PinUP: Pinning User Files to Known Applications

William Enck, Patrick McDaniel, and Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Laboratory
Department of Computer Science and Engineering, The Pennsylvania State University
{enck,mcdaniel,tjaeger}@cse.psu.edu

Abstract

Users commonly download, patch, and use applications such as email clients, office applications, and media-players from the Internet. Such applications are run with the user's full permissions. Because system protections do not differentiate applications, any malcode present in the downloaded software can compromise or otherwise leak all user data. Interestingly, our investigations indicate that common applications often adhere to recognizable workflows on user data. In this paper, we take advantage of this reality by developing protection mechanisms that "pin" user files to the applications that may use them. These mechanisms restrict access to user data to explicitly stated workflows–thus preventing malcode from exploiting user data not associated with that application. We describe our implementation of PinUP on the Linux Security Modules framework, explore its performance, and study several practical use cases. Through these activities, we show that user data can be protected from untrusted applications while retaining the ability to receive the benefits of those applications.

1 Introduction

Users frequently download, patch, and use largely untrusted applications from the Internet. The common approach to limit exposure to untrusted applications–sandboxes–provides an easy mechanism to protect a system by containing an application within a resource-constrained environment. However, sandboxes are limited for managing programs such as email clients, office applications, and media-players that must necessarily access sensitive user data. Conversely, current system protections provide little defense against hidden malcode; downloaded applications in commodity operating systems can access any and all data that the user can. In a sense, traditional OS protections do not limit the sharing of information between applications and thus fail to provide least privilege over user data.

Interestingly, applications processing sensitive user data (e.g., Quicken) often use application-specific files. Further, our investigations indicate inter-application sharing is frequently well-defined (e.g., localized to the creating application, or limited to applications within a suite), fitting within recognizable workflows. Ignoring for the moment system operations such as backups, the degenerate and most frequent workflow occurs when a user file is created and used only by a single application. Additionally, in some environments, inter-user sharing occurs between systems via only a few applications (e.g., email) [8]. This reality represents an opportunity for new protection schemes. We propose mechanisms that govern access by ensuring that every user application adheres to its expected workflows. Thus, in contrast to sandboxing which limits access based on a general notion of containment, these new protection mechanisms identify access policy as positive application-aware workflows.

In this paper, we propose the PinUP[1] access control overlay system. We design and implement a mechanism to pin user files to applications (and thus enforce application workflow protections). In doing so, we found that realizing this conceptually simple policy in a protection mechanism is more complex than one might initially expect. Such complexity lies not only in understanding the often subtle relationships between applications and data, but also in anticipating and dealing with unexpected but essential sharing. We implement PinUP upon the existing Linux Security Modules framework [31] and we show that user data can be protected from untrusted applications.

We now illustrate the new protection mechanism by considering a Quicken accounting application scenario. The high value files created by Quicken, e.g., `.qdf` files, only need to be accessed by Quicken. It seems natural that the system should enforce that policy. Of course, the user may wish upon occasion to email the Quicken file to an accountant. However, in this case, the email client should only have read access to the file and only for the short duration of sending the email. A protection mechanism must handle such exceptions. Such exceptions are at the heart of the mechanism complexity, and their recognition and handling is considered in depth in the latter sections of this paper.

PinUP is intended to augment existing security services within the OS. It is designed to operate above MAC sys-

[1] PinUP conceptually "pins" files to specific applications.

tems such as SELinux [19] and AppArmor [20], which we believe are better suited to protect system files. However, these system level mechanisms are not appropriate for protecting user files. 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. The PinUP access control overlay provides a straightforward method to protect the user files. We explore the degree of applications this works well in Section 6.

The remainder of this paper considers the design and implementation of the PinUP system. This work represents a new tack at achieving system level least privilege. In so doing, we focus our 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 as an administrator).

Emerging approaches, such as SELinux [19] and AppArmor [20], enforce 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.

PinUP builds upon the existing MAC infrastructure to sustain the integrity of the underlying system, enforcing bindings between applications and files. PinUP policies work in unison to govern flows of data between applications–the aggregate behavior of which will define legal application workflows. For example, PinUP can restrict Excel spreadsheet `.pdf` output to printing applications, whose output is restricted to print spoolers, etc. However, PinUP is not a yet another DAC system; every change to PinUP policies occurs through administrative tools that require per-use user authentication. In particular, *user processes do not have the ability to modify PinUP policy.*

[2]Other forms of MAC, such as multilevel security [4], 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.

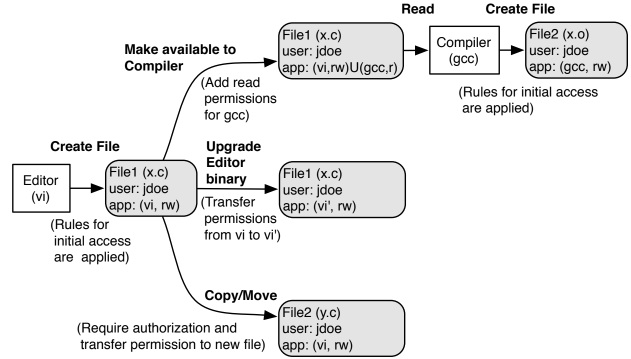

Figure 1. A summary of the administrative tasks for managing access to files.

