Making Linux Protection Mechanisms Egalitarian with UserFS

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 17 2011
LIBRARIES

by

Taesoo Kim

B.S., Korea Advanced Institute of Science and Technology (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

ARCHIVES

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2011

Certified by . . . .
Nickolai Zeldovich
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Making Linux Protection Mechanisms Egalitarian with UserFS

by

Taesoo Kim

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

UserFS provides egalitarian OS protection mechanisms in Linux. UserFS allows any user—not just the system administrator—to allocate Unix user IDs, to use `chroot`, and to set up firewall rules in order to confine untrusted code. One key idea in UserFS is representing user IDs as files in a `/proc`-like file system, thus allowing applications to manage user IDs like any other files, by setting permissions and passing file descriptors over Unix domain sockets. UserFS addresses several challenges in making user IDs egalitarian, including accountability, resource allocation, persistence, and UID reuse. We have ported several applications to take advantage of UserFS; by changing just tens to hundreds of lines of code, we prevented attackers from exploiting application-level vulnerabilities, such as code injection or missing ACL checks in a PHP-based wiki application. Implementing UserFS requires minimal changes to the Linux kernel—a single 3,000-line kernel module—and incurs no performance overhead for most operations, making it practical to deploy on real systems.

Thesis Supervisor: Nickolai Zeldovich
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my advisor, Prof. Nikcolai Zeldovich, for his invaluable comments and endless supports. I would also like to give special thanks to Prof. M. Frans Kaashoek and Prof. Robert Morris for making me motivated every single moment at MIT. I also owe a lot to my colleagues of PDOS, namely, Ramesh Chandra, Chris Laas, and Xi Wang, and thank their ingenious suggestions to complete this work. I should also thank Samsung Scholarship Foundation for their support of my graduate study. I am delightful to thank my parents, Junghee Kim and Younghye Kim, and my sister Munjung Kim for their endless love and support. Without their patience, I could not have been studying in foreign country for my graduate study. Last but not least, thank God for giving me a fortunate chance to study at prestigious MIT.

Most of the work in this thesis was published as "Making Linux Protection Mechanisms Egalitarian with UserFS" in *Proceedings of the 19th Usenix Security Symposium.*

Contents

1 Introduction **7**

2 Motivation and Goals **10**

3 Kernel Interface Design **13**

3.1 User ID allocation 13

3.2 Restriction mechanisms 20

4 Implementation **24**

5 Applying UserFS **26**

5.1 DokuWiki 26

5.2 Command-line tools 29

5.3 User authentication 30

5.4 Chromium browser 31

6 Evaluation **32**

6.1 Kernel security 32

6.2 Application security 33

6.3 Performance 34

7 Related Work **36**

8 Limitation and Future Work **40**

9 Conclusion **43**

Chapter 1

Introduction

OS protection mechanisms are key to mediating access to OS-managed resources, such as the file system, the network, or other physical devices. For example, system administrators can use Unix user IDs to ensure that different users cannot corrupt each other's files; they can set up a `chroot` jail to prevent a web server from accessing unrelated files; or they can create firewall rules to control network access to their machine. Most operating systems provide a range of such mechanisms that help administrators enforce their security policies.

While these protection mechanisms can enforce the administrator's policy, many applications have their own security policies for OS-managed resources. For instance, an email client may want to execute suspicious attachments in isolation, without access to the user's files; a networked game may want to configure a firewall to make sure it does not receive unwanted network traffic that may exploit a vulnerability; and a web browser may want to precisely control what files and devices (such as a video camera) different sites or plugins can access. Unfortunately, typical OS protection mechanisms are only accessible to the administrator: an ordinary Unix user cannot allocate a new user ID, use `chroot`, or change firewall rules, forcing applications to invent their own protection techniques like system call interposition [15], binary rewriting [31] or analysis [13, 48], or interposing on system accesses in a language runtime like Javascript.

