Design and Implementation of Views: Isolated Perspectives of a File System for Regulatory Compliance

Matthew W. Pagano Zachary N. J. Peterson
The Johns Hopkins University
Baltimore, Maryland, USA
{mpagano,zachary}@cs.jhu.edu

Abstract

We present Views, a file system architecture designed to meet the role-based access control (RBAC) requirement of federal regulations, such as those in HIPAA. Views allows for discrete IO entities, such as users, groups or processes, to have a logically complete but isolated perspective of the file system. Entities may perform IO using the standard system call interface without affecting the views of other entities. Views is designed to be file system independent, extremely easy to use and manage, and flexible in defining isolation and sharing polices. Our implementation of Views is built on ext3cow, which additionally provides versioning capabilities to all entities. Preliminary results show the performance of Views is comparable with other traditional disk file systems.

1 Introduction

The Health Insurance Portability and Accountability Act of 1996 (HIPAA) was written to develop standards for the normalization of individual health records and to encourage the use of electronic records pursuant to these goals. To protect the sensitive data these health records can hold, HIPAA includes provisions that address the way patient medical records are accessed and stored. Specifically, HIPAA mandates the use of role-based access control (RBAC) [7] to protect the privacy of patient information [11]. In traditional file systems, such as ext3 or NTFS, unauthorized modifications to data are commonly prevented through kernel-enforced permissions or access control lists – inodes store metadata about who may access data. However, even in cases where modifications are authorized, unforeseen problems may arise. This is because all users, groups and processes share the same namespace – they have the same "view" of the data. For most file system applications, this is preferential as it easily allows for the sharing of data (*e.g.* applications and configuration files) and avoids wasting disk space with many copies of the same data. The ease with which data can be shared can be detrimental when dealing with sensitive information, *e.g.* accidently copying a health record to the `/tmp` directory may make it accessible to all users. Another obvious drawback to this architecture is that any changes to application data, be they benign or malevolent, will be seen by all users. For example, a trojan horse unintentionally installed by a single user will affect all who execute it.

We present the design and implementation of Views, a system for providing isolated perspectives of a file system designed to meet the RBAC requirements of HIPAA. The primary design goal of Views is to provide every *entity* in the system with its own isolated *view* of the file system. We define an entity to be anything that can perform IO within the file system. An entity could be a user, a group, a process, a family of processes, *etc*. A view is a logically complete perspective of the file system; every entity sees an entire file system hierarchy. Any data modifications made by an entity are, by default, only viewable by that entity. In one extreme example, a user could perform an `rm -rf /` operation without affecting any other user. Views features a robust policy management mechanism, allowing namespaces to be customized for every entity. Providing data isolation at the file system level allows for maximum flexibility of role-based isolation and sharing policies with variable permanency. Views supports temporal policies that dictate how long a view is valid. For example, an entity that is suspected of misbehaving may be analyzed and all modifications it has made easily purged.

Other design goals include minimizing performance and storage overheads. Our implementation of Views uses a copy-on-write mechanism to store only the blocks that have been modified, although copy-on-write is not a requirement for our implementation. Because Views is implemented in the kernel, it provides throughput performance that compares well to an unmodified file system, as we discuss in Section 4.

This work was supported in part by Independent Security Evaluators, Baltimore, Maryland, USA.

Our overall design is largely file system independent in that it relies upon Linux's built-in Extended Attribute interface [2]. This allows Views to be implemented on a wide variety of file systems. Views does not modify any kernel interfaces, requires no special software and is completely transparent to the user. Our design is indiscriminate of backup and restore techniques and allows for easy integration into information life-cycle management (ILM) systems.

We have implemented Views in the ext3cow file system [6]. Ext3cow is a freely-available, open-source versioning file system designed to meet other regulatory compliance requirements, such as fine-grained data authenticity and encryption. Ext3cow-views is available at: `www.ext3cow.com`.

2 Related Work

Views is designed to incorporate three prevailing techniques in current operating systems research: kernel modification to improve security, isolation between IO entities via distinct namespaces, and versioning of files to achieve regulatory compliance. There has been significant research in each of these three areas. Security-Enhanced Linux, or SELinux [4], is a system created by the National Security Agency to introduce a flexible mandatory access control (MAC) architecture to Linux. As with Views, SELinux modifies the Linux kernel by implementing the native extended attribute structure to assign labels to the entities of a file system. Using these labels, SELinux can determine what are permissible interactions between entities based on customizable security policies. Mounts [3] is a system designed by Al Viro that uses the `mount` operation to create isolated filesystem namespaces for each user. Because the isolation in the original design was too restrictive, Mounts has been further developed to include shared mounts when data needs to be shared in a two-way mode, and slave mounts when data needs to be shared in a one-way mode. Users can define the location and nature of each mounted filesystem using pluggable authentication modules and scripts for system startup and user creation. Lastly, there are versioning file systems that provide simple procedures and effective means of revision control [5, 8, 9] and regulatory compliance [6]. These types of systems generally support versioning mechanisms that allow users to easily create and review versions of a given file.