Moreover, policies only need modification when application or local administrator provided polices are insufficient.

We now illustrate important tasks that PinUP must consider via the example in Figure 1. Assume that user `jdoe` creates 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 are allowed subsequent file access. In this case, `vi` is given read and write access. `x.c` is a source file, so user `jdoe` desires read-only access for a compiler through a command line tool. Note that such a policy need not be manual–an alternate policy may dictate that all `.c` files created by `vi` are made available to `gcc` automatically. The compiler can then create a new object file `x.o` that it can write and can be read by the linker. Similar policy enforcement will occur as applications cascade over data to consume and create protected files.

Figure 1 also illustrates a number of interesting design and implementation challenges. For example, how one securely identifies applications and propagates 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 MAC systems. However, due to 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) administrative functions that enable specification and evolution of such rights and (2) an authorization mechanism that can enforce such access. 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]From this point, we will use the terms *user files* or *user data* to mean high-value user files/data unless it is ambiguous.

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 administrative operations that must be supported to enable application-level control of user files. Each operation presents unique challenges:

Creating a File: Initial access rights are are required at file creation. If rights are not set correctly, no applications may be able to read or write the file. Further, we must not allow access by maliciously modified applications, so we grant access based on application binaries.

Granting Access: Determining the set of applications that can access a file is non-trivial. A complete list of applications may be indeterminable at file creation, therefore manual manipulation is necessary. However, the mechanism must establish an authenticated channel with the user.

Upgrading Applications: Applications will be upgraded over the lifetime of the system, so access policy update accordingly. This process must be secure and efficient.

File System Manipulation: Users occasionally rearrange and delete files. File system utilities, e.g., `cp`, `mv`, `rm`, cannot be added to application access lists, otherwise malicious applications may acquire indirect access.

3.1 Creating a File

A file's lifetime begins with its creation. Providing application-level control of user files requires an initial file access policy. As a safe default, a high value file's access list should at least contain the application that created. More complex initial policies are described in Section 3.2.

File access policy specification requires identifying specific application binary instances. The enforcement mechanism must differentiate an authentic application from one modified by a virus; even the slightest variation in the application should deny access. Therefore, applications are identified by the cryptographic digest of their executable file (a common method of binary code measurement [29]). This ensures 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 with the file as is typical in MAC systems (e.g., via the ext3 extended attributes as used by SELinux).

3.2 Granting Access

Determining the complete set of applications to assign to a new file is non-trivial. As a safe default, a high value file's access list should at least contain the application that created it. Beyond this, there are a number of hints that can be leveraged. 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. In this section, we discuss methods of assigning an initial access lists via access automation rules. However, this list can not always be determined at file create time; therefore, we must provide a method of manual policy manipulation.

```
# PinUP System Configuration Rules
# <Application> <file type> <add> <application>

# Allow "ld" to access all object files ".o"
# generated by "gcc"
gcc object add ld

# Allow "gv" to access all object files ".ps/.pdf"
# generated by "latex"
latex postscript add ghostview
latex pdf add ghostview
```

Figure 2. Example access automation rules for a simple PinUP system configuration.

3.2.1 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* indicating how files' access lists should be automatically defined upon creation. 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 application's access to a file once it is closed. We defer deeper analysis of policy automation to future work.

3.2.2 Manual Policy Specification

While we envision access automation rules to encompass the mass majority of policy specification, exceptions occur

when file's set of authorized applications cannot always be determined at a file's creation time, e.g., temporarily granting an email client access. 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 a mutual authenticated channel between the system and the user.

Both directions of authentication are important. The system must authenticate the user to ensure a malicious application has not requested the policy change, and the user must authenticate the system to ensure a malicious application cannot intercept and modify policy modification requests. Unfortunately, the latter is an existing problem in system security. Therefore, our current design only accounts for authenticating the user to the system. Note that while mutual authentication is desirable, a malicious application acting as the user poses a more significant threat.

Currently, we establish an authenticated channel by requiring the user to enter a password. While we use a password to establish the authenticated channel, any mechanism that allows the system to authorize policy modification may be substituted. However, we note that simply prompting the user for confirmation without any context is a bad approach. Users commonly accept all operation requests without reading why the operation was requested.

Once the authenticated channel has been established, the access policy may be modified. In traditional DAC 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.

We abstract the assignment of binaries to file access lists using numeric application identifiers and groups of applications (similar to UNIX *uid* and *gid* values). To aid usability, the user is presented the application name associated with the numeric identifier. If a user frequently specifies multiple applications for several files, redundant operations can be reduced by creating predefined application groups, referenced by application group identifiers (*agid*s). Hence, users need only specify an group name and the desired rights. Allowing users to specify both applications and application groups provides a flexible interface for users to influence file access policy. While application groups must exist before they can be used, the system can predefine useful application groups that can be extended by the user as needed.