This paper presents the design of UserFS, a kernel framework that allows any application to use traditional OS protection mechanisms on a Unix system, and a prototype implementation of UserFS for Linux. UserFS makes protection mechanisms *egalitarian*, so that any user—not just the system administrator—can allocate new user IDs, set up firewall rules, and isolate processes using `chroot`. By using the operating system's own protection mechanisms, applications can avoid race conditions and ambiguities associated with system call interposition [14, 45], can confine existing code without having to recompile or rewrite it in a new language, and can enforce a coherent security policy for large applications that might span several runtime environments, such as both Javascript and Native Client [48], or Java and JNI code.

Allowing arbitrary users to manipulate OS protection mechanisms through UserFS requires addressing several challenges. First, UserFS must ensure that a malicious user cannot exploit these mechanisms to violate another application's security policy, perhaps by re-using a previously allocated user ID, or by running setuid-root programs in a malicious `chroot` environment. Second, user IDs are often used in Unix for accountability and auditing, and UserFS must ensure that a system administrator can attribute actions to users that he or she knows about, even for processes that are running with a newly-allocated user ID. Finally, UserFS should to be compatible with existing applications, interfaces, and kernel components whenever possible, to make it easy to incrementally deploy UserFS in practical systems.

UserFS addresses these challenges with a few key ideas. First, UserFS allows applications to allocate user IDs that are indistinguishable from traditional user IDs managed by the system administrator. This ensures that existing applications do not need to be modified to support application-allocated protection domains, and that existing UID-based protection mechanisms like file permissions can be reused. Second, UserFS maintains a shadow generation number associated with each user ID, to make sure that setuid executables for a given UID cannot be used to obtain privileges once the UID has been reused by a new application. Third, UserFS represents allocated user IDs using files in a special file system. This makes it easy to manipulate user IDs, much like using the `/proc` file system on Linux, and applications can use file descriptor

passing to delegate privileges and implement authentication logic. Finally, UserFS uses information about what user ID allocated what other user IDs to determine what setuid executables can be trusted in any given `chroot` environment, as will be described later.

We have implemented a prototype of UserFS for Linux purely as a kernel module, consisting of less than 3,000 lines of code, along with user-level support libraries for C and PHP-based applications. UserFS imposes no performance overhead for most existing operations, and only performs an additional check when running setuid executables. We modified several applications to enforce security policies using UserFS, including Google's Chromium web browser, a PHP-based wiki application, an FTP server, `ssh-agent`, and Unix commands like `bash` and `su`, all with minimal code modifications, suggesting that UserFS is easy to use. We further show that our modified wiki is not vulnerable by design to 5 out of 6 security vulnerabilities found in that application over the past several years.

The key contribution of this work is the first system that allows Linux protection and isolation mechanisms to be freely used by non-root code. This improves overall security both by allowing applications to enforce their policies in the OS, and by reducing the amount of code that needs to run as root in the first place (for example to set up `chroot` jails, create new user accounts, or configure firewall rules).

The rest of this paper is structured as follows. Section 2 provides more concrete examples of applications that would benefit from access to OS protection mechanisms. Section 3 describes the design of UserFS in more detail, and Section 4 covers our prototype implementation. We illustrate how we modified existing applications to take advantage of UserFS in Section 5, and Section 6 evaluates the security and performance of UserFS. Section 7 surveys related work, Section 8 discusses the limitations of our system, and Section 9 concludes.

Chapter 2

Motivation and Goals

The main goal of UserFS is to help applications reduce the amount of trusted code, by allowing them to use traditionally privileged OS protection mechanisms to control access to system resources, such as the file system and the network. We believe this will allow many applications to improve their security, by preventing compromises where an attacker takes advantage of an application's excessive OS-level privileges. However, UserFS is not a security panacea, and programmers will still need to think about a wide range of other security issues from cryptography to cross-site scripting attacks. The rest of this section provides several motivating examples in which UserFS can improve security.

Avoiding root privileges in existing applications

Typical Unix systems run a large amount of code as root in order to perform privileged operations. For example, network services that allow user login, such as an FTP server, `sshd`, or an IMAP server often run as root in order to authenticate users and invoke `setuid()` to acquire their privileges on login. Unfortunately, these same network services are the parts of the system most exposed to attack from external adversaries, making any bug in their code a potential security vulnerability. While some attempts have been made to privilege-separate network services, such as with OpenSSH [41], it requires carefully re-designing the application and explicitly moving state between privileged and unprivileged components. By allowing processes to

explicitly manipulate Unix users as file descriptors, and pass them between processes, UserFS eliminates the need to run network services as the root user, as we will show in Section 5.3.

