Journal version of USENIX Annual Technical Conference 2006 paper

Privacy Analysis & Enhancements for Data Sharing in *nix Systems

Aameek Singh[†] **Ling Liu** **Mustaque Ahamad**

Georgia Institute of Technology

Abstract

Linux and its various flavors (together called *nix) are growing in mainstream popularity and many enterprise infrastructures now are based on *nix platforms. An important component of these systems is the ingrained multi-user support that lets users share data with each other.

In this paper, we first analyze *nix systems and identify an urgent need for better privacy support in their data sharing mechanisms. In one of our studies it was possible to access over 84 GB of private data at one organization of 836 users, including over 300,000 emails and 579 passwords to financial and other private services websites. The most surprising aspect was the extremely low level of sophistication of the attack. The attack uses no technical vulnerabilities, rather inadequacies of *nix access control combined with user/application's privacy-indifferent behavior.

We present two solutions to address this problem: (1) an administrative auditing tool which can alert administrators and users when their private data is at risk, and (2) a new View Based Access Control (VBAC) model which provides stronger and yet convenient privacy support. We also describe a filesystem based implementation and performance analysis of VBAC. A unique property of our implementation is that individual users can choose to switch on/off the VBAC access control model. Our evaluations with three well-known filesystem benchmarks show very little overhead of using VBAC. For example, our implementation resulted in only 3% total overhead with the popular Andrew Benchmark.

1 Introduction

With increasing popularity of open source operating systems like Linux, many enterprises are opting to set up their intranets using such *nix systems. The market research firm IDC expects linux server sales to hit US $9.1 billion by 2008 and that linux servers will comprise 25.7% of total server units shipped in 2008 [1]. Consequently, many big companies like IBM, Dell actively support the open source *nix movement.

One of the important features of *nix systems is their ingrained multi-user support. These operating systems are designed for simultaneous multiple users and provide seamless mechanisms to share data between different users. For example, user *alice* can set up appropriate access "permissions" on the data she wants to share with her group *students* by executing a simple `chmod` command [29]. These access permissions are dictated by the *nix *access control model*.

In this paper, we take a critical look at the *nix access control model. Our objective is to analyze the **privacy** support in its data sharing mechanisms. For example, how does the system assist a user to share data only with desired users and prevent private information from being leaked to unauthorized users? In order to successfully do this, we also need to look at the **convenience** of using *nix data sharing mechanisms in typical situations. This is due to the fact that lack of convenience typically leads to users compromising (intentionally or mistakenly) their security requirements to conveniently fit the specifications of the underlying access control model.

Please note that we use the seemingly oxymoronic phrase "*private sharing*" to indicate the desire of sharing data only with a select set of authorized users.

As part of our analysis, we looked at how users use the access control for their data sharing needs in practice. We conducted experiments at two *nix installations of a few hundred computer-literate users each. Surprisingly, we found that large chunks of private data was accessible to unauthorized users. In many cases, the user's definition of an "authorized user" does not match the underlying system's definition, that leads to such a breach. This observation is best exemplified in the following scenario. Many users attempt to privately share data by using execute-only permissions[1] for their home directories. This prevents other users from listing the contents of the directory, but any user who knows the name of a subdirectory can `cd` into it. The data owner *authorizes* some users by explicitly giving them the names of the subdirectory through out-of-band mechanisms like personal communication or email. However, this authorization is not the same as the system's authorization. From the underlying system perspective, it is assumed that any user who issues the command with the right directory name is authorized. Thus, users who simply guess the subdirectory name can also access the data. Along with that, setting execute-only permissions on the home directory to share *one* subdirectory, puts all other subdirectories (sibling to the shared directory) also at risk, which if not protected appropriately, can be accessed.

We were able to effectively exploit these shortcomings in our studies. At one organization of 836 users, over 84 GB of data was accessible, including more than 300,000 emails and 579 passwords to financial websites like *bankofamerica.com* and other private websites like medical insurance records. Importantly, the attack does not always need to *guess* directory

[†]Supported by IBM PhD Fellowship 2005-06

[1]Also represented as $--x$ or numeric '1'. Complete *nix access permissions are discussed later in Section-2

names, but can find actual names from unprotected command history files (*.history, .bash_history*) or standard application directory names (*.mozilla*). The reason for this surprisingly large privacy breach without exploiting technical vulnerabilities like buffer overflows or gaining elevated privileges, is the combination of lack of system support and user or even applications' privacy-indifferent behavior either mistakenly or for lack of anything better. Also, as we discuss later in Section-2, even for an extremely privacy-conscious user with all available tools, it is tough to protect private data in many situations.

The attack described in this paper is a form of an **insider** attack, in which the attacker is inside the organization. The attacker could be a disgruntled employee, contractor or simply a curious employee trying to access the salaries chart in the boss's home directory. According to a recent study by the US Secret Service and CERT [28], such attacks are on a rise with 29% of the surveyed companies reporting[2] having experienced an insider attack in the past year [19]. Also, in complete congruence to our attack, the report finds that:

> "`Most incidents were not technically sophisticated or complex`"

We propose two solutions to address this problem:

- A Privacy Auditing Tool that analyzes the "*privacy health*" of an enterprise. Such auditing can be combined with the periodic virus scans and other security audits regularly employed by enterprises.
- A new access control model which provides stronger privacy protection and yet convenient data sharing mechanisms. The **View-Based Access Control (VBAC)** model allows data owners' to define views of their data that can be seen by other users, while protecting other data from unauthorized users. We provide a file system implementation of VBAC and show that the overheads incurred by it are minimal.

To summarize, this paper makes four unique **contributions**:

1. **Privacy Analysis of *nix Systems**: To the best of our knowledge, this is the first work that specifically analyzes the *nix access control from the privacy perspective. We present results on how users share their data and how much private information can be accessed by unauthorized users in real *nix installations.
2. **Data Sharing Mantras**: We identify a number of useful mantras that allow for private and convenient data sharing in multi-user environments. We also analyze existing access control tools like *nix permissions model, POSIX ACLs [26], `umask` [29] on these mantras.
3. **Privacy Enhancements**: We present two approaches that can enhance the privacy characteristics of *nix systems. The first approach is a conservative approach that only measures the privacy health of the system and alerts users of potential threats. The second approach is more proactive and utilizes View-Based Access Control to provide safer data sharing. We also describe an implementation of VBAC and its performance analysis.
4. **User Education**: The privacy analysis and data sharing mantras presented in this paper help us in devising various privacy enhancement approaches. More importantly they bring into light a massive threat to user privacy in current systems, which can be exploited overnight by a few hundred lines of code. This work thus contributes directly to increasing user awareness and education in protecting private data.

The rest of the paper is organized as follows. In Section-2, we discuss various issues that lead to privacy breaches. We also present case studies at two *nix installations which demonstrate these breaches. In Section-3, we describe a number of useful privacy mantras and analyze existing tools. Section-4 gives an overview of the two privacy enhancement approaches. Section-5 presents a detailed discussion of the VBAC access control model including its design goals, implementation and performance evaluation. We describe related work in Section-6 and conclude in Section-7.

2 Data Privacy: Vulnerability Analysis

In this section, we discuss various scenarios that lead to privacy breaches and present the results of our case studies. This analysis will provide us insights into desirable privacy and convenience characteristics. We start off with a description of *nix access control, to establish a common reference model for subsequent discussions.

2.1 *nix Access Control

The *nix access control primarily follows from the original UNIX access control model [34, 33]. In this discretionary access model, each file system object (file, directory, links) has an associated owner who controls the access to that object. This access can be granted to three kinds of users: (1) *owner*, (2) *group*, and (3) *others*. The *owner* is the object owner, the *group* is a set of users to which the owner belongs (for example, a user group for students, faculty) and *others* are all other users (all users except the owner and the group). Also, note that the *owner* permissions take precedence over *group*, that is, the owner will have access as dictated by the *owner* permission bits, even though he/she also belongs to the *group*. Further, the granted access is of three types:

- **Read:** For a file, this means that a user can read a file. For a directory, this means that a user can list its contents using `ls` [29]. For links, the permissions are for the object that are pointed-to by the link and the link's permissions itself are not used. The read permission is represented by a 'r' or a numeric value of 4.
- **Write:** For a file, the write permission allows a user to write to it. For a directory, it allows a user to create, remove or rename directory contents. For links, permissions are again for the pointed-to object. The write permission is represented by a 'w' or a numeric value of 2.

[2] It is believed that such attacks are usually much under-reported for lack of concrete evidence or fear of negative publicity [40]

	Owner	Group	Others
Read	X	X	
Write	X		
eXecute	X	X	X

Figure 1: Example Directory Permissions

- **eXecute:** For a file, the execute permissions allows running the file as a program (for example, a shell script or a compiled C program). For directories, it allows users to traverse that directory and if the directory contents have appropriate permissions, the user can then access those contents. For example, to access directory `grand-child` with path `dir/child/grand-child`, both `dir` and `child` need to have execute permissions. The execute permission is represented by a 'x' or a numeric value of 1.