3.3 Upgrading Applications

The file access policy restricts which application binary instances can read, write, or execute a file. However, systems commonly upgrade application binaries for improved features and security. Therefore, the system must differentiate between a binary modification resulting from a legitimate system upgrade from one resulting from a virus. Upgrades are managed by updating access policy to reference the latest binary instantiation of the application.

The update of file access policy corresponding to application upgrade requires special attention. The system requires a trusted mechanism to execute in tandem to existing upgrade management tools. This tool must establish an authenticated channel with the system administrator. Then, it measures the digest of both the old and new application instances. Finally, all references to the old digest are replaced.

This trusted application upgrade procedure need be intensive. In PinUP, each file stores a list of application identifiers (*aid*s), and application group identifiers (*agid*s). The associated application digests are stored in a central location, hence, the process need only inspect the central store.

3.4 File System Manipulation

Files are occasionally rearranged and deleted. While these file operations do not directly access content, they influence the file's state (delete removes the file entirely) and position within the file system. Adding commands such as `rm`, `cp`, and `mv` to a file's application access list allows malware to circumvent the protection system by executing the command. Hence, file operations require special attention.

Users perform file operations with special utilities. We propose a PinUP-aware utility for each operation, e.g., `pinrm`, `pincp`, and `pinmv` for `rm`, `cp`, and `mv`, respectively. Each utility establishes an authenticated channel with the user. Like the policy manipulation utilities described in Section 3.2, these file utilities currently acquire an authenticated channel by prompting for the user's password (such operations are expected to be infrequent, similar to manual policy specification). Again, other authenticated channel mechanisms can be substituted. These special file utilities have implicit access to files by virtue of the authenticated channel, because they ensure the operation is desired by the user. Finally, the PinUP file utilities can be designed to function as drop in replacements that only require an authenticated channel when modifying a protected file, providing seamless integration with the existing system.

4 Implementation

Figure 3 describes PinUP from an implementation perspective. In the figure, the user has specified application and application group lists in the file system's volume policy (note that a system configuration could require the application and application group lists to be specified by a administrator upon system installation). This is done using human readable names not shown in the figure. Application A, Application B, and an unknown application all attempt to read and write a PinUP protected file. The file system permissions indicate that the user may read and write the file. Each application was invoked by the user that owns the file; therefore, without PinUP in place, all access requests would be granted. After performing a successful user permission check, PinUP is invoked. PinUP obtains the application binary digest from the current process table. Next,

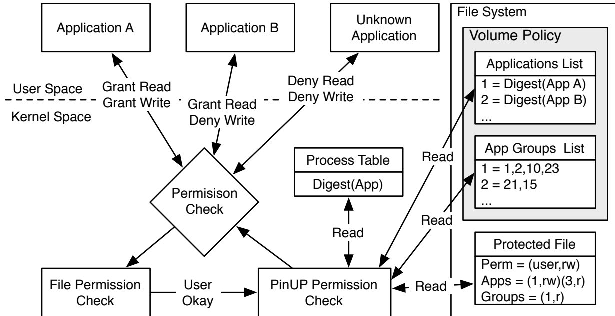

Figure 3. The PinUP File Protection System. The user specified volume and file policy indicates that Application A (`aid 1`) can read and write the file; however Application B (in `agid 1`) may only read it.

the file's application access lists (for applications and application groups) are read from the file's attribute storage. The identifiers are looked up in the appropriate application and application group lists defined for the file system volume. Finally, the digest for the current process is matched against the entries. In the figure, Application A has been specified explicitly on the file with both read and write permission (in the Named Applications List), therefore both requests are granted. Application B belongs to application group 1, which has read permissions on the protected file (in the Named Application Group List). Therefore, Application B can read, but not write the file. Finally, the unknown application is not listed in the file's access control policy, therefore both read and write are denied.

The design goals of Section 2 present a number of implementation challenges. The remainder of this section describes our implementation of PinUP using the Linux Security Modules framework and custom administrative tools.

4.1 Application Identification

The PinUP model security hinges on the method of identifying applications. We have chosen to identify each application using the cryptographic digest of its binary executable. Identifying an application by its digest ensures not only the application is correct, but also no modifications have occurred (e.g., modified by a virus).

Calculating the digest of a process is straightforward in the standard case. On process creation, we use a Linux Security Module (LSM) hook, `bprm_set_security`, to gain access to the specific executable file. The contents are digested and the resulting value is stored in the new process's `task_struct` for later comparison (see Section 4.4).

Unfortunately, not all applications follow the standard binary executable model. Many system utilities are interpreted scripts written in various languages: Perl, Python, etc. In this case, both the binary interpreter and textual script define the application. To accommodate, PinUP digests both the interpreter and the script. The two values are XORed and stored as the digest value used to identify applications. Note that the interpreter and script are identifiable in both user and kernel space; however, the latter requires careful inspection the Linux binary loader data structures.