In addition to network services, users themselves often want to run code as root, in order to perform currently-privileged operations. For instance, `chroot` can be useful in building a complex software package that has many dependencies, but unfortunately `chroot` can only be invoked by root. By allowing users to use a range of mechanisms currently reserved for the system administrator, UserFS further reduces the need to run code as root.

Sandboxing untrusted code

Users often interact with untrusted or partially-trusted code or data on their computers. For example, users may receive attachments via email, or download untrusted files from the web. Opening or executing these files may exploit vulnerabilities in the user's system. While it's possible for the mail client or web browser to handle a few types of attachments (such as HTML files) safely, in the general case opening the document will require running a wide range of existing applications (e.g. OpenOffice for Word files, or Adobe Acrobat to view PDFs). These helper applications, even if they are not malicious themselves, might perform undesirable actions when viewing malicious documents, such as a Word macro virus or a PDF file that exploits a buffer overflow in Acrobat.

Guarding against these problems requires isolating the suspect application from the rest of the system, while providing a limited degree of sharing (such as initializing Acrobat with the user's preferences). With UserFS, the mail client or web browser can allocate a fresh user ID to view a suspicious file, and use firewall rules to ensure the application does not abuse the user's network connection (e.g. to send spam), and Section 5.2 will describe how UserFS helps Unix users isolate partially-trusted or untrusted applications in this manner.

Enforcing separation in privilege-separated applications

One approach to building high-security applications is to follow the principle of least privilege [42] by breaking up an application into several components, each of which has the minimum privileges necessary. For instance, OpenSSH [41], qmail [3], and the Chromium browser [2] follow this model, and tools exist to help programmers privilege-separate existing applications [7]. One problem is that executing components with less privileges requires either root privilege to start with (and applications that are not fully-trusted to start with are unlikely to have root privileges), or other complex mechanisms. With UserFS, privilege-separated applications can use existing OS protection primitives to enforce isolation between their components, without requiring root privileges to do so. We hope that, by making it easier to execute code with less privileges, UserFS encourages more applications to improve their security by reducing privileges and running as multiple components. As an example, Section 5.4 shows how UserFS can isolate different processes in the Chromium web browser.

Exporting OS resources in higher-level runtimes

Finally, there are many higher-level runtimes running on a typical desktop system, such as Javascript, Flash, Native Client [48], and Java. Applications running on top of these runtimes often want to access underlying OS resources, including the file system, the network, and local devices such as a video camera. This currently forces the runtimes to implement their own protection schemes, e.g. based on file names, which can be fragile, and worse yet, enforce different policies depending on what runtime an application happens to use. By using UserFS, runtimes can delegate enforcement of security checks to the OS kernel, by allocating a fresh user ID for logical protection domains managed by the runtime. For example, Section 5.1 shows how UserFS can enforce security policies for a PHP web application. In the future, we hope the same mechanisms can be used to implement a coherent security policy for one application across all runtimes that it might use.

Chapter 3

Kernel Interface Design

To help applications reduce the amount of trusted code, UserFS allows any application to *allocate new principals*; in Unix, principals are user IDs and group IDs. An application can then enforce its desired security policy by first allocating new principals for its different components, then, second, setting file permissions—i.e., read, write, and execute privileges for principals—to match its security policy, and finally, running its different components under the newly-allocated principals.

A slight complication arises from the fact that, in many Unix systems, there are a wide range of resources available to all applications by default, such as the `/tmp` directory or the network stack. Thus, to restrict untrusted code from accessing resources that are accessible by default, UserFS also allows applications to *impose restrictions* on a process, in the form of `chroot` jails or firewall rules. The rest of this section describes the design of the UserFS kernel mechanisms that provide these features.