The permissions bits are listed in the "*owner, group, others*" order. Figure-1 shows an example directory with its 3x3 access control matrix. It will be listed as `rwx r-x --x` or in numeric terms 751 (7 = `r(4)+w(2)+x(1)` and so on). For that directory, the owner can list its contents, create, remove, or rename its children and can traverse into that directory. The users belonging to the group of the owner can list its contents and traverse into the directory but not modify its contents. The *other* users can only traverse into the directory and not list or modify its contents.

To improve the granularity of user permissions, lately POSIX Access Control Lists (ACLs) [26] have been introduced into many *nix variants. They allow setting permissions at individual user levels as opposed to *group* or *others*. However, the {*owner, group, others*} is still the most dominant paradigm and many *nix installations do not even have ACLs enabled. We will analyze the ACLs later in Section-3.

Additionally, a user-specific `umask` [29] setting is used to decide permissions for a newly created file. The command specifies a mask for permissions to be **disallowed** for the owner, group or other users. For example, to always create a new file with `rwx` permissions for *owner* and no permissions for *group* or *others* (that is, permission 700), the command "`umask 077`" can be stored in the user initialization file (like *.profile*).

It is possible to override the *nix access control at a file system level. For example, the distributed Andrew File System (AFS) [36] has a different set of permissions settings and thus data residing on that file system can use its settings. In fact our privacy enhancing VBAC approach also modifies the access control at that layer. We discuss this later in Section-5.

2.2 Privacy Breaches

In this section, we discuss various privacy breaches that occur in *nix systems. We also present results of our study at two *nix installations that demonstrate the insufficiency of current access control mechanisms. The discussion below primarily explores the popular {*owner, group, others*} *nix paradigm. The advanced mechanisms like ACLs enhance privacy in only a few cases and we will demonstrate a clear need of a new, more complete solution.

2.2.1 Selective Data Sharing

The first kind of privacy breach occurs due to the need to "*selectively*" share data. The selectivity can be of two kinds:

- **Data Selectivity**: Data selectivity is when a user wants to share only a few (say one) of the subdirectories in the home directory. So, an authorized user is allowed to access only the shared subdirectory, but not any of the sibling directories. In order to do this correctly, the owner needs to follow two steps - (a) set appropriate permissions to the shared subdirectory (at least execute permissions on the entire path to the subdirectory and the sharing permissions on the subdirectory), and (b) *remove permissions from the sibling subdirectories*. The second step is unintuitive, since the user needs to act on secondary objects that are not the focus of the transaction. Also, any new file being created needs to be protected.
- **User Selectivity**: In many situations, users need to share data with an adhoc set of users that do not belong to a single user group or are only a subset of a user group, and are not the entire *others* set. In this situation, the permissions for *group* or *others* are not sufficient. Also, creating a new user group requires administrative assistance which is not feasible in all cases.

In order to do selective data sharing, currently owners mostly use execute-only permissions on the home directories. The perception is that since users can not list the contents of the directory, they cannot go any further than traversing into the home directory unless they know the exact name of the subdirectory. Now, the owner can authorize desired users by giving them the names of the appropriate subdirectories. Those *authorized* users can traverse into the home directory and then use the subdirectory name to `cd` into it (without having to list the contents of the home directory). From data selectivity perspective, it is assumed that they cannot access the rest of the contents and from user selectivity perspective, unauthorized users cannot access any contents.

However, the underlying system cannot distinguish between such authorized or unauthorized users. Any user who can **guess** the subdirectory name can actually access the data. For an attacker inside the organization, this is not a herculean task. For example, for a computer science graduate school, it is highly likely that users will have directories named *research*, *classes* or *thesis*. An easy way of creating such a list of names is by collecting names from users that actually have *read* permissions on the home directories. Within the context of a single organization, or in general human psychology, it is likely that many users have similar directory names. This is essentially a form of *social engineering* [21] in which users and not systems are manipulated to reveal confidential information[3]. Of course, one simple solution is to

[3]A well-know hacker, Kevin Mitnick said "... *social engineering was extremely effective in reaching my goals without resorting to using a technical exploit ...*" [39]

use cryptic directory names unlikely to be guessed (security-by-obfuscation). The problem then becomes similar to the fundamental problem with passwords. Keeping commonly used passwords means that they can be guessed and cryptic passwords are tough to remember. However, the problem is more severe, since while a single password is enough to access thousands of files, a user cannot be expected to keep cryptic names for tens or hundreds of files and directories.

Secondly, many times directory names do not need to be guessed at all. The names can be extracted from **history** files (like *.history* or *.bash_history*), that contain the commands last executed by the owner, like `cd`, which will include real directory names. In fact, in our experiments we found around 20-30% of all users had readable history files and around 40% of the total leaked data was obtained from the directory names extracted from these history files.

Thirdly, it is not always user created directories that leak information. Many **applications** use standard directory names and fail to protect critical information. For example, the famous Mozilla web browser [10] stores the profile directory in `~/.mozilla` and had that directory world-readable [32] in many cases, till as late as 2003 (the Mozilla project was initiated in 1994). Many *nix installations with the browser installed before that have this vulnerability and we were able to obtain around 575 password to financial and private websites (because users saved passwords without encrypting them). In addition, their browser caches, bookmarks, cookies and histories were also available. The browser Opera [12] also has a similar vulnerability, though to a lesser extent. While it can be argued that it is the responsibility of application developers to ensure that this does not happen, we believe that the underlying system can assist users and applications in a more proactive manner.

The POSIX ACLs [26], if used help in achieving only user selectivity. They do not address the data selectivity requirements or prevent leaking of application data.

2.2.2 Metadata Privacy

So far, we have only talked about the privacy breach for file *data*. However, there are many situations in which users are interested in protecting even the metadata of the files. The metadata contains information like ownership, access time, updation time, creation time and file sizes. There are scenarios where a user might obtain confidential information by just looking at the metadata. For example, an employee might be interested in knowing how big is his annual review letter or did the boss update it after the argument he had with her?

The *nix access control does not provide good metadata privacy. Even if users only have execute permissions on a directory, as long as they can guess the name of the contained file, its metadata can be accessed even if the file itself does not have any read, write or execute permissions on it. Thus, if a user has to share even a single file/directory within the home directory (thus, requiring atleast execute permissions), all other files contained in the home directory have lost their metadata privacy.

Of course, a solution is to put such files in a separate directory and protect them. However, in many cases these files might be accessed by standardized applications making it infeasible to move (for example, how to protect metadata privacy of shell initialization files (*.profile*), or history files, which are always created in the home directory). Also, from our experience, many users like to keep their active files in the home directory itself and find it inconvenient to have a deep directory structure.

Again, lack of convenient support from the system leads to privacy breaches, in this case leaking file metadata.

2.2.3 Data Sharing Convenience

User convenience is an important feature of an access control implementation. If users find it tough to implement their security requirements, they are likely to compromise the security requirements to easily fit the underlying access control model. This can be seen as one of the reasons why encryption file systems are not in widespread use, even though they guarantee maximum security.

From our analysis of the *nix access control, along with some of the issues discussed earlier, we found the following two data sharing scenarios in which there is no convenient support for privacy.

- **Sharing a Deep-Rooted Directory**: For a user to share a directory that is multiple levels in depth from the home directory, there needs to be at least execute permissions on all directories in the path. This in itself (a) leaks the path information, (b) puts sibling directories at risk and (c) leaks metadata information for sibling directories. In order to prevent this, since most operating systems do not allow hard links to directories anymore, a user would have to create a new copy of the data. And since users are more careless with permissions for deep rooted directories (they protect a higher level directory and that automatically protects children directories), a copy of such a directory could have privacy-compromising permissions.
- **Representation of Shared Data**: In many circumstances the way one user represents data might not be the most suitable way for another user. For example, while an employee might keep his resume in a directory named `job-search`, it is clearly not the most apt name to share with his boss. The employee might want her to see the directory simply as `CV`. Changing the name to meet the needs of other users is not an ideal solution. This again shows the lack of adequate system support for private and convenient data sharing.

It is important to recognize that even an extremely privacy-conscious user can not protect data at all times. Exhaustive user efforts to maintain appropriate permissions on all user and **application created** data will still be insufficient to protect metadata privacy or allow private sharing of deep rooted directories with user-specific representation.

2.3 Case Studies

As part of our study, we conducted experiments at two geographically and organizationally distinct *nix installations.

Users at both installations (CS graduate schools) are highly computer literate and can be expected to be familiar with all available access control tools.

For our analysis, we consider the following data to be private:

- All user emails are considered private.
- All data under an execute-only home directory is considered private.
- Browser profile data (including saved passwords, caches, browsing history, cookies) is considered private.