The digest technique described for standard binaries and scripts provides adequate security; however, it does not account for runtime dependencies. In Linux, applications read the trusted system file `/etc/ld.so.cache` to determine dynamic library locations. This file and the dynamic libraries it references are a part of the system configuration and are out of the scope of PinUP's protection. As such, PinUP relies on system protections, e.g., SELinux, to control the integrity trusted system dynamic libraries and interpreter modules (e.g., for Perl, Python, etc.). The interaction between system protections and PinUP is inherently subtle. We defer analysis of this interaction to future work.

Environment variables (e.g., `LD_LIBRARY_PATH`, `PERLLIB`, `PYTHONPATH`, etc.) may also determine library and module location. Malicious modification of environment variables to load alternate libraries are well known. The best method of mitigating this threat is to disable the use of these environment variables altogether, as they are an insecure substitute for an improperly configured system.

As an alternative solution, future versions of PinUP can use an existing LSM hook to authorize a library load. The `file_mmap` LSM hook mediates all library loads by the dynamic linker. We note that kernel-based integrity measurement (e.g., Linux IMA [22]) uses this LSM hook to ensure the measurement of all dynamically linked libraries. Thus, PinUP can ensure that a library meets an expected hash value (e.g., is a system library). For scripting languages, the interpreters must be modified to perform such authorization. We note that integrity measurement requires the same modifications, so PinUP can leverage these as well.

4.2 Encoding Policy in the File System

The second component of the PinUP implementation is the method of encoding policy within the file system. To simplify the policy description and maintenance, e.g., upgrading applications, the PinUP policy encoding spans the file system and the files. The file system contains information relevant to all files, while files contain only enough information to describe specific access control restrictions.

As described in Section 3.2, users specify file access lists in terms of applications and application groups. Similar to how UNIX systems use centrally located files to define users and groups (e.g., `/etc/passwd` and `/etc/group`), a PinUP enabled file system maintains a special file, `.pinup`, at its root. This file specifies the application digest (described in Section 4.1) along with a unique numeric application identifier (`aid`) and a human readable name. Similarly, the file contains a list of application groups specifying the unique application group identifier (`agid`), human readable name, and list of `aid`s belonging to the group. All files in

the file system volume used the defined aid and agid values to specify per-file access lists and permissions.

The per-file access lists and permissions are stored in each file's inode extended attributes (xattr). PinUP defines two new extended attribute handles, security.pinup.apps and security.pinup.groups, for lists of applications and application groups, respectively. Each list is a binary array (similar to the character array stored by SELinux) that specifies either the aid or agid and the associated permissions, i.e., read (r), write (w), or read and write (rw).[4] For storage purposes, the identifier and permissions are marshaled into a single 32-bit integer, using the two most significant bits for the permissions.

4.3 Specifying Policy

Users and administrators modify PinUP policy using special utilities. The PinUP console, pincon, manipulates the file system volume policy data, and pinmod (like chmod) specifies permissions on specific files. Both utilities use human readable names associated with the aid and agid values. In many ways, working with PinUP policy parallels administration of user-based access control, which uses uid and gid values to identify users and groups.

The pincon utility allows users and administrators to add, delete, and otherwise modify the application and application group definitions for a file system volume. When adding a new application, pincon calculates the SHA1 digest of the provided file path and assigns the next highest aid value. Similar operations result for application upgrade. pincon also allows the user to create application groups. New groups are assigned the next highest agid value, and the user adds and deletes applications from a group using the human readable names. Note that deleting numeric identifiers has identical concerns as reassigning uid values. Finally, after each update, pincon notifies the PinUP kernel module to reread the policy by writing the path of the policy file to a SecurityFS interface.

The pinmod utility references the aid and agid values managed by pincon when specifying access permissions on files. Users administer their own policy using pinmod much in the way they currently use chmod. Users add and delete applications and application groups by name along with a right (r/w/rw). Accordingly, pinmod updates the extended attribute by encoding the identifier and right.

Both pincon and pinmod provide special administrative privilege within the PinUP system, and therefore are included in agid 0, which has implicit access to all PinUP protected files. This functionality is similar to how the root user in UNIX systems is treated with respect to user file system protections. Applications within agid 0 must

[4]We chose not to initially include the execute right. It introduces unnecessary policy complexity for a right rarely applied to high-value user files. When access control based on execute rights is required, the discretionary file system execute bit can be used.

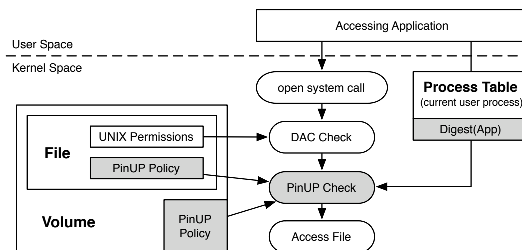

Figure 4. PinUP Enforcement Mechanism

take special precautions to avoid invocation by a malicious application. As discussed in Section 3.2, each invocation must authenticate the user's intentions. Our implementation prompts for the user's password using PAM [23] authentication. Note that PAM authentication requires root privileges, therefore the executables are owned by root with setuid permissions. Implications of this ownership are discussed in Section 4.4. Note that this configuration is also required to modify the .pinup file and file extended attributes.