Unlike its predecessors, Views is designed to achieve all aforementioned functionality in an efficient and transparent manner. While SELinux is effective at preventing unwarranted exposure of data within a system, its vast policy configurations can be labor-intensive to administer. This can potentially impede vital functionality if it is mismanaged. Furthermore, the system still has only one filesystem, so no namespace isolation is achieved. If a policy is misconfigured, data will be left exposed as all entities still operate within the same filesystem. Views aims to extend upon this by creating distinct yet complete filesystems for each entity. This will guarantee that an entity cannot access the filesystem of another entity, but that each entity will have full access to the data it needs to perform its functions. While Mounts does achieve a level of filesystem isolation, current documentation on Mounts focuses mainly on isolation based on user accounts. Moreover, the isolation in Mounts is created by calling the mount operation from user space. In contrast, Views is implemented on an open-ended framework that allows isolation for any entity that performs IO on the system, such as users, groups, individual processes, families of processes, *etc*. Views also achieves a finer and more robust degree of isolation at a lower level by creating inode hierarchies in the file system on disk. By using defined policies, the administrator can define a customized view as well as set a time-to-live for any entity, so views are as temporal or permanent as the administrator chooses. Lastly, Views is designed to be independent of any versioning model and capable of being implemented in a wide variety of system designs.

3 Views Architecture

Views is built on top of the ext3cow file system [6]. Ext3cow is a variant of the ext3 file system [1] that provides file versioning accessed through a time-shifting interface, authenticated encryption and other technologies designed to meet regulatory compliance. Ext3cow is good platform for a Views implementation as it provides versioning and supports Linux's Extended Attributes API. Extended Attributes is a file system facility available in ext3cow (and other file systems) that provides a means to associate additional metadata with files and directories. Extended Attributes, often abbreviated as xattr, have been used for implementing access control lists [2] and security tags in SELinux [4].

Figure 1 shows an example of isolated views for two users. Both users share a view of the `bin/` directory that is part of the master view. The `matt` user creates a directory `foo/` that only exists in his view. Similarly, the `zachary` user creates a view-specific directory named `bar/`. Both users create a file with the name `test.txt`, but while the name appears in the both views, they represent different files with different contents.

3.1 File Views

In most file systems, every file has a single, corresponding inode that contains that file's metadata, including

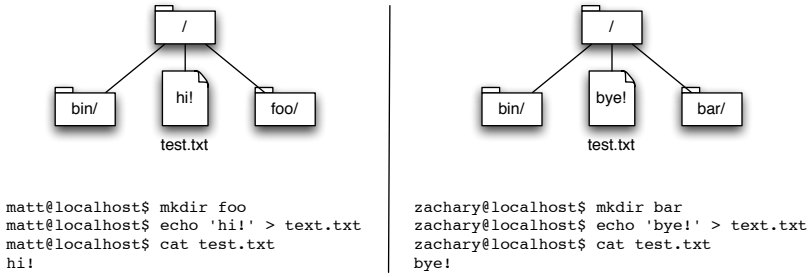

Figure 1: An example of Views' isolation model.

modification time, permissions, and pointers to the file's data blocks. If the file system supports Extended Attributes, the inode also contains space or a block pointer in which to store Extended Attribute information – typically a name-value pair. When a Views file system is created by the root user, every original file is represented by a single inode. We call this base file system the *master view* and each inode is a *master inode*. The master inode may be changed over time, for system maintenance for example (See Section 3.3), but it represents the basis from which all additional views are created.

When an entity makes an initial modification to a master view file, a new view for that entity is created. To create a new view, a new *entity inode* is allocated, the metadata from the master inode is copied into the new entity inode, and the entity inode is linked to the master inode with an extended attribute (see Figure 2). Views for a file are represented by entity-inode pairs stored as extended attributes. When the file is modified by an entity, all modifications are recorded in the corresponding entity inode, isolating the changes to only that view. We improve upon ext3cow's copy-on-write model to achieve Views. All data modifications made to an entity inode are copy-on-written: a new block is allocated for the modification and old block is preserved as part of the master view. When an entity modifies a data block, the new block is stored only in that entity's inode, and therefore viewable only by that entity.

When an entity creates a file, only a single inode is allocated. Because the name and inode are not accessible by any other entity and the file is not part of the master view, no master inode or extended attribute is created or needed.