The second assumption above merits further justification. It can be argued that not every subdirectory under an execute-only home directory is meant to be private (for example, a directory named *public*). However, we believe our definition to be a practical one. The ideal measurement would require active user participation in the study who, by anecdotal experience, once told of the threat immediately removed all permissions from their home directories. Also, the semantics of the execute-only permission set dictate that any user other than the owner cannot list the contents of the directory and since the owner never *broadcasts* the names of the shared directories, an unauthorized user *should not* be able to access that data. And since we do not include in our measurements any obviously-private data from home directories of users with *read* permissions (for example, world-readable directories named `personal`, or `private`), we believe the two effects to approximately cancel out.

2.3.1 Modus Operandi

Next, we describe the design of our attack[4] that scans user directories and measures the amount of private data accessible to unauthorized users. This discussion is also important since the design is eventually used to develop an auditing tool discussed later.

The attack works in multiple phases. The first step is to obtain directory name lists which can be tried against users with execute-only home directories. Three strategies are used to obtain these lists:

- *Static Lists*: These are manually entered names of directories likely to be found in the context of the organizations - CS graduate schools. For example, "*research*", "*classes*", "*papers*", "*private*" and their variants in case ("*Research*") or abbreviations ("*pvt*").
- *Global Lists*: These lists are generated by obtaining the directory names from home directories of users that have *read* permissions.
- *History Lists*: These are user specific lists generated by parsing users' history files, if readable. We used a simple mechanism, parsing only `cd` commands with directory names. It is possible to do more by parsing text editor commands (like `vim`) or copy/move commands.

[4]Due to the sensitive nature of the task (measuring *private* content) we took precautions to ensure that our study does **not** violate user privacy. These details are included in Appendix-A

In the next step the tool starts a multi-threaded scanning operation that attempts to scan each user directory. For users with **no** permissions, no scanning is possible. For users with **read** permissions, as discussed earlier, since there is no precise way of guessing which data would be private, we only measure email and browser profile statistics. Finally, for users with **execute-only** permissions, along with email and browser profile statistics, we also attempt to extract as much data as possible using the directory name lists prepared in the first step.

Evaluating Email Statistics

This is done by attempting to read data from standard mailbox names - "*mail*", "*Mail*", "*mbox*" in the home directory and the mail inboxes in */var/mail/userName*. A `grep` [29] like tool is used to measure (a) number of readable emails, (b) number of times the word "*password*" or its variants appeared in the emails.

Evaluating Execute-only Data Statistics

For users with execute-only permissions on the home directory, the scanner uses the combination of static, global and the user's history lists to access possible subdirectories. Double counting is avoided by ensuring that a name appearing in more than one list is accounted for only once and by not traversing any symbolic links. While scanning the files, counts are obtained for the total number of files and the total size of the data that could be accessed.

Evaluating Browser Statistics

The mozilla browser [10] stores user profiles in the `~/.mozilla` directory. This directory used to be world-readable till as late as 2003 when the bug was corrected [32]. Within that profile directory, there are subdirectories for each profile that has been used by that user. The default profile is usually named "*default*" or "*Default User*". So even in case the .mozilla directory had execute-only permissions, it is possible to access default profile directories (unless a user specifically removed permissions). Within the profile directory, there is another directory with a randomized name (for security purposes!) ending in "*.slt*"[5]. Since the parent directories had *read* permissions, the randomization provides no security and the name is visible. Within this directory, the following files exist and (with this bug) were readable:

- **Password Database:** A file with the name of type "*12345678.s*". This contains user logins and passwords saved by mozilla when the user chooses to save them. Ideally, users should use a cryptographic master key to encrypt these passwords, but as our results will show many users do not encrypt their passwords. For such cases, mozilla stores the passwords in a base-64 encoding (indicated by the line starting with a ~ in the passwords file), which can be trivially decoded to get plaintext passwords.

[5]Please see [9] for complete profile directory contents and their location.

- **Cookies:** The `cookies.txt` file contains all browser cookies. Many websites including popular email services like Gmail [4], Hotmail [8] allow users to automatically login by keeping their usernames and passwords (encrypted) in the cookies file. Hijacking this cookie can allow a malicious user to login into these accounts. For many other cookies related attacks, see [38].
- **Cache:** This is a subdirectory that contains the cached web pages visited by the user.
- **History Database:** Web surfing history, which many sophisticated viruses and spyware invest resources to collect, are also readable.
- **Forms Database:** Mozilla allows users to save their form data, stored in a file of type "*23456789.w*", that can be automatically filled. This could include credit card numbers, social security numbers and other potentially sensitive information. Here again, users should use a master key to encrypt this information.

2.3.2 Results

The complete characteristics of the two organizations are shown in Table-1. Both organizations are computer science graduate schools at two different geographical locations within the United States. At both the organizations, a significant number of users (68% and 77%) used execute-only permissions on their home directories.

Org.	# Users	# ReadX	# NoPerms	# X-only
Org-1	836	198	54	573
Org-2	768	136	39	593

Table 1: Case Study Organization Characteristics. # ReadX is the number of users with read and execute permissions to their home directories, # NoPerms are users with no permissions and # X-only are the users with only execute permissions

Table-2 lists the amount of data extracted from execute-only home directories at Org-1 and 2.

Org.	# Hit Users	# Hits	# Files	Data Size
Org-1	462	2409	983086	82 GB
Org-2	380	911	364932	25 GB

Table 2: Data extracted from X-only home directory permissions. # Hit Users is the number of users that leaked private information. # Hits is the total number of directory name hits against all X-only users. # Files is the number of leaked files and Data-Size is the total size of those files

As can be seen, a large fraction of users indeed leaked private information - 55% and 49% of total users respectively. Recall that we do not extract any data from users with *read* permissions on their home directories; so a more useful number is the fraction of X-only users that revealed private information. That number is 80% and 64% respectively. Also, on an average, 2127 files and 177 MB of data is leaked in the first organization for each X-only user and 960 files and 65 MB of data is leaked in the second organization. A partial reason for the lower numbers in the second organization could be the fewer number of users with *read* permissions, which would have impacted the global name lists creation. Overall, we believe this to be a very significant privacy breach.

As mentioned earlier, many times the names of the subdirectories do not need to be guessed and can be obtained from the history files in the user home directories. Table-3 lists the success rate of the attack in exploiting history files. As it shows, around 40% of X-only users had readable history files which led to 40-50% of total leaked data in size.

Org.	# History Hits	# Files	Data Size
Org-1	253	561254	35 GB
Org-2	237	155826	14 GB

Table 3: Exploiting History Files. # History Hits is the number of users with readable history files. # Files is the number of private files leaked due to directory names obtained from history files and Data-Size is the size of the leaked data

Email Statistics

Table-4 presents the results of the email data extracted from users in both organizations. Recall that this data is obtained for both X-only users and the users with *read* permissions on their home directories.

Org.	# Folders	# Emails	Size	# *Password*
Org-1	2509	315919	4.2 GB	6352
Org-2	505	38206	120 MB	237

Table 4: Email Statistics. # Folders is the number of leaked email folders. # Emails is the total number of leaked emails. Size is the size of leaked data and # *Password* is the number of times the word "*password*" or its variants appeared in the emails

As can be seen, a large number of emails are accessible to unauthorized users (especially at Org-1). Also, the number of times the word "password" or its variants appear in these emails is alarming. Even though we understand that some of these occurrences might not be accompanied by actual passwords, by personal experience, distributing passwords via emails is by no means an uncommon event.

Browser Statistics

The second organization did not have the mozilla vulnerability since they had a more recent version of the browser installed, by which time the bug had been corrected. So the results shown in Table-5 have been obtained only from the first organization. Looking at the results, the amount of accessible private information is enormous. Figure-2 contains a sample of the websites that had their passwords extractable and clearly most of these websites are extremely sensitive and a privacy breach of this sort is completely unacceptable.

As an interesting side statistic, out of the 579 passwords, only 308 were unique passwords, that is, 36% of passwords were repeated. This indicates that users repeat their passwords, a common habit by anecdotal experience. Also as seen from Figure-2, some obtained passwords were for accounts in other institutions and a few of them are likely to be *nix systems.

# Users with accessible `.mozilla`	311
# Users with readable password DB	149
# Passwords Retrievable	579
# Users with readable cookies DB	207
# Cookies Retrievable	19456
# Users with accessible caches	233
# Cached Entries	20907
# Users with readable browsing histories	256
# URLs in History	130,503

Table 5: Browser Statistics at Org-1

Financial Websites	**Personal Websites**
www.paypal.com	adultfriendfinder.com
www.ameritrade.com	www.hthstudents.com
www.bankofamerica.com	www.icers911.org
Email Accounts	**Other Institutions**
mail.lycos.com	cvpr.cs.toronto.edu
my.screenname.aol.com	e8.cvl.iis.u-tokyo.ac.jp
webmail.bellsouth.net	systems.cs.colorado.edu

Figure 2: Sample accounts with retrievable passwords

Thus, it is conceivable that this password extraction can be used to **expand to other *nix installations** and thus be much more severe in scope than a single installation.