3.1 User ID allocation

The first function of UserFS is to allow any application to allocate a new principal, in the form of a Unix user ID. At a naïvely high level, allocating user IDs is easy: pick a previously unused user ID value and return it to the application. However, there are four technical challenges that must be addressed in practice:

- When is it safe for a process to exercise the privileges of another user ID, or to change to a different UID? Traditional Unix provides two extremes, neither of which are sufficient for our requirements: non-root processes can only exercise the privileges of their current UID, and root processes can exercise everyone's privileges.

- How do we keep track of the resources associated with user IDs? Traditional Unix systems largely rely on UIDs to attribute processes to users, to implement auditing, and to perform resource accounting, but if users are able to create new user IDs, they may be able to evade UID-based accounting mechanisms.

- How do we recycle user ID values? Most Unix systems and applications reserve 32 bits of space for user ID values, and an adversary or a busy system can quickly exhaust 2^{32} user ID values. On the other hand, if we recycle UIDs, we must make sure that the previous owner of a particular UID cannot obtain privileges over the new owner of the same UID value.

- Finally, how do we keep user ID allocations persistent across reboots of the kernel?

We will now describe how UserFS addresses these challenges, in turn.

3.1.1 Representing privileges

UserFS represents user IDs with *files* that we will call *Ufiles* in a special `/proc`-like file system that, by convention, is mounted as `/userfs`. Privileges with respect to a specific user ID can thus be represented by *file descriptors* pointing to the appropriate Ufile. Any process that has an open file descriptor corresponding to a Ufile can issue a `USERFS_IOC_SETUID` *ioctl* on that file descriptor to change the process's current UID (more specifically, euid) to the Ufile's UID.

Aside from the special *ioctl* calls, file descriptors for Ufiles behave exactly like any other Unix file descriptor. For instance, an application can keep multiple file descriptors for different user IDs open at the same time, and switch its process UID

back and forth between them. Applications can also use file descriptor passing over Unix domain sockets to pass privileges between processes. This can be useful in implementing user authentication or login, by allowing an authentication daemon to accept login requests over a Unix domain socket, and to return a file descriptor for that user's Ufile if the supplied credential (e.g. password) was correct.

Finally, each Ufile under `/userfs` has an owner user and group associated with it, along with user and group permissions. These permissions control what other users and groups can obtain the privileges of a particular UID by opening its via path name. By default, a Ufile is owned by the user and group IDs of the process that initially allocated that UID, and has Unix permissions 600 (i.e. accessible by owner, but not by group or others), allowing the process that allocated the UID to access it initially. A process can always access the Ufile for the process's current UID, regardless of the permissions on that Ufile (this allows a process to always obtain a file descriptor for its current UID and pass it to others via FD passing).

3.1.2 Accountability hierarchy

Ufiles help represent privileges over a particular user ID, but to provide accountability, our system must also be able to say what user is responsible for a particular user ID. This is useful for accounting and auditing purposes: tracking what users are using disk space, running CPU-intensive processes, or allocating many user IDs via UserFS, or tracking down what user tried to exploit some vulnerability a week ago.

To provide accountability, UserFS implements a hierarchy of user IDs. In particular, each UID has a parent UID associated with it. The parent UID of existing Unix users is root (0), including the parent of root itself. For dynamically-allocated user IDs, the parent is the user ID of the process that allocated that UID (which in turn has its own parent UID). UserFS represents this UID hierarchy with directories under `/userfs`, as illustrated in Figure 3-1. For convenience, UserFS also provides symbolic links for each UID under `/userfs` that point to the hierarchical name of that UID, which helps the system administrator figure out who is responsible for a particular UID.

Path name	Role
/userfs/ctl	Ufile for root (UID 0).
/userfs/1001/ctl	Ufile for user 1001 (parent UID 0).
/userfs/1001/5001/ctl	Ufile for user 5001 (allocated by parent UID 1001).
/userfs/1001/5001/5002/ctl	Ufile for user 5003 (allocated by parent UID 5001).
/userfs/1001/5003/ctl	Ufile for user 5003 (allocated by parent UID 1001).
/userfs/1002/ctl	Ufile for user 1002 (parent UID 0).
/userfs/5001	Symbolic link to 1001/5001.
/userfs/5002	Symbolic link to 1001/5001/5002.
/userfs/5003	Symbolic link to 1001/5003.