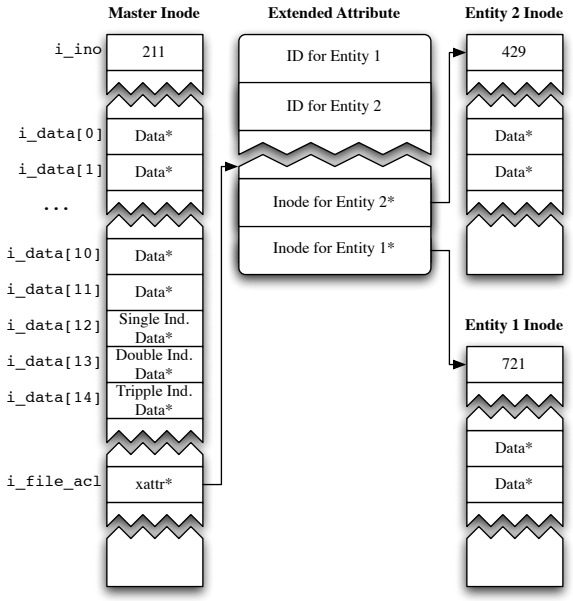

Figure 2: Architecture of Views using extended attributes.

3.2 Directories and Naming

In addition to having isolated data modifications, entities may also modify the file system name space in isolation. This is easily accomplished in Views as directories are simply inodes whose data are directory entries. Name modifications (addition, removal or change of a name) by an entity to a directory that has a master inode (*i.e.* part of the master view) will create an entity inode and extended attribute entry, similar to what is shown in Figure 2. Likewise, any name modification to a directory created by an entity will result in only a single directory inode.

To perform a lookup of a name in Views, the file system reads the inode of the parent directory. The inode may be a master inode if the parent directory is part of the master view, or an entity inode if the parent directory is part of an entity view. If the parent directory is an entity inode, a normal lookup is performed. If the parent directory is a master inode, the file system searches the extended attributes for an identifier that matches the

entity performing the lookup. If an identifier is found, the entity inode is returned. If no identifier is found and the operation is read-only the master inode of the corresponding name is returned, otherwise the file system returns a file not found error.

Because the same name can exist in different views but represent different data, modifications to the directory entry cache (dcache) are necessary. This is achieved by writing the ID of the entity that owned a given directory entry (dentry) to its filesystem-specific data parameter, `d_fsdata`. The dentry cache is modified such that when a cached dentry being analyzed belongs to a ext3cow-views partition, a comparison is performed between the ID of the entity searching the cache and the `d_fsdata` parameter of the cached dentry. Our modified dentry cache only returns a valid dentry if the ID matches and NULL otherwise.

3.3 Policies of Views

The root user is a special entity in Views. In particular, the root user controls the master view. By default, the root user sees only the master view, but has the ability, through an additional API, to see all entity views. When the root user adds a file or directory, it automatically becomes part of the master view for all future entities. This allows the root user, for example, to perform system-wide modifications and maintenance. It is important to note that changes to the master view do not affect an entity's already existing views. Said another way, if an entity has created a view for a file and the master view changes above it, the entity's view is unchanged. Only future views created on that file will see the master view changes. This is to ensure view consistency for entities.

3.4 Views and Versioning

In addition to role-based access controls, HIPAA requires an auditable trail of changes that is accessible in real time. This requirement mandates the use of versioning in a file system. Views fits well into ext3cow's versioning model. In ext3cow, versions of a file are implemented by chaining inodes together, where each inode represents a version. The file system traverses the inode chain to generate a point-in-time view of a file. Implementing Views in ext3cow allows an entity to have an isolated view of a file as well as enjoy versioning capabilities. Every master and entity inode is capable of maintaining a version chain without affecting any other entities' views.

The addition, removal and modification of names over time is also supported by ext3cow-views. Names in ext3cow are scoped to times using additional metadata in the directory entry structure. Every directory entry contains and birth and death time-stamp, defining a period of time for which that name is valid. This allows file names to be added, removed and re-added over time while maintaining past data. Should an entity add or remove a name, a new entity inode is created and the corresponding directory entry is modified. Because Views provides individual entity directory inodes, all names scope properly, even if they used by other entities or re-used by the same entity.

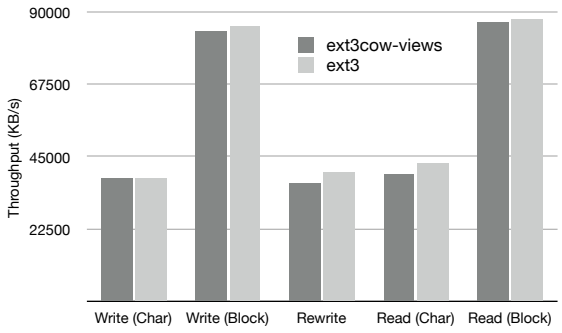

Figure 3: The throughput results from the Bonnie++ benchmark suite.