Miscellaneous Statistics

Among few other applications at Org-1, 17 users had their Opera [12] browser's cookies file readable and 497 users had their email address books, used by the Pine email client [13] and stored in `~/.addressbook` readable. 18,308 email addresses could be obtained from these address books which can be potentially used for highly targeted spam!

2.4 Attack Severity

It is important to highlight the severity of this attack:

- **Low Technical Sophistication**: The attack is extremely low-tech; the commands used in a manual attack would be `cd`, `ls` and such. This aspect makes the threat significantly more dangerous than most other vulnerabilities.
- **Low Detection Possibility**: A version of the attack that targets only a few users a day and thus keeps overall disk activity normal has a very low probability of detection. Typical *nix installations do not keep extensive user activity logs and it is highly likely that such an attack will go unnoticed. Even if an individual user notices an unusual last-access time on one of the files, without extensive logging, it is impossible to pin point the perpetrator.
- **No Quick Fix**: Unlike other security vulnerabilities like buffer overflows, this attack uses a **design** shortcoming combined with user/application carelessness and no patches would correct this problem overnight.
- **High Success Rate**: It is important to notice that the attack had a high success rate at installations where most users are computer literate. With increasing mainstream penetration of *nix systems, most users in the future would be ordinary users who cannot be expected to fully understand the vulnerabilities. This makes this attack a very potent threat.

3 Private-Sharing Mantras

The results presented in the previous section clearly establish the fact that there needs to be much better privacy protection in *nix installations. The possible solutions might lie in user education, use of new access control tools like ACLs [26], a new access control mechanism or possibly a combination of all three. In this section, we will concretely describe important features required for *good* privacy protection and try to identify possible solutions.

MANTRA #1: Do Not Risk More Than You Need To

This mantra implies that sharing one directory should not endanger the privacy of data in other unrelated directories. A majority of privacy breaches occurred due to users needing to have execute permissions on their home directory to share one subdirectory, but failing to adequately protect the sibling subdirectories. We believe that while sharing some data, a user should not be expected to remember if there is any private data already existing in some sibling subdirectory. Similarly, while creating new data, the user should not be expected to remember what is being shared and how that could effect the new data. It is unintuitive and likely to fail.

MANTRA #2: Do Not Trust Applications Completely

The second mantra dictates that protecting privacy of application created data should not be left entirely on the application developer. Even for a popular project like Mozilla, it took many years to notice and correct its data leak. Secondly, many applications might simply fail to foresee a privacy impact. For example, knowing that Java byte code can be decompiled back into source [6], the java compiler should ensure that its generated *.class* file has atleast the same permissions as the source[6]. Such seemingly unrelated design goals can be easily missed by application developers.

MANTRA #3: Increase Granularity of Protection

The third mantra advises to increase the granularity of data as well as users in private sharing. From users perspective, it should be possible to share data with a few users without requiring them to be a part of a certain user group (or be the complete *others* set). The new ACLs [26] do provide this facility; however, they are under-used in actual installations primarily due to lack of user education. Secondly, from the data perspective it should be possible to protect even the metadata of files, if desired.

MANTRA #4: Convenience, Convenience, Convenience

The fourth mantra emphasizes the need for user convenience in private data sharing. Lack of convenience, for example, while sharing a deep rooted directory or inability to represent the shared data differently for the user, will inevitably lead

[6]The Sun javac compiler does not ensure this

to improper permissions on certain data and risking privacy compromise. The *nix access control model, while admittedly not inconvenient in ordinary situations, fails to facilitate slightly involved situations like the ones described above.

MANTRA #5: Monitor and Remind Users
Finally, due to the unavoidable and error-prone human element, there should be mechanisms that can monitor the system and get a measure of the *privacy health*. We believe that this should be as ingrained into auditing tools as virus scans and boot-up password enforcement, as done by many enterprise security applications [30].

3.1 Evaluating Existing Mechanisms

Before we present our proposed solutions in the next section, let us analyze existing access control tools on the mantras described above. For the purpose of this discussion, we consider the following *nix access control mechanisms. Interested readers can refer to Appendix-B for a similar analysis of the Windows access control mechanisms.

- *Baseline *nix (BASE)*: The baseline *nix access control (*owner, group, others*) model as described in Section-2.1.
- *ACLs (ACL)*: ACLs [26] allow users to specify permissions at the granularity of an individual user. However, the types of permissions are the same *read, write, execute* as the baseline *nix model.
- *umask 077 (UMASK)*: The `umask 077` [29] will ensure that any new file created by a user or an application will have no permissions for users other than the owner. This can be seen as a potential way of preventing inadvertent leaking of private information.
- *Richer Access Control Semantics like AFS (RACS)*: The Andrew File System (AFS) [36] has additional privileges like looking up a directory, deleting files, creating new files. Using it as an example, we intend to evaluate if making the access control richer could be the solution.

Table-6 contains our analysis of these four access control mechanisms for the private sharing mantras described above with X indicating lack of support, ↑ indicating partial support and ↑↑ indicating good support.

Mantra #	BASE	ACL	UMASK	RACS
1	X	X	↑	X
2	X	X	↑	X
3	X	↑	X	↑↑
4	X	X	X	X
5	X	X	X	X

Table 6: Analyzing Tools for Mantras

As described earlier, the baseline *nix model fails on all these mantras. The ACLs provide a higher level of granularity as far as users are concerned, but they do not provide data level granularity like metadata privacy. They also fail on other mantras like protecting application data, or convenient means of sharing deep-rooted directories. The `umask 077` provides partial support in preventing against inadvertent exposure of user or application files by ensuring that when the files are created, they have no readable permissions for non-owners. However, in case the user ever modifies the permissions to share anything (even temporarily), it provides no further assistance. Also, `umask 077` is inconvenient to use since it is a global setting per user. For example, if working on a shared project with this setting, the user will have to specifically correct permissions every time a new file is created (a better mechanism would be using a user and directory specific umask). Using *richer access control semantics* of having separate permissions for listing directories, creating/removing files only helps in improving the granularity of protection. Without a concrete mechanism of directly and selectively sharing deep rooted data or protecting application data, this mode fails on the rest of the mantras. None of the existing *nix tools provide any auditing facilities for data privacy, thus failing on the fifth mantra.

Note that even with a combination of all four mechanisms, there is still inadequate support for privacy protection.

4 Privacy Enhancements

The analysis presented in the previous section indicates a need for better access control and monitoring mechanisms. In this section, we present our two solutions that can be used independently or together to facilitate stronger privacy protection in *nix systems. The first solution is a Privacy Auditing Tool that monitors the privacy health of an organization and can alert users/administrators of potential threats (the fifth mantra). The second solution is a new access control model, View-Based Access Control, that modifies the data sharing mechanisms of *nix systems and succeeds on the remaining private-sharing mantras. Using the two solutions together provides an excellent data sharing environment.

4.1 Privacy Auditing Tool

The aim of the privacy auditing tool is to periodically monitor user home directories and identify potential private data exposures. A similar approach is used by most enterprise security applications like [30] that audit user systems and enforce compliance to security policies, for example, requiring laptop owners to keep a boot-up password, or system administrators to enforce stricter password rules and so on. In a similar vein, the privacy auditing tool will scan user home directories and alert administrator or the users directly if their private data can be accessed by unauthorized users.

The design of such a tool is very similar to the design of the attack described in Section-2.3.1. A number of test accounts are created on the monitored system with different group memberships, since it is possible that some user group might have access to more private data than others. The tool is then run from these test accounts to identify exposed private data. The auditing tool can be used either to obtain only higher level statistics, as described in the attack or more user-specific information which can alert users directly of their *pri-*

vate directories that can be accessed by unauthorized users. A variant would be allowing users to themselves invoke an audit of their home directory. Yet another variant would be to allow the tool to automatically correct some of the obvious misconfigurations like emails.

Even though this solution does not solve the underlying access control problem, it has the following **advantages**:

- The privacy auditing tool does not require operating system or file system changes and so can be easily incorporated into enterprise infrastructures. The auditing solution is the quickest way of mitigating this vulnerability.
- Since existing security auditing tools operate in a similar mode, this tool can be easily added on as a privacy protection module to such tools.
- Even with a better access control model, the unavoidable and error-prone human involvement in protecting private data makes such a tool an important component of a secure enterprise.

Next, we discuss a more proactive solution to address the *nix privacy shortcomings.