Figure 3-1: An overview of the files exported via UserFS in a system with two traditional Unix accounts (UID 1001 and 1002), and three dynamically-allocated accounts (5001, 5002, and 5003). Not shown are system UIDs that would likely be present on any system (users such as bin, nobody, etc), or directories that are implied by the ctl files. Each ctl file supports two *ioctls*: USERFS_IOC_SETUID and USERFS_IOC_ALLOC.

In addition to the USERFS_IOC_SETUID ioctl that was mentioned earlier, UserFS supports three more operations. First, a process can allocate new UIDs by issuing a USERFS_IOC_ALLOC ioctl on a Ufile. This allocates a new UID as a child of the Ufile's UID, and the value of the newly allocated UID is returned as the result of the ioctl. A process can also de-allocate UIDs by performing an rmdir on the appropriate directory under /userfs. This will recursively de-allocate that UID and all of its child UIDs (i.e. it will work even on non-empty directories), and kill any processes running under those UIDs, for reasons we will describe shortly. Finally, a process can move a UID in the hierarchy using rename (for example, if one user is no longer interested in being responsible for a particular UID, but another user is willing to provide resources for it).

Finally, accountability information may be important long after the UID in question has been de-allocated (e.g. the administrator wants to know who was responsible for a break-in attempt, but the UID in the log associated with the attempt has been de-allocated already). To address this problem, UserFS uses syslog to log all allocations, so that an administrator can reconstruct who was responsible for that UID at any point in time.

3.1.3 UID reuse

An ideal system would provide a unique identifier to every principal that ever existed. Unfortunately, most Unix kernel data structures and applications only allocate space for a 32-bit user ID value, and an adversary can easily force a system to allocate 2^{32} user IDs. To solve this problem, UserFS associates a 64-bit *generation number* with every allocated UID[1], in order to distinguish between two principals that happen to have had the same 32-bit UID value at different times. The kernel ensures that generation numbers are unique by always incrementing the generation number when the UID is deallocated. However, as we just mentioned, there isn't enough space to store the generation number along with the user ID in every kernel data structure. UserFS deals with this on a case-by-case basis:

Processes

UserFS assumes that the current UID of a process always corresponds to the latest generation number for that UID. This is enforced by killing every process whose current UID has been deallocated.

Open Ufiles

serFS keeps track of the generation number for each open file descriptor of a Ufile, and verifies that the generation number is current before proceeding with any ioctl on that file descriptor (such as `USERFS_IOC_SETUID`). Once a UID has been reused, the current UID generation number is incremented, and leftover file descriptors for the old Ufile will be unusable. This ensures that a process that had privileges over a UID in the past cannot exercise those privileges once the UID is reused.

Setuid files

Setuid files are similar to a file descriptor for a Ufile, in the sense that they can be used to gain the privileges of a UID. To prevent a stale setuid file from being used

[1] It would take an attacker thousands of years to allocate 2^{64} UIDs, even at a rate of 1 million UIDs per second.

to start a process with the same UID in the future, UserFS keeps track of the file owner's UID generation number for every setuid file in that file's extended attributes. (Extended attributes are supported by many file systems, including ext2, ext3, and ext4. Moreover, small extended attributes, such as our generation number, are often stored in the inode itself, avoiding additional seeks in the common case.) UserFS sets the generation number attribute when the file is marked setuid, or when its owner changes, and checks whether the generation number is still current when the setuid file is executed.

Non-setuid files, directories, and other resources

UserFS does not keep track of generation numbers for the UID owners of files, directories, system V semaphores, and so on. The assumption is that it's the previous UID owner's responsibility to get rid of any data or resources they do not want to be accessed by the next process that gets the same UID value. This is potentially risky, if sensitive data has been left on disk by some process, but is the best we have been able to do without changing large parts of the kernel.