Finally, we understand that command line utilities are often awkward for some users. Therefore, our utilities are built upon a special PinUP library allowing logic reuse. We expect future PinUP implementations to include graphical counterparts to pincon and pinmod.

4.4 Enforcement

The PinUP policy specified by pincon and pinmod is enforced by the PinUP Linux Security Module (LSM). The implementation uses the inode_permission hook to determine when a process attempts to access a file. Shown in Figure 4, the enforcement mechanism retrieves the application identity (as described in Section 4.1) and file policy (stored in inode attributes as described in Section 4.2), interprets the permissions (using an internal representation of the policy stored in the .pinup file described in Section 4.2), and determines if the accessing application may perform its desired action, i.e., read or write.

As PinUP is only concerned with reads and writes, the module interprets execution requests as reads and append requests as writes. PinUP begins by iterating through the list of application identifiers specified for the file. If the requested operation meets the permissions for a specified aid, the volume policy is consulted for the digest value associated with the aid. If the value matches that of the current process, access is allowed. This iteration is continued until a match is found. If no match is found, PinUP moves to the application group list. PinUP iterates through this list with an extra level of indirection, consulting each aid associated with a agid when the specified rights allow the requested operation. Finally, if no specified agid contains an application with a digest value matching the current process, PinUP checks agid 0, which contains the list of administrative operations. Upon failing to match any digest value, the hook returns -EACCES, indicating access denied.

This mediation technique works well to protect files; however, it is inappropriate for directories. A directory is opened and written to whenever the contents its contents is modified. Protected files within a directory will have different access control policies. As a result, determining which applications can read and write a directory is difficult. However, completely ignoring directories is not appropriate either. If a user has write permissions to a directory, `rm` can override file permissions and remove files within it.

PinUP controls directory access by mediating the operations performed to the files contained within. PinUP mediates file removal using the `inode_unlink` LSM hook. If a file contains a PinUP extended attribute, the hook only requires the application to be in `agid 0`. A similar technique is applied when moving and renaming files using the `inode_rename` hook, with the addition that protected files cannot be be moved between file systems where volume policy may redefine numeric values in access lists. The special PinUP system utilities used to perform file manipulation are discussed in Section 4.5.

Finally, PinUP mediates file operations that modify the extended attributes used by file policies. Using the `inode_setxattr` and `inode_removexattr` LSM hooks, PinUP ensures the current process is in `agid 0` *and* that the process's `uid` matches that of the file's owner. The latter protects systems that requiring root access to set extended attributes, thereby forcing `pinmod` to execute as root. Without this check, cross user policy modification may occur.

4.5 File Manipulation Utilities

The PinUP implementation requires one final component to allow file manipulation. As discussed in Section 3.4, `pinrm`, `pinmv`, and `pincp` use password authentication to establish an authenticated channel. As with `pinmod` and `pincon`, the file manipulation utilities authenticate the user using PAM, thereby requiring the same root ownership with `setuid` permission. PinUP is designed to supplement user-based access control, therefore, each file utility subsequently relinquishes permission using `seteuid()`.

All three utilities are included in `agid 0` indicating they are trusted applications. Due to the access control logic in the `inode_unlink` and `inode_rename` LSM hooks, `pinrm` and `pinmv` simply execute the `unlink()` and `rename()` system calls after authenticating the user. `pincp`, however, must also copy extended attributes.

5 Performance

In this section, we consider the run time costs of PinUP. These costs principally occur at load time (application hashing) and during the enforcement of policy (access checks). In support of these tests, we have implemented the PinUP file protection system (as described throughout) on the Linux kernel 2.6.17 within the Ubuntu v6.10 distribution. All experiments were performed on a 2.8GHz Intel Pentium 4 with 1GB RAM and observed 95% confidence intervals less than two orders of magnitude of the mean.

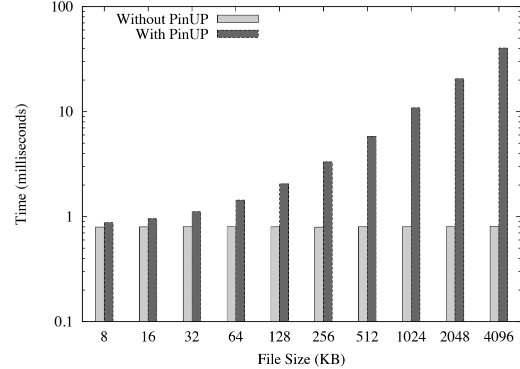

Figure 5. Process Start-up Cost

The initial PinUP user space utilities and libraries consist of around 4,500 lines of C. The libraries provide a simple interface for interacting with PinUP file and volume policy, digesting applications, and performing user authentication. The PinUP kernel code consists of an additional 2,000 lines of C. The code currently exists as an LSM module and small kernel source patches. The open source code is available for download from `http://siis.cse.psu.edu/pinup.html`.