4.2 View-Based Access Control (VBAC)

Our second solution is the design of a new access control mechanism called the View-Based Access Control (VBAC). Similar to the *nix access control, VBAC is also a discretionary access control model, thus keeping security within the control of the data owner. VBAC is motivated by the private-sharing mantras described in Section-3 and is based on the following design goals:

- *Act only on primary objects*: Private sharing in *nix in its current form requires a two step approach of (a) sharing desired data, and (b) protecting other unrelated data. The second step, adequately protecting sibling directories or newly created directories, is unintuitive and should be removed. This design feature will help us adhere to the first mantra.
- *Keep application data only in the owner's view*: A severe privacy breach occurred due to improper handling of application profile data. Such data should be viewed only by the owner and unless specifically allowed, should not be visible to other users. This design feature adheres to the second mantra.
- *Allow hiding of sibling directories from other users*: The POSIX ACLs increase granularity of protection for users, allowing data to be shared with individual users. Combining this with an approach that can completely hide sibling data from other users, thus protecting file metadata will comply with the third mantra.
- *Allow extracting deep rooted directories*: In order to share deep rooted directories, it should be possible to simply pluck them from the file system tree and put them in the view of desired users. Also, it should be possible to share a different representation of the data without impacting the owners' view of the file system. This will provide the convenience desired by the fourth mantra.

Based on these design goals, the VBAC access control model creates a new file system primitive called a "**view**". Informally speaking, a data owner can define a view of her home directory, dictating what another user gets to see when he attempts to access it. By adding only the data she wants to share into this view, other data remains protected. Also, it is possible to add a deep rooted directory directly to this view and it can be represented differently. Using such a mechanism, unless the owner explicitly adds her application data into a shared view, it will be always hidden from other users. More details follow in the next section.

5 VBAC: Design and Implementation

VBAC extends the *nix access control model by adding a *view* primitive, that presents a different file system structure to different users. For every user, there is one owner-view of the home directory which is the same as the standard home directory in current systems. In addition, the owner can define new views of the home directory for other individual users, user groups or the *others* set. For example, a user *bob* can create a view of his home directory for a user *alice* and another view for his user group *faculty* and yet another view for all *other* users. He can then add desired data to appropriate views depending on what he wants to share. The added data could be a deep rooted directory and can be shared using a different name. Other users can access their view of *bob*'s home directory using the same `~bob`. The underlying system **automatically routes** them to their appropriate view and users continue to see the view directory as `~bob` . This ensures that no current scripts or access habits are disturbed.

The VBAC model uses the same permission types as baseline *nix - *read*, *write* and *execute* and they have the same semantics. VBAC only adds another layer of access control by making a higher level decision of what a user gets to see or not. After that decision, whatever a particular user has in his/her view, it is access controlled using the baseline *nix permissions. We believe that this feature makes VBAC an elegant extension of the *nix model.

Also, a user can decide to **switch off** the additional VBAC layer providing other users with the same view as the owner-view (of course, access to data is controlled by the lower layer of *nix access control). This implies that VBAC can be incrementally introduced into a system without forcing all users to migrate to it immediately.

Separating the owner view from the views of other users offers us great advantages:

1. No sibling directories are put at risk, since only the directories added to user views (that is, the directories that were to be shared) are visible to other users (the first mantra).
2. No user application files are visible to other users. Only the owner-view contains files like *.history*, *.mozilla*, which were responsible for significant privacy breaches (the second mantra).
3. File metadata can be protected simply by not adding them to another user's view. If a file is solely in the

owner's view, other users can not access metadata or data or even find out existence of a file. In addition, VBAC uses POSIX ACLs for fine-grained user access control (the third mantra).

4. As mentioned above, VBAC allows sharing a deep rooted directory without giving permissions throughout the entire path. Also the shared directory can be represented differently in different views (the fourth mantra).

To avoid managing views at the granularity of individual users, we provide two mechanisms for doing selective sharing of data by using the group views.

This first mechanism uses **security-by-obfuscation** (SBO). Recall that we mentioned that in order to do selective sharing under an execute-only directory, a simple solution was to use tough-to-guess directory names. However, it was deemed infeasible since the owner would have to keep tens or hundreds of cryptic names. With the separation of views, an owner can now share a particular directory with a tough-to-guess name in the view, while keeping the original name in the home directory (owner-view). To achieve this, the other users' view is set to execute-only permissions and while adding a directory to the view, the owner also gives a *passphrase* which is used to encrypt the name of the directory being added. The encrypted cipher text is appended to the directory name and the resulting string is used as the target name in the view. Now, the owner authorizes a user to access this directory by giving that user the original directory name and the *passphrase*. Notice the similarity with the authorization mechanism in original execute-only *nix - only one extra piece of information, the passphrase, is delivered using the same out-of-band channels like email. By using cryptographically tough-to-guess names, we are able to bring this user authorization definition much closer to the underlying system's definition, a shortcoming in the originial *nix model.

In order to prevent users having to remember the passphrase every time they need to access another user's shared directory, we provide another mechanism to do selective data sharing using a group view. This second method uses **POSIX ACLs**. In this case, when an owner adds a directory to a view, it is shared with the desired name, but with no permissions for any users other than the owner. Along with that, the directory is flagged and the passphrase is stored in a secure location only accessible by the superuser[7]. The owner still authorizes the users in the same manner - by informing them of the directory name and the passphrase. However, before accessing the shared directory users have to perform an additional step. They first obtain access by executing a new trusted command (see Appendix-C) that, provided the passphrase was correct, updates the ACL associated with that directory allowing access to that user. All accesses after this step can proceed without having to enter the passphrase. *An added advantage of this method is that all users that access the directory have their names included in the ACL, which can be viewed by the owner, thus serving as an excellent auditing tool.*

[7] It is stored as a trusted extended attribute [7]. Also, similar to OS passwords, the passphrase can be hashed and salted for greater security

5.1 Implementation

In this section, we describe our implementation of the VBAC access control model.

The first decision that we had to make was to identify the OS layer at which to implement VBAC. We had two choices, (a) the Virtual File System (VFS), or (b) a new underlying filesystem. The Linux Virtual File System (VFS) is a file system interface that sits over all underlying filesystems. It provides a default implementation of most filesystem functions like `lookup`, `mkdir` and also provides an implementation of *nix access control checks. Individual filesystems can choose to override these functions by implementing them in the filesystem.

We chose the second option for the following reasons - (a) modifying VFS would have impacted all filesystems that use the default VFS implementation for their functions, (b) developing an individual filesystem is relatively easier as it can be developed and debugged as a loadable kernel module, (c) we do not foresee the VBAC model to be very relevant for data other than user home directories (for example, */usr, /etc, /var*) and (d) if VBAC is accepted by the community, it is easier to distribute a new filesystem than to get linux kernel developers to accept and distribute a new kernel!

This new file system is called **viewfs** and is based on the linux ext2 filesystem [20]. Most of ext2 functions like disk placement of data blocks are reused. viewfs is developed as a loadable kernel module and can be loaded into the kernel without kernel recompilation. viewfs was implemented on a Linux 2.6 kernel. Next, we explain the implementation of important VBAC features. Please note that for some of the details, little familiarity with the Linux VFS is helpful.

5.1.1 VIEW

The foremost VBAC concept is that of a view. From an implementation perspective, a view is a regular directory within the directory containing the owner's home directory (like `/home`). In other words it is a sibling directory to the owner home directory. However, the view directory has a special name of the form: "*.owner.uview.username*" or "*.owner.gview.groupname*" or "*.owner.oview*". This name is restricted, that is, users cannot use this name for naming other directories. This restriction helps to identify a directory being a view of another directory and is used to do automatic routing of users to their views. The restriction is enforced in the file system `mkdir` function implementation. The first type of name "*.owner.uview.username*" is used to create a view for an individual user. For example, if *bob* creates a view for *alice*, the view will be called "*.bob.uview.alice*". The second type is for a user group and the third is for *other* users. Access is controlled to these views, for example, to prevent user *cathy* from accessing a view for *alice*, by ACLs set at view creation. The period ('.') before the view names keeps them hidden from plain view.

While it would appear that a more elegant way is to create

the view directories within the owner's home directory (no clutter in */home*), it is not feasible for two reasons:

- First, since we use the same *nix permission types *rwx*, in order to provide access to a view directory within owner's home directory, we would be required to give execute permissions on the home directory, which was the original cause of the privacy breach!
- Secondly, it is easier to implement the view directory as a sibling to home directory due to the VFS implementation of path lookups. VFS looks up a path name `dir/child/grand-child` in a loop by first looking up `child` in the directory `dir` and then looking up `grand-child` in `child`. For routing *alice* to her view of *bob*'s directory, the lookup seeking `bob` in the directory `/home` just returns the entry for `.bob.uview.alice` instead.

Next, we concretely describe the automatic routing to views. A VFS directory lookup ocurrs in the following manner. Given a name to look up, the directory entry (or *dentry*) cache (or *dcache*) is looked up for the desired directory name. The dcache indexes cached dentries by hashes of their names. If there is a cache miss, the call passes onto the underlying filesystem for looking up that name. That is followed by the inode number lookup to find the inode number corresponding to the name and if it exists, the inode lookup that gets the object metadata.