There are several ways of addressing the problem of leftover files, which may be adopted in the future. First, the on-disk inode could be changed to keep track of the generation number along with the UID for each file. This approach would require significant changes to the kernel and file system, and would impose a minor runtime performance overhead for all file accesses. Second, the file system could be scanned to find orphaned files, much in the same way that UserFS scans the process table to kill processes running with a deallocated UID. This approach would make user deallocation expensive, although it would not require modifying the file system itself. Finally, each application could run sensitive processes with write access to only a limited set of directories, which can be garbage-collected by the application when it deletes the UID. Since none of the approaches are fully satisfactory, our design leaves the problem to the application, out of concern that imposing any performance overheads or extensive kernel changes would preclude the use of UserFS altogether.

3.1.4 Persistence

UserFS must maintain two pieces of persistent state. First, UserFS must make sure that generation numbers are not reused across reboot; otherwise an attacker could use a setuid file to gain another application's privileges when a UID is reused with the same generation number. One way to achieve this would be to keep track of the last generation number for each UID; however this would be costly to store. Instead, UserFS maintains generation numbers only for allocated UIDs, and just one "next" generation number representing all un-allocated UIDs. UserFS increments this next generation number when any UID is allocated or deallocated, and uses its current value when a new UID is allocated. To ensure that generation numbers are not reused in the case of a system crash, UserFS synchronously increments the next generation number on disk. As an important optimization, UserFS batches on-disk increments in groups of 1,000 (i.e., it only update the on-disk next generation number after 1,000 increments), and it always increments the next generation counter by 1,000 on startup to account for possibly-lost increments.

Second, UserFS must allow applications to keep using the same dynamically-allocated UIDs after reboot (e.g. if the file system contains data and/or setuid files owned by that UID). This involves keeping track of the generation number and parent UID for every allocated UID, as well as the owner UID and GID for the corresponding Ufile. UserFS maintains a list of such records in a file (`/etc/userfs_uid`), as shown in Figure 3-2. The permissions for the Ufile are stored as part of the owner value (if the owner UID or GID is zero, the corresponding permissions are 0, and if the owner UID or GID is non-zero, the corresponding permissions are read-write). The generation numbers of the parent UID, owner UID, and owner GID are not tracked; the parent UID is necessarily current (otherwise this child would have been deallocated), and the owner UID and GID are left up to the Ufile owner.

UserFS lazily updates this on-disk data structure; deletion is implemented in-place by setting the UID value to -1. If an application wants to rely on the Ufile being

UID	Parent UID	Generation number	Owner UID	Owner GID
32 bits	32 bits	64 bits	32 bits	32 bits

Figure 3-2: Record stored by UserFS on disk for each allocated UID, totaling 24 bytes per allocated UID.

present after reboot, it can force that Ufile's persistent record to be written to disk by issuing an `fsync` on the Ufile's file descriptor.

As an optimization, UserFS also allows non-persistent UIDs to be allocated (for isolating processes that do not store any persistent data in the file system under their UID). To implement this, the `USERFS_IOC_ALLOC` ioctl takes one argument that indicates whether the new UID should be persistent or not; persistent UIDs can only be allocated to persistent parents.

As a practical matter, UserFS partitions the 32-bit UID space into UIDs reserved for system use (0 through $2^{30} - 1$), persistent dynamically-allocated UIDs (2^{30} through $2^{31} - 1$), non-persistent dynamically-allocated UIDs (2^{31} through $2^{31} + 2^{30} - 1$), and more reserved UIDs ($2^{31} + 2^{30}$ through $2^{32} - 1$). This makes it easy to determine whether a particular UID is persistent, and avoids conflicts with most system-allocated UIDs at either end of the UID number space. UserFS provides modified `adduser` and `deluser` programs that create and delete Ufiles when they add or remove users from the system (to allow those users to allocate new UIDs via ioctls on their Ufile), and assumes that the system administrator will not use UIDs in the dynamically-allocated range.

3.2 Restriction mechanisms

To prevent malicious code from accessing resources that are accessible to everyone by default (such as `/tmp` or the network), UserFS allows applications to take advantage of existing restriction mechanisms: `chroot` to limit access to the file system namespace, and firewall rules to limit access to the network.