5.1 Load Time Costs

PinUP performs a cryptographic digest of every executable as it is loaded. The time required to digest a binary as it is loaded is directly dependent on the size of the file. Thus, an understanding of load cost must first be based on an analysis of the size of executables. A quick investigation of executables in the `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` directories of Ubuntu v6.10 (2327 files in total) ranged from 4MB to a few kilobytes. We found that more than 60% of executables are smaller than 20KB, and more than 80% are less than 100KB. Hence, it seems that the majority of executables are indeed small. However, one must be careful in assuming that that applications are used at the same rate–some large executables may be extremely popular. Thus, for the experiments that follow we conservatively measure the costs of executables of powers of two between 8KB to 4MB. We created executables of precise sizes by padding a small application with `nop` operators.

We measured load time costs in two ways. First we performed a microbenchmark of the digest mechanism alone. Unsurprisingly, we found the digest cost scales linearly with 16KB files taking around 100 μs and 4MB files taking around 40 ms. Next, the experiment was repeated, but instead, the measurement code was instrumented in userspace, thereby capturing the observable startup cost. Figure 5 summarizes our findings. Interestingly, the start-up cost does not vary with file size when PinUP LSM is not used. This results from optimizations in the kernel process loader where code pages are read on demand, whereas with

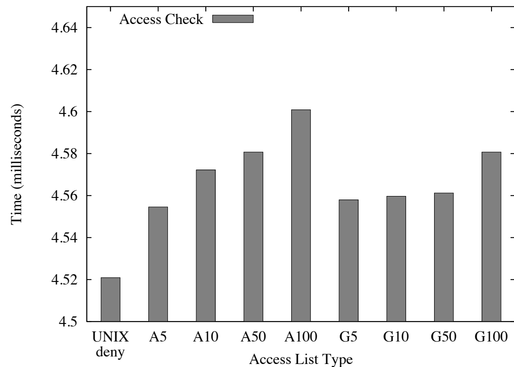

Figure 6. Permission Check Latency

PinUP the entire binary must be read before execution can begin. Figure 5 also demonstrates that without PinUP, a process start-up is only around 0.8 ms. For 16KB files, which accounts for nearly 60% of executables, only a 0.2 ms overhead is added. Even for the largest binaries, 4MB, only a 40 ms delay is added. Such files are typical of graphical applications that are infrequently executed. In these cases, the additional 40 ms ($1/25th$ a second) is not noticeable by the user. Thus, PinUP incurs negligible load-time overhead.

5.2 Policy Enforcement Costs

PinUP policy is enforced through the kernel hooks to the LSM. Called only after successful completion of the normal file system access checks, the enforcement code searches access lists to see if the governed application is allowed to access the file. We analyze these costs by investigating the search time as a function of access list lengths (the central determinant in enforcement cost). The tests measure two cases: when the application list is directly on the file and when applications referenced by a single application group.

Figure 6 displays the results of accessing a file with an access list of 5, 10, 50, and 100 applications (prefixed by 'A') and a file with an access list including one application group referencing 5, 10, 50, and 100 applications (prefixed by 'G'). An experimental baseline is set by measuring (failed) access cost (not invoking PinUP) denoted *UNIX deny*. The experiments indicate an access check overhead on the order of tens of microseconds. Note that the y-axis in Figure 6 starts at 4.5 *milliseconds*. Hence, even for unrealistically large lists, the runtime costs of PinUP enforcement are completely dominated by other access costs, e.g., context switches and file I/O. Finally, the experiment indicates greater efficiency when using application groups. The small difference is explainable due to the reduced I/O access, i.e., only one `agid` is read from disk.

6 Evaluation

PinUP targets the objects of risk: user files. These files are often subject to malware and commonly contain sensitive data and/or cannot be easily recreated in the event of compromise. Many previous works attempt to protect user files by applying application "sandboxes." In such works, applications are sandboxed either by specifying resource policies for applications (e.g., Janus [13]) or relinquishing privileges before execution (e.g., TRON [5]). This section compares PinUP to the application sandboxes, showing that PinUP provides greater protection more straightforward administration. We then consider three usage scenarios to evaluate the ability to automate administrative actions.

6.1 Comparison to Sandboxes

PinUP provides greater protection for user files than application sandboxes. Traditional application sandboxes do not provide explicit protection of user files. Rather, user files are indirectly protected by ensuring untrusted applications execute in a limited environment. However, to ensure user files are protected, the sandbox model assumes all untrusted applications can be identified. The complex environments in current systems allow malicious software to be introduced via unpredictable means. Additionally, there is no separation in the level of trust in a trusted application. An application trusted to access a certain type of user file may incorrectly gain access to other files. Hence, we believe PinUP is more appropriate for protecting user files as the set of trusted applications for a specific file can be easily determined from the application access list for that file.