4 Preliminary Results

We measure the impact Views has on file system performance by comparing the IO throughput of ext3cow-views with that of ext3. We use the Bonnie++, a benchmarking suite used to quantify five aspects of file system performance based on observed I/O bottlenecks in a UNIX-based file system. Bonnie++ performs I/O on large files (for our experiment, two 1-GB files) to ensure I/O requests are not served out of the disk's cache. The five tests Bonnie++ performs are: (1) each file is written sequentially by character, (2) each file is written sequentially by block, (3) the files are sequentially read and rewritten, (4) the files are read sequentially by character, and (5) the files are read sequentially by block. Figure 3 presents the throughput results for each Bonnie++ test, which show that our implementation of Views has little to no impact on the throughput performance of the file system.

5 Future Work

Entities. Because Views is currently a proof-of-concept, only users and groups are supported entities. We hope to develop additional classes of entities. Processes pose a unique challenge as they are difficult to accurately identify over time. Process numbers change with every execution and even the name of the binary may change. One

possible solution is identify a process or group of processes by a cryptographic hash of their contents or portion of their contents.

Once processes are a supported entity, Views can be extended to provide isolation for any application. This will be extremely useful in examining the behavior of an application without having to commit the application's modifications to the master view. This approach is similar to work done by Sun *et al.* [10]. However, because Views transparently provides each entity with its own complete view of the file system, we can avoid errors that might surface from transferring the properties of the isolated SEE environment to the host system. Views still provides the same degree of low-level isolation at minimal overhead, but Views can continue the file system isolation permanently if desired. This would allow a user to permanently use two different versions of `gcc`, for example, without conflicting with any other view of the file system.

Furthermore, enabling processes as a supported entity will facilitate an application checkpoint and recovery solution. By setting up the application's process or family of processes as an entity within Views, all of the application's data will be saved to that entity's view, which can optionally be saved and exported to a different view.

Policies. The Views framework was designed to be flexible for implementing and enforcing varying access control policies. Views' current policies and policy management system are rudimentary. The general policy Views enforces is: the master view is read-only accessible by all entities; any additional data created by an entity is accessible only by that entity.

Future work for Views will include an configuration mechanism that will allow administrators to merge views and resolve any subsequent conflicts based on certain criteria. For example, an administrator might wish to merge two views and in the event of a duplicate file, keep only the more recently modified file. Moreover, future work will include configurations to allow data to be shared among certain entities, as well as the access controls to govern this shared data. The access controls in this case will be especially important in ensuring that any role-based access controls that are enabled using Views are not bypassed when configuring shared data.

Views is designed to enable administrators to more granularly define the properties of each view. For example, the administrator might wish to generate a view for an entity that is only valid for a certain period of time, after which point the view should be merged with another view or discarded altogether. The administrator might also wish to define which, if any, views can modify the master view such that all future views have access to those modifications. These types of policies will be made available to the administrator using a defined API.

6 Conclusions

We have introduced Views, an open-source file system extension that uses Linux's Extended Attribute API to provide isolated, complete and configurable views of a file system namespace for role-based access control compliance. In this system, any entity that can perform IO is able to modify the file system without affecting other entities in the system – an entity is only able to see the changes it makes. Views is designed to support flexible isolation policies, allowing the permanence of file system modifications to be defined. Preliminary experimental results show that ext3cow-views performs comparably to an unmodified ext3 file system. Our implementation of Views was built on ext3cow, an open-source versioning file system, and is available at: `www.ext3cow.com`.

References

[1] CARD, R., TS'O, T. Y., AND TWEEDIE, S. Design and implementation of the second extended file system. In *Proceedings of the Amsterdam Linux Conference* (1994).

[2] GRÜNBACHER, A. POSIX access control lists on Linux. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2003), pp. 259–272.

[3] HALLYN, S. E., AND PAI, R. Applying mount namespaces. http://www.ibm.com/developerworks/linux/library/l-mount-namespaces.html, 2007.

[4] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2001), pp. 29–42.

[5] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2004), pp. 115–128.

[6] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage 1*, 2 (2005), 190–212.

[7] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *IEEE Computer 29*, 2 (February 1996), 38–47.

[8] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (December 1999), pp. 110–123.

[9] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 43–58.

[10] SUN, W., LIANG, Z., SEKAR, R., AND VENKATAKRISHNAN, V. N. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Network and Distributed System Security Symposium* (2005), pp. 265–278.

[11] UNITED STATES CONGRESS. The Health Insurance Portability and Accountability Act (HIPAA) Security Rule. 45 CFR Parts 160, 162 and 164, 1996.