3.2.1 File system namespaces

To prevent processes from accessing files that are accessible by default, UserFS allows any user to invoke `chroot`. There are two potential problems associated with this: setuid programs that will behave incorrectly in a `chroot` environment, and arbitrary programs attempting to escape from a `chroot` jail by recursive use of `chroot` itself.

Setuid programs

If a setuid program runs in a `chroot` environment, it can behave in unpredictable ways—for instance, a setuid-root `su` program may read a user-supplied `/etc/passwd` file and grant the caller root access because it assumed that root's password in its version of `/etc/passwd` was authentic. UserFS relies on the user ID hierarchy to address this problem. In particular, after user U calls `chroot`, UserFS will only honor setuid bits for files owned by UIDs that are descendants of U. In the corner case of root invoking `chroot`, every user is a descendant of root, and thus every setuid program will still be honored, as on a regular Linux system.

UserFS only keeps track of the last UID to call `chroot` for a given process (inherited across fork). If one user performs `chroot` inside a second user's jail, it is the responsibility of the first user to verify that it's creating a `chroot` environment acceptable to all of its descendants. In practice, we expect that the first user will be a descendant of the second user (because he is executing inside the second user's jail), so this requirement will not pose significant problems.

Escaping chroot

The Linux `chroot` mechanism works by effectively maintaining a single "barrier" at the specified root directory that prevents the process from evaluating `..` (parent directory) of that process's root directory. A process can escape a `chroot` jail by obtaining a reference (either a file descriptor or current working directory) to a directory outside the `chroot`'ed hierarchy, and using that reference to walk up the `..` pointers to the true file system root. Even if an application properly uses `chroot` to confine a process,

the kernel only keeps track of one root directory pointer per process, so a malicious process in a `chroot` jail could confine itself to a second `chroot` jail while maintaining a handle on a directory outside this second jail, and use that handle to escape both jails.

To prevent this problem, UserFS enforces three rules for `chroot` invoked by non-root users. First, to ensure a process cannot maintain a current working directory outside the `chroot` environment, UserFS requires that `chroot` callers set their directory to the `chroot` target directory ahead of time. Second, UserFS checks that a process calling `chroot` has no open directory file descriptors. Finally, UserFS ensures that a process cannot receive a directory file descriptor via file descriptor passing from outside the jail: it annotates Unix domain sockets with the sender's root directory (or a "prohibited" value if there are senders with different root directories) on `sendmsg`, and checks that the sender's root directory matches the recipient process root directory on `recvmsg`, if the message contains a directory file descriptor.

3.2.2 Firewall rules

Ideally, we would like users to be able to run a process with a set of firewall rules attached to it, and for those firewall rules to apply to any child processes spawned by that process, much in the same way that `chroot` applies to all child processes. Unfortunately, this would require changing the core Linux kernel: at the very least, it would be necessary to track the "current firewall ruleset" for each process. Since we wanted to implement UserFS purely in terms of loadable kernel modules, we compromised, and associated firewall rules with UIDs instead. The kernel already keeps track of the UID for each process, and propagates the UID to the children of that process, so UserFS simply needs to ensure that firewall rules for newly-allocated UIDs inherit the firewall rules for the parent UID.

UserFS's firewall system consists of rules, which form rulesets, which are in turn associated with UIDs. At the lowest level, rules are of the form ⟨*action*, *proto*, *address*, *netmask*, *port*⟩. Our prototype supports two kinds of actions, ALLOW and BLOCK, and two protocols, TCP and UDP. The protocol, address, netmask, and port are matched

against the destination of outgoing packets or the source of incoming packets; port value 0 matches any port. Supporting just TCP and UDP protocols suffices because, on Linux, a non-root process cannot open a raw socket to send arbitrary packets that are neither TCP or UDP. For kernels that support other protocols, such as SCTP, UserFS's rules could be augmented to track additional protocols.