We modify this filesystem lookup procedure by first checking if there exists an appropriate view directory. For example, if *alice* is looking for a directory `bob`, we check for the existence of a directory called "*.bob.uview.alice*". At this stage, we have the complete state necessary for completing this lookup (since view is the sibling to the home directory). If the view exists then the dentry associated with the view directory is returned and is cached in the *dentry* cache on return. In order to work with different user views in the cache, we modify the hash of the dentry by hashing the view name as opposed to the original directory name. This is feasible since the VFS hash function can be overridden by the underlying filesystem.

An important point to note is that this implementation does not interfere with the I/O paths at all, that is when file data is being read or written back to disk. One potential performance impact of the implementation is that a single directory name lookup can cause multiple view name lookups. However, as our initial experiments with benchmarks show, the total overheads are still minimal. Secondly, we foresee *nix installation using viewfs only for user home directories. Therefore the vast majority of system lookups that are to standard OS and other infrastructure files contained in `/usr`, `/etc`, `/bin` are not affected at all.

As mentioned before, an individual owner can choose to switch off the VBAC model for users accessing her home directory. This is accomplished by keeping an extended attribute [7] with the user home directory. The lookup described above first checks for this extended attribute and in case the user chooses to switch off VBAC, the normal ext2 file system lookup is performed providing the basic *nix model.

5.1.2 SHARING DATA

The next important viewfs implementation is its mechanisms for adding data to views. It allows adding deep rooted directories directly to user views and possibly with a different name. Also, changes made to the directory in one view should be immediately reflected in all other views. The first idea that comes to mind to facilitate this is directory hard links. A directory hard link is the same inode as the original directory but can have a different name and can be created at any location within the filesystem without worrying about the access along the path to the original directory (unlike a symbolic link). In fact there is no way to distinguish a hard link from the original directory.

However, directory hard links are not allowed by most operating systems including Linux even though it is not mandated by the POSIX standard. The reason is that many OS mechanisms like reference counting and locking consider the file system to be an acyclical tree and hard links can cause cycles. For example, for `a/b/c`, commands "*link e a/b*" and "*link e/c a*" cause a cycle. This will also break many existing applications that traverse the file system assuming it to be a tree. Symbolic links can also cause a cycle but they are easily identifiable since the link is a different inode that stores the complete path to the original pointed-to location. Hard links, as mentioned before, are unidentifiable and cycle detection for hard links can be expensive.

In the context of viewfs however, we can very easily prevent any cycle formation. We can do this by only allowing users to create a link from inside the view pointing outside and never in the other direction. This can be checked while adding data to a view (creating the hard link) and since views have restricted name, such a check would not be expensive. Even though we had a working implementation of this approach, there is an additional issue specific to the linux operating system and its implementation of the VFS dentry cache. This view implementation can break in certain situations, for example, when a directory and its hard link are both cached and one is deleted. Interested readers can follow Linus Torvald's explanation debating Reiser4 filesystem's directory aliasing at `http://kerneltrap.org/node/3749` [14]. It is unclear if the same holds for other variants like FreeBSD. However, we preferred linux and opted to take an alternate path.

Similar objectives can be achieved in linux using a bind `mount` [29]. A bind-mount mounts one portion of the filesystem tree at another location. Since it is a separate file system mount, the linux implementation issue discussed earlier does not apply [14]. This provides both sharing the deep rooted directory and sharing it with a different name. For viewfs, an additional requirement is to first create the mount point which will be the desired directory name in the view and is done while adding data to the view. Appendix-C describes new commands created for facilitating this viewfs implementation and an example viewfs session demonstrating its usage.

5.1.3 VBAC Usability

The following characteristics make VBAC and the viewfs implementation superior in design in regards to usability and integration with existing *nix systems.

- VBAC elegantly complements the *nix access control by only adding a higher level *can-see/cannot-see* decision. Also since the view concept is regularly practiced as an access control tool in databases [37, 42], we believe it will not be tough for users to transition to it.
- Individual users can choose to switch on/off the VBAC access control model, thus allowing incremental introduction of VBAC. In addition, only directories mounted on viewfs are impacted, thus isolating its influence.
- In viewfs, users only need to perform operations on data which they want to share. All other data, including application data, is automatically protected without any explicit user commands.
- VBAC requires only data owners to perform explicit operations. Users accessing shared data continue to do so in normal *nix fashion (like `cd ~bob`), except when using the SBO or ACL private-sharing methods.

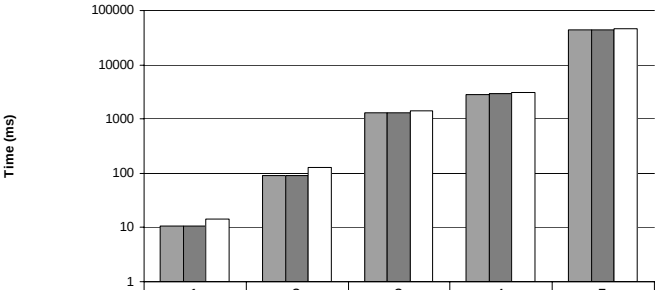
5.2 Performance Evaluation

We evaluated our viewfs implementation against three popular filesystem benchmarks and compared it with the baseline ext2 performance, the file system viewfs is based upon. The experiments were conducted on a P4 1.6 GHz Dell Inspiron 8200 with 512 MB RAM running RedHat Linux with kernel 2.6.11.3. All results were averaged over multiple runs. We used two viewfs scenarios - (1) *viewfs-owner*: when an owner is accessing her data, and (2) *viewfs-other* when a user is accessing the data from an *others* view. Note that additional view name lookups occur only in the latter scenario, so the former scenario is an indication of any other viewfs overheads, for example, of using bind mounts and would also be an estimate of impact on users with switched off VBAC model.

5.2.1 Andrew Benchmark

Andrew benchmark [27] is a popular filesystem benchmark. It emulates a software development workload and has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files. For our experiments, it compiled an OpenSSH-2 client. In the viewfs-other implementation the benchmark was run by an *other* user in the owner's view directory. Figure-3 plots the times (in ms) on log scale for each of the phases. Table-7 show the overheads of the viewfs-other implementation over ext2.

Note that the viewfs-owner implementation performs very similar to the ext2 implementation. This implies there is little impact on performance of owners or users with switched off VBAC model. Also, the total overhead for viewfs-other over ext2 is only 3%, implying reasonable overheads of additional view-name lookups under this workload. The overheads in



	1	2	3	4	5
Ext2	10.72	89.46	1287.81	2870.67	44922.25
Viewfs-owner	10.77	90.84	1288.52	2904.28	44949.72
Viewfs-other	14.62	126.92	1429.48	3140.55	45977.71

Figure 3: Andrew Benchmark Results

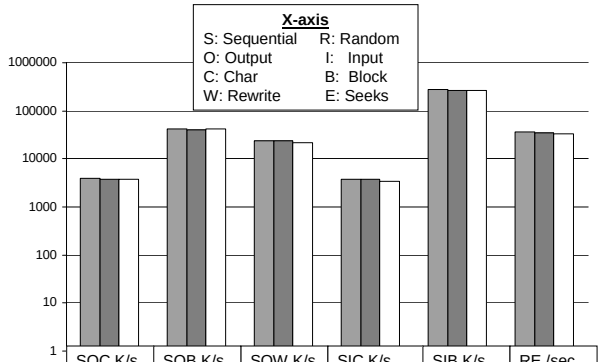
Phase #	Overheads
1	36%
2	41%
3	11%
4	9%
5	2%
TOTAL	3%

Table 7: Andrew Benchmark overheads of viewfs-other over ext2

phases one and two are greater since their lookup costs are comparable to the total costs (less than 100 ms). For the later phases, other costs are more dominant.

5.2.2 Bonnie

In order to test our thesis that viewfs should not impact I/O performance, we evaluated viewfs against ext2 on the Bonnie benchmark [3]. Bonnie tests the speed of file I/O using standard C library calls. It does reads and writes of blocks in random or sequential order and also evaluates updates to a file. The tests were run for a 400 MB file. Figure-4 shows the results for the three implementation for various I/O modes. The X-axis lists the I/O modes and the Y-axis plots the speeds for those operations. As can be seen, for all read/write modes, there is practically no difference between ext2 and viewfs. This proves that viewfs does not have I/O overheads.