PinUP often requires less overall user effort than application sandboxes. Traditional sandboxing techniques require the user to ensure a properly restricted execution environment *every* time the application is executed. For example, in TRON, the user must manually execute `tron_fork` to delegate less privileges to a child process. In contrast, a PinUP permission need only be specified once per file, and can be often created automatically. For policy-based sandboxes such as Janus, the may automatically reuse previously defined policies; however, the user must still perform the arduous task of creating a new custom policy for every new application. This requires determining the entire set of files the application needs to access. Such file sets are often esoteric and vary between applications. In PinUP, the user must only modify automatically created policy when a file needs to be accessed by a new application.

Applications sandboxes also limit flexibility at run time. For example, sandbox policies typically provide file protection by specifying file paths, or more commonly, directory paths. In doing so, the sandbox restricts where the user may store files, which can be cumbersome or confusing. PinUP, on the other hand, is not restricted by directory structure, thereby allowing users to store `.qdf` files along side related `.doc` files without compromising the security of either.

6.2 Usage Scenarios

To further examine PinUP's appropriateness for a user oriented platform, we created three example scenarios: (1) a document development lifecycle; (2) a GUI application

to edit files; (3) web browser file retrieval. The experiments study the policy implications of the file operations incurred by real application use in each scenario. The operations were inferred from file system open system calls collected by the OS kernel. Our primary goal was to investigate the appropriateness of the workflow model for access automation rules described in Section 3.2.1. With minor exception, all enforcement scenarios conformed to our expectations about policy–the file accesses are restricted to well known applications as determined user workflows.

Document Lifecycle: The first example consists of retrieving the LaTeX document files from a Subversion repository, editing the document files using `vim`, building a PDF, emailing the PDF, and receiving the PDF from an email. This example requires two PinUP access automation rules. The first permits editors to modify LaTeX files downloaded using subversion, and the second permits the Adobe viewer `acroread` to view the generated (from `pdflatex`) or downloaded (from the email client on receipt) PDF. We envision that since the user manually selects the files to send, we should not give the email client access to all PDF files by default. Thus, the user will have to grant the email client access to the specific files using `pinmod`. As we have shown in previous work [8], some common environments exhibit minimal inter-machine sharing implying explicit policy changes should not be overbearing.

Graphical Editor: In the second example, we examine use of a GUI editor, OpenOffice, to edit the files rather than `vim`. It turns out that the GUI editor works better with PinUP than others because it does not generate backup files. Some editors (e.g., `vim`) save files by renaming the original file to a backup file before creating the saved file in a new file object (of the original's name). This is problematic because a file's application access lists are stored with the inode. Note, however, that OpenOffice did not use this save technique. In fact, such function is not strictly necessary to `vim`, as it will not create a backup if the rename is not authorized. However, we expect such use will be common, so we envision PinUP rules will require extension to capture the case where an application renames a file and creates a file with the same name as the original.

Web Browser: In our final scenario, we view a PDF with and without the Adobe Acrobat Plugin for Firefox. With the plugin, Firefox opened the file from its cache directory, forking the Adobe `acroread` binary, which subsequently created a file in `/tmp`; the plugin API provided the raw data to `acroread`. Other plugins were observed to behaved similarly. Without the plugin, Firefox created a file in `/tmp`, which was subsequently read by `acroread`. In the former case, no additional rules are necessary, but in the latter case, a PinUP access automation rule is needed to give `acroread` access to Firefox downloaded PDF files. A potential concern is that Firefox can provide data access via the plugin API. However, the mapping of plugins to file types is similar to the PinUP access automation rule notion (no binary hash, however). Integration of application-level access automation and PinUP access automation may be warranted.

7 Related Work

File system protection has received much attention. Originally, all system daemons ran as root. To contain vulnerable daemons, systems administrators run each daemon as a limited user. Unfortunately, some daemons require root privileges. Separation between root-level daemons is provided by reference monitors [2], which allow processes and files labeling (DTE [3]) and manage label transitions (RBAC [11, 10]). SELinux [19], a derivative of Flask [27], is the most widely used reference monitor in commodity systems; however, writing polices describing all system process activities is nontrivial, therefore, targeted mode, which sandboxes specific applications, is most common. The sandbox approach is a common approach for isolating system daemons [6, 20, 18]. Unfortunately, writing sandbox policies requires files to either exist or be isolated to directories. Ko et. al. [16] attempt to overcome this by specifying polices of expected application behavior. Deviations are viewed as anomalies and dealt with accordingly.

User files are more difficult to protect; their creation and access patterns are often unpredictable only understood by the user. File access control lists (ACLs) [9] provide basic flexible user controls; however provide no differentiation between a user's applications. Previous work proposes executing applications in limited environments. This technique, known as sandboxing, uses either pre-specified policies for untrusted applications [1, 13, 15, 21] or privileges reduction for child processes via capabilities [5, 17, 25] or sub-user identities [12, 26]. As discussed in Section 6.1, PinUP provides a more secure default and often requires less overall user effort. Polaris [28] attempts to minimize user effort by providing application-identity profiles; however, the user experience still leaves much to be desired [7]. Finally, SubOS [14] extends the process user identity with a "sub-user" identify specified on an input file. However, SubOS focuses more on controlling process activities resulting from reading a malicious file than protecting user files.

