to allow sharing. In this section, we examine the PinUP mechanisms, and in the next section whether the assumption of infrequent, manual administration appears feasible.

PinUP manages file access by controlling which applications can access which files for each user. When a file is created, only the creating application has access to the file by default, but PinUP *access automation rules* can describe how files created by one application can be accessed by another and how files of certain types can be accessed by certain applications. Changes to these PinUP policies can be made, but every change requires per-use user authentication (similar to the administrative interfaces of the OS X operating system). In particular, **user processes do not have the ability to modify PinUP policy**. We detail the implementation of the PinUP system in an extended technical report [7], and defer design issues to that text.

We now illustrate PinUP via the example in Figure 2. Assume that the user `jdoe` creates a file `x.c` using 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 `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–in PinUP policy rules can be stated that 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 2, PinUP presents a number of other interesting design and implementation challenges. For example, how one 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.

3.1 Host-Level Policy

At a very high level, PinUP exists to enforce a rudimentary intra-application information flow policy. In this, there are two facets of policy that are relevant to safe operation. First, the enforcement must determine how the output files created by an application should be automatically associated by other applications, e.g., the `.o` object file output by `gcc` should be automatically readable by the `ld` linker. In another example, a `.qdf` Quicken file should only be accessible via the Quicken program. The set of policy rules that govern these permissions is a reflection of the workflows and environmental practices of the user/host. Constructing these policies is essential to closing the vulnerabilities presented by existing access controls, and is an open problem. However, we expect that many application usage policies will be common across all users.

Where workflows are not known or where the user needs to perform atypical manipulation of files, PinUP tools must be used to modify the application associations. For example, a user who wishes to burn an encrypted version of the Quicken file onto a CD would need to modify the PinUP permissions to make associations between the files and the program that will manipulate them. Such policy changes are implemented in the current system through `pinmod`, which requires the user to enter her password before changing the internal policy associated with a file.[2]

Note that these operations apply only to those files the user deems to be of "high-value". PinUP access automation rules will describe which files are to be governed by PinUP. Other files not being governed by PinUP will only be subject to the normal system access controls.

3.2 Distributed System Policy

Since users each have their own computing devices, enabling sharing among users becomes a distributed systems policy issue. Thus, in addition to some user administration to enable sharing, some system administration will also be required. In this section, we examine the types of system administration that the PinUP approach needs.

In the distributed case, each host in the distributed system would apply a common, system-administered policy that states relationships between applications and the rights they have to read, write, and execute the file content. The only additional systemic requirement this extended model would place on the host is a service for obtaining and updating the access policies to be enforced. Put another way, an environment-wide policy would be obtained from a central authority and would supersede the local user/host policy.

The distributed service would need to extend the model of software identity to include versioning. For example, many different

[2]This ensures that a malicious application cannot circumvent the protections by launching the `pinmod` without the aid of user.