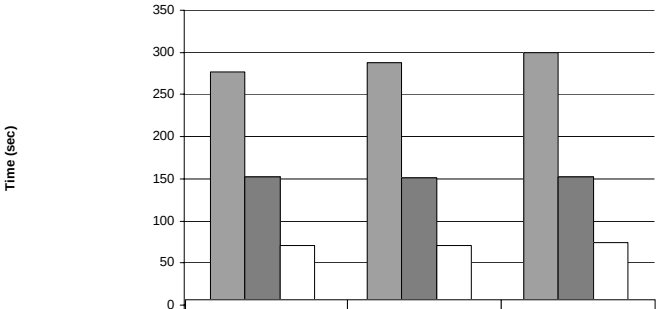


	SOC K/s	SOB K/s	SOW K/s	SIC K/s	SIB K/s	RE /sec
Ext2	3916	41195	24020	3706	272410	35984
Viewfs-owner	3850	40327	23677	3691	262909	34591
Viewfs-other	3827	41044	21562	3500	265958	33282

Figure 4: Bonnie Benchmark Results

5.2.3 Am-utils

The final benchmark used was the Berkeley Automounter [2] am-utils. It is a filesystem performance benchmark that configures and compiles the am-utils software package inside a given directory. Overall, this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, symlink. It is believed to be a good representation of a usual system workload.



	ext2	viewfs-owner	viewfs-other
Elapsed Real Time	276	287.106	299.362
User CPU Time	151.978	151.035	151.678
System CPU Time	70.17	70.777	74.092

Figure 5: Am-utils Benchmark Results

Figure-5 shows the results of timing the viewfs-other and viewfs-owner implementations with ext2 using the `time` [29] utility. The graph shows that there is only a slight overhead introduced by viewfs (8% in elapsed real time, time between invocation and termination, for viewfs-other). This shows that for typical system usage, viewfs will perform very close to the existing filesystems.

Overall, the experimental evaluation shows that viewfs has an acceptable performance and is viable for real systems.

6 Related Work

We compare our work to three pieces of related work - (1) *nix access control models, (2) alternate technologies like cryptographic file systems, and (3) use of views as access control tools in other domains.

6.1 *nix Access Control Models

The *nix access control model follows primarily from the original UNIX access control [34, 33]. It is a discretionary access control (DAC) model [16] and access is granted based on the identity of the subject (user or user group). It is discretionary since access to an object is controlled by the object owners. Most of the research in *nix access control model aims to either improve the granularity of protection (for example, POSIX ACLs [26]) or counter the threat of malicious programs exploiting the `root` (superuser) privileges. One variant of Linux developed by National Security Agency [11], called the Security-Enhanced Linux (SELinux) [15] supports a mandatory access control model [17], in which an administrator sets a security policy which is used to determine the access granted to an object and users have limited control on their data. Another access control model is the Role-Based Access Control (RBAC) model [22, 35], supported by Sun Solaris operating system, in which security attributes can be assigned to user roles (a process or task). This helps reduce the threat of malicious programs exploiting the `root` privileges. In a similar vein, the Rule-Set Based Access Control (RSBAC) model [31] aims to protect against `root` vulnerabilities and improve granularity of protection. There has also been work on a privacy model [23, 24, 25] in access control. However, that work is aimed at guiding organizations on how to control their information flow to ensure privacy of collected user-data (for example, healthcare records). To the best of our knowledge this work is the first critical look at the privacy support for *data sharing in a multi-user *nix operating system*. Our proposed View-Based Access Control (VBAC) model is a specialized access control model aimed at providing stronger privacy protection in such environments.

6.2 Alternate Technologies

A common argument against privacy protection mechanisms is that users who care about their privacy would use a stronger security mechanism like encryption. However, sharing encrypted data with many users is a challenging problem. Cryptographic file systems like CFS [18], CryptFS [46] provide transparent mechanisms of ensuring data confidentiality using cryptographic primitives. Data is stored in an encrypted format on disk and is decrypted (or re-encrypted) on-the-fly while reading from (or writing to) the disk. In practice, such filesystems are seldom used in multiuser environments. This is due to the I/O performance effects and also importantly, lack of user education in encryption technologies. In fact a user study reported that even experienced computer users could not use PGP 5.0 in less than 90 minutes and that one-quarter of the test subjects accidentally revealed the secret they were supposed to protect [43]. A recent cryptographic file system NCryptFS [45] while providing better convenience features, modifies various kernel components like process management, dentry cache and inode cache. This will limit its widespread adoption. Secondly, users have to specifically *attach* to a shared directory, as opposed to continuing to use "$\sim$bob" in viewfs. Thus, for its *convenience in use and easier integration with existing systems*, we believe viewfs to be a better suited tool for privacy protection.

6.3 Access Control using Views

The use of views as an access control tool has been primarily researched in the area of databases [37, 42]. Using database *views*, users are only shown the relevant data that they have access to. This is similar in concept to VBAC in which only data that needs to be shared is added to a user's view. A view-based access control model has also been used in networking to control access to management information in the Simple Network Management Protocol (SNMP) [44]. There is also an attempt of creating a new operating system called View-OS [41] that presents a different view of the system resources including the filesystem, to an OS process. Also, `chroot` [29] can be used to present a different *root* (/) directory to a process. In

contrast, our view based mechanism is a comprehensive privacy protection mechanism *preventing many kinds of privacy breaches* and *works with existing *nix systems*.

7 Conclusions and Future Work

We have critically analyzed the *nix access control model for privacy support in its data sharing mechanisms. We identified a number of design inadequacies that, combined with user or application's privacy-indifferent behavior, lead to privacy breaches. We tested two *nix installations of a few hundred users and found that a massive amount of private data is inadequately protected including emails and actual passwords to financial and other sensitive websites. Based on our analysis, we promoted five data sharing mantras that should be followed for best privacy protection. Then, we proposed two solutions which when used jointly adhere to all the mantras and provide strong privacy protection. As part of the second solution, we propose a new view-based access control (VBAC) model which separates the owner's view of the home directory from other users. We also presented a new VBAC-enabled file system, called viewfs. Our experiments with three popular filesystem benchmarks prove that viewfs has acceptable performance, with little overheads. As part of our future work, we plan to deploy viewfs at a real *nix installation and conduct a thorough usability study.

References

[1] *http://news.com.com/IDC+Linux+server+sales+to+hit+9.1+billion+in+2008/2100-1010%5F3-5479681.html*.

[2] Berkeley Automounter http://www.am-utils.org.

[3] Bonnie Benchmark http://www.textuality.com/bonnie.

[4] Google Mail http://www.gmail.com.

[5] http://support.microsoft.com/default.aspx?scid=kb;en-us;304040#xslth4157121124120121120120.

[6] Jad: Java decompiler http://www.kpdus.com/jad.html.

[7] Linux manual pages (5). man 5 attr.

[8] Microsoft Hotmail http://www.hotmail.com.

[9] Mozilla Contents http://www.holgermetzger.de/pdl.html.

[10] Mozilla http://www.mozilla.org.

[11] National security agency http://www.nsa.gov.

[12] Opera http://www.opera.com.

[13] Pine http://www.washington.edu/pine/.

[14] Reiser4 aliasing debate http://kerneltrap.org/node/3749.

[15] Security-enhanced linux http://www.nsa.gov/selinux/.

[16] Wikipedia Discretionary Access Control http://en.wikipedia.org/wiki/Discretionary%5Faccess%5Fcontrol.

[17] Wikipedia Mandatory Access Control http://en.wikipedia.org/wiki/Mandatory%5Faccess%5Fcontrol.

[18] M. Blaze. A Cryptographic File System for Unix. In *ACM Conference on Computer and Communications Security*, 1993.

[19] D. Cappelli and Michelle Keeney. Insider threat: Real data on a real problem. *Available from: http://www.cert.org/nav/present.html*.

[20] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, 1995.

[21] Wikipedia Social Engineering. *http://en.wikipedia.org/wiki/Social%5Fengineering %5F(computer%5Fsecurity)*.

[22] D. Ferraiolo and D. Kuhn. Role Based Access Control. In *15th National Computer Security Conference*, 1992.

[23] S. Fischer-Hubner. Towards a Privacy-Friendly Design and Usage of IT-Security Mechanisms. In *17th National Computer Security Conference*, 1994.

[24] S. Fischer-Hubner. Considering Privacy as a Security-Aspect: A Formal Privacy-Model. *DASY Papers No. 5/95, Computer and System Sciences, Copenhagen Business School*, 1995.

[25] S. Fischer-Hubner and A. Ott. From a Formal Privacy Model to its Implementation. In *21st National Information Systems Security Conference*, 1998.

[26] A. Grunbacher and A. Nuremberg. POSIX Access Control Lists on Linux. *http://www.suse.de/ agruen/acl/linux-acls/online*.

[27] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[28] Carnegie Mellon Software Engineering Institute. CERT. *http://www.cert.org*.

[29] Linux Manual Pages. *man command-name*.

[30] Symantec Security Manager. http://enterprisesecurity. symantec.com/ products/products.cfm?productid=45.

[31] A. Ott and S. Fischer-Hubner. The Rule Set Based Access Control (RSBAC) Framework for Linux. *Karlstad University Studies*, 28, 2001.

[32] Mozilla Bug Report. Bug 59557. *https://bugzilla.mozilla.org/show%5Fbug.cgi?id=59557*.

[33] D. Ritchie. The UNIX Time-Sharing System: A Retrospective. *Bell Systems Technical Journal*, 57(6):1947–1969, 1978.

[34] D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[35] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based Access Control Models. *IEEE Computers*, 29(2):38–47, 1996.

[36] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), 1990.

[37] A. Sheth and A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):180–236, 1990.