Wichers et. al. [30] were the first to add program access lists (PACLs) to files in an attempt to mitigate attacks by viruses and Trojans. Unfortunately, programs are identified by the path to the executable, which is less secure than the cryptographic digest used by PinUP. Additionally, PACLs are inherited from directories. This approach is not ideal when multiple file types exist in the same directory. Finally, while PACLs was designed and simulated, a usable implementation was never built. LIDS [18] also provides a protection model similar to PinUP is also available in LIDS [18]; however, the interface, which is more commonly used for sandboxing is restricted to systems admin-

istrators. Finally, FileMonster [24] adds application-based protection labels to Microsoft Windows ACLs; however, violations allow users to continue by accepting a pop-up.

8 Conclusion

This paper introduced the PinUP access control overlay system. PinUP enforces application workflows by “pinning” files to the set of applications to which they are intended to be used. Specifically, applications are allowed access to high-value user files only if both system and access overlay permissions allow it. The overlay tags a user’s files with the applications that are allowed to access it. However, as we found, implementing a system meeting this seemingly simple policy turned out to be more complex than one would initially think. We described and evaluated our implementation and found policy could be enforced with very modest performance impact. A consideration of use cases confirmed that the policy adds a significant and important new dimension to user file security.

In this work, we have shown how to extend the protections of current systems to enhance access control; however, more experience with PinUP and real applications is required to develop proper and safe automated access rules. It is these user experiences and design decisions, as well as the experiences resulting from providing these tools to the larger user community that will fully expose the long term value of the PinUP system.

Acknowledgements

We thank Kevin Butler, Patrick Traynor, and the rest of the SIIS lab for their continual feedback on this work.

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[2] J. P. Anderson. Computer security technology planning study. ESDTR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806).

[3] L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of USENIX Security Symposium*, 1995.

[4] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[5] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of USENIX Technical Conference*, 1995.

[6] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *USENIX conference on System administration*, 2000.

[7] A. DeWitt and J. Kuljis. Aligning usability and security: A usability study of polaris. In *Proceedings of the Symposium On Usable Privacy and Security*, July 2006.

[8] W. Enck, S. Rueda, Y. Sreenivasan, J. Schiffman, L. S. Clair, T. Jaeger, and P. McDaniel. Protecting users from ”themselves”. In *Proceedings of ACM CSAW*, 2007.

[9] G. Fernandez and L. Allen. Extending the Unix protection model with access control lists. In *Proceedings of the 1988 USENIX Summer Symposium*, pages 119–132, 1988.

[10] D. Ferraiolo, J. Cugini, and D. R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of Annual Computer Security Application Conference*, 1995.

[11] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of National Computer Security Conference*, 1992.

[12] C. Friberg and A. Held. Support for discretionary role based access control in ACL-oriented operating systems. In *ACM workshop on Role-based access control*, 1997.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applicationos: Confining the wily hacker. In *USENIX Security Symposium*, 1996.

[14] S. Ioannidis, S. Bellovin, and J. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of ACM SIGOPS European workshop*, 2002.

[15] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of USENIX UNIX Security Symposium*, July 1996.

[16] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of ACSAC*, 1994.

[17] N. Lai and T. Gray. Strengthening discresionary access controls to inhibit trojan horses and computer viruses. In *Proceedings of USENIX Summer Symposium*, 1988.

[18] Linux intrusion detection system (LIDS). `http://www.lids.org`, accessed January 2007.

[19] National Security Agency. Security-enhanced Linux (SELinux). `http://www.nsa.gov/selinux`.

[20] Novell. AppArmor application security for linux. `http://www.novell.com/linux/security/apparmor/`.

[21] N. Provos. Improving host security with system call policies. In *Proceedings of USENIX Security Symposium*, 2003.

[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.

[23] V. Samar. Unified login with pluggable authentication modules (PAM). In *Proceedings of ACM CCS*, 1996.

[24] M. Schmid, F. Hill, and A. Gosh. Protecting data from malicious software. In *Proceedings of ACSAC*, 2002.

[25] M. Seaborn. Plash. `http://plash.beasts.org`.

[26] P. Snowberger and D. Thain. Sub-identities: Towards operating system support for distributed system security. Technical Report 2005-18, University of Notre Dame, Department of Computer Science and Engineering, Oct. 2005.

[27] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of USENIX Security Symposium*, Aug. 1999.

[28] M. Stiegler, A. Karp, K.-P. Yee, and M. Miller. Polaris: Virus safe computing for windows xp. Technical Report HPL-2004-221, HP Laboratories Palo Alto, Dec. 2004.

[29] Trusted Computing Group. TCG specification architecture overview: Revision 1.2. `http://www.trustedcomputinggroup.org`, Apr. 2004.

[30] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL’s: An access control list approach to anti-viral security. In *Proceedings of the 13th National Computer Security Conference*, 1990.

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