[38] Emil Sit and Kevin Fu. Web Cookies: Not Just a Privacy Risk. *Communications of the ACM*, 44(9), 2001.

[39] Slashdot. Kevin Mitnick Interview. *http:// interviews.slashdot.org/article.pl?sid=03/02/04/2233250 &mode=nocomment&tid=103&tid=123&tid=172*.

[40] United States Secret Service and CERT Coordination Center. 2004 E-Crime Watch Survey. *Available from: http://www.cert.org/nav/allpubs.html*.

[41] View-OS: A Process with a View. http://savannah.nongnu.org/projects/view-os.

[42] C. Wang and D. Spooner. Access Control in a Heterogeneous Distributed Database Management System. In *Symposium on Reliability in Distributed Software & Database Systems*, 1987.

[43] A. Whitten and J. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security*, 1999.

[44] B. Wijnen, R. Presuhn, and K. McCloghrie. View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP). *RFC 2275*, 1998.

[45] C. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *USENIX Annual Technical Conference*, 2003.

[46] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. *Tech Report CUCS-021-98, Computer Science, Columbia University*, 1998.

Appendix

A. Privacy Study

A study that evaluates data and user privacy characteristics in real systems can potentially raise certain ethical issues on whether such experiments should be conducted and potential threats reported to a public forum. We appreciate any user concern in this regard.

First, we carefully crafted our experiments to prevent any privacy violation from the study itself. Our scanning tool does **not** store any user-specific information on disk. When the tool runs, it obtains the list of users from `/etc/passwd`, anonymizes the users and randomizes the order of scanning. This order is never written out to disk or printed on screen and is purged off the memory once the tool terminates. In case of measuring the number of emails that are readable, a `grep` [29] like tool is used to only measure the number of times the word "*Subject:*" appears at the start of a line. Also, for measuring the number of passwords that can be extracted (since mozilla stores them in base-64 encoding) only the number of such encoded passwords is calculated. This way, we ensure that we do **not** violate any user privacy, though such an attack is very feasible and a malicious user can obtain user specific private data. The US Secret Service and CERT study [19] proves that such malicious users exist on the inside and trusting internal users to be ethical is incorrect.

Secondly, we believe that this study was essential to create user awareness on this very important issue and expose existing threats to user privacy, that can be exploited without significant technical sophistication. Also, our proposed defenses can solve or at least mitigate these privacy protection problems immediately.

B. Windows Access Control

While we focus primarily on the *nix access control model, it is interesting to briefly analyze the windows access control model. Windows is most commonly used in a Personal Computer (PC) environment, so data sharing for users on the same system is not very common. However, the Windows NT line of operating systems (NT, Windows 2000, XP) does have multi-user support.

Windows access control is richer than *nix and includes additional permissions for listing contents of a directory (distinct from the *read* permissions), creating files in a directory (distinct from the *write* permission) and more. Also, permissions can be granted at the granularity of individual users. A major difference for our analysis is the fact that in Windows NT/2000/XP Pro, it is possible to share a directory with other users without giving permissions for the complete path to that directory. For example, to share `grand-child` with the path `dir/child/grand-child`, there is no requirement for execute permissions on `dir` or `child`. However, in Windows XP operating system, the *recommended* way (for XP *Home*, it is the only way) of sharing a folder with other users on the same computer is to **move** it to the "*Shared Documents*" folder [5]. This is an undesirable property since it forces the owner to change the directory structure.

For applications, Windows recommends creating private files in the hidden "*Documents and Settings / userName / Application Data*" directory, thus creating a single location which can be more easily protected. However, there is no certain way to ensure that applications follow this principle.

Overall, this access control (except in XP Home) provides good support for preventing inadvertent exposure, only partial support for protecting application data and improves the granularity of protection. In regards to convenience, Windows allows a deep rooted directory to be shared easily (no need to set parent permissions). However, it does not let owners represent their data differently for sharing it with other users.

C. viewfs Commands

Following new commands have been implemented for viewfs:

1. **createview** -[u | g| o] [user | group]
This commands creates a user/group/others view for the invoker's home directory. It also sets appropriate ACLs to control access to the view. For example, bob executing "`createview -u alice`" or "`createview -o`".

2. **add2view** -[u | g| o] [user | group] [-b | -a] [src] [target]
This command adds the src directory to the appropriate view and sets its name to be *target*. Also if -b option is used, a passphrase is obtained form the user and the directory is shared using the security-by-obfuscation (SBO) method described in Section-5. If -a option is used, the ACL method is used. For example adding *jobsearch* directory to an *others* view as *CV* using SBO - "`addview -o -b jobsearch CV`". This will prompt the user to enter a passphrase (pp) and the directory is shared with the name "CV-$\{CV\}_{pp}$" where {} indicates encryption. To allow ordinary users to perform a mount, this script is setuid root, though root privileges are dropped at initialization and gained only for the mount operation using `seteuid` [29].

3. **getacc** [target]
This setuid command prompts for a passphrase that protects access to the target directory, when using the ACL method. If the passphrase is correct, the ACL of target is modified to give access to the invoking user.

4. **vcd** [target]
This command is to `cd` into a directory protected by the SBO method. It prompts for a passphrase and encrypts the target name with it to obtain the actual directory name.

There are also other functions to remove a view, unshare a directory from a view etc. Figure-6 contains an example viewfs session, in which *bob* has created a view for *others* and *alice* is accessing that view.

Lines 2–8 show the contents of *bob*'s home directory. In lines 9, 10 *bob* adds his *personal* directory to *others* view using the ACL method. In lines 11, 12 he adds his *CS6210* directory using the security-by-obfuscation (SBO) method. Finally he adds his research and web directory in lines 13, 15. Lines 18–25 show the contents of the view directory. Notice the name of the shared *CS6210* directory, the permissions of the shared *personal* directory, as done in the ACL method (see Section-5) and execute-only permissions of the view directory (line 20) for SBO protection.

Alice accesses *bob*'s home directory ∼bob, but cannot list its contents (line 4). She can access the *public_html* and *research* directories, since they have not been protected by SBO or ACL methods. In order to access the *personal* directory, she executes the `getacc` command (line 11) and enters the passphrase, which *bob* would have given her out-of-band. After the command she can access the directory. And listing the ACL of *personal* directory shows the modification allowing access to *alice*. Finally, in lines 25–28, she accesses the *CS6210* directory, protected by the SBO method, using `vcd`.

#	BOB	ALICE
0	**[bob@localhost bob]$** pwd	**[alice@localhost alice]$** cd ∼bob
1	/mnt/vw/homes/bob	**[alice@localhost bob]$** pwd
2	**[bob@localhost bob]$** ls -l	/mnt/vw/homes/bob
3	total 7	**[alice@localhost bob]$** ls -l
4	drwxr-xr-x 2 bob bob 1024 Apr 23 01:19 courses	ls: .: Permission denied
5	drwxr-xr-x 2 bob bob 1024 Mar 26 05:14 mail	**[alice@localhost bob]$** ls public_html
6	drwxr-xr-x 2 bob bob 1024 Apr 23 20:59 personal	index.html
7	drwxr-xr-x 4 bob bob 1024 Apr 8 17:06 research	**[alice@localhost bob]$** ls research
8	drwxr-xr-x 2 bob bob 1024 Mar 26 05:16 www	linux-2.6 paper
9	**[bob@localhost bob]$** add2view -o -a personal personal	**[alice@localhost bob]$** ls personal
10	Enter Passphrase: ***** (bob12)	ls: personal: Permission denied
11	**[bob@localhost bob]$** add2view -o -b courses CS6210	**[alice@localhost bob]$** getacc personal
12	Enter Passphrase: ****** (group4)	Enter Passphrase: ***** (bob12)
13	**[bob@localhost bob]$** add2view -o research research	**[alice@localhost bob]$** ls personal
14		passwords
15	**[bob@localhost bob]$** add2view -o www public_html	**[alice@localhost bob]$** getfacl personal
16		# file: personal
17	**[bob@localhost bob]$** cd ../.bob.oview/	# owner: bob
18	**[bob@localhost .bob.oview]$** ls -al	# group: bob
19	total 10	user::rwx
20	drwx- -x- -x 6 bob bob 1024 Apr 23 20:59 .	user:alice:r-x
21	drwx- -x- -x 8 bob bob 1024 Mar 20 21:56 ..	group::−−−
22	drwxr-xr-x 2 bob bob 1024 Apr 23 01:19 CS6210-CSYrzy3dw1uIs	mask::r-x
23	drwx- - - - - - 2 bob bob 1024 Apr 23 20:59 personal	other::−−−
24	drwxr-xr-x 2 bob bob 1024 Mar 26 05:16 public_html	
25	drwxr-xr-x 4 bob bob 1024 Apr 8 17:06 research	**[alice@localhost bob]$** vcd CS6210
26		Enter Passphrase: ****** (group4)
27		**[alice@localhost CS6210-CSYrzy3dw1uIs]$** ls
28		intro.6210

Figure 6: Example viewfs session