A ruleset is an ordered sequence of rules, used to determine whether a packet should be allowed or blocked. When checking a packet against a ruleset, UserFS finds the earliest rule in the ruleset that matches the packet, and uses that rule's action to determine if the packet should be allowed or blocked. Each ruleset contains two implicit rules at the end, ⟨ALLOW, TCP, 0.0.0.0, 0.0.0.0, 0⟩ and ⟨ALLOW, UDP, 0.0.0.0, 0.0.0.0, 0⟩, which allow any packets by default. Each UID is associated with a ruleset, and applications can modify that UID's ruleset by adding or removing rules as necessary.

One potential worry in associating rulesets with a UID is that a malicious process can create a child UID with less-restrictive firewall rules. To mitigate this problem, UserFS checks not only the UID's own firewall ruleset, but also the rulesets of all parent UIDs, and only allows packets if they are allowed by every ruleset in this chain.

UserFS provides a Ufile ioctl to add or remove rules from that UID's firewall ruleset. However, there is a slight complication: on the one hand, we want to ensure that a process cannot modify its own firewall ruleset, but on the other hand, a process can always open its own Ufile. To address this problem, UserFS allows the firewall ioctl to be invoked only by the parent UID of a Ufile. This ensures that a process cannot change firewall rules for itself through its own Ufile.

Chapter 4

Implementation

We have implemented UserFS as a kernel module for version 2.6.31 of the Linux kernel. The UserFS kernel module comprises a little less than 3,000 lines of code, excluding unit tests and the user-space `mount.userfs` command. UserFS relies heavily on the LSM framework [47] for checking generation numbers on setuid files (using file_permission and inode_setattr hooks), for confining `chroot` processes (using socket_sendmsg and socket_recvmsg hooks), and on netfilter for implementing network filtering (using `NF_INET_LOCAL_IN` and `NF_INET_LOCAL_OUT` hooks). UserFS also adds support to allow a process to `chown` or `chgrp` files between different UIDs that the process has privileges over.

Because UserFS is implemented as a kernel module, and does not modify core kernel code, it makes some trade-offs. For example, the kernel's versions of `chown`, `chgrp`, and `chroot` are not flexible enough for UserFS to implement its desired security policy from a kernel module. As a workaround, UserFS provides ioctls that implement equivalent functionality with its own security policy. Integrating UserFS into the core kernel code would both simplify our implementation and offer a more coherent interface to applications.

We have also implemented helper libraries for applications using UserFS, for both C and PHP. The C library comprises about 1,500 lines of code, including functions to execute a program in a newly-allocated jail and under a fresh user ID, to fork with a new UID, and to manipulate user IDs. The C library is careful to open all Ufiles

with the O_CLOEXEC flag to avoid accidentally leaking Ufile file descriptors to other processes. The PHP library adds about 600 more lines on top of the C library to allow PHP applications to manipulate Ufiles.

Chapter 5

Applying UserFS

To illustrate how UserFS would be used in practice, we modified several applications to take advantage of UserFS, including the Chromium web browser, the DokuWiki web application, Unix command-line utilities, and an FTP server. The rest of this section reports on these applications, focusing on the changes we had to make to each application in order to use UserFS, and the resulting benefits from doing so.

5.1 DokuWiki

Many web applications implement their own protection mechanisms, since they do not typically run as root, and thus cannot allocate user IDs for each application-level user. This can lead to vulnerabilities if the application developers make a mistake in performing security checks [9]. To show how UserFS can prevent similar problems, we modified DokuWiki [10], a wiki application written in PHP that supports read-protected and write-protected pages [11] and that stores wiki pages in the server's file system, to enforce the protection of wiki pages using file system permissions.

Our modified version of DokuWiki allocates a separate UID for each wiki user, and sets Unix permissions on wiki page files to reflect the protection of that page (we use ACL support in the ext4 file system [20] to represent ACLs that involve multiple users). To minimize the amount of damage that an attacker can do, our modified version of DokuWiki executes each HTTP request in a separate process, and allocates

a new ephemeral user ID for the initial processing of each request[1]. If an HTTP request provides the correct password for a user account, the DokuWiki PHP process handling that request can obtain a file descriptor for that user's Ufile, and change its UID to that user, by using the UserFS PHP module. This in turn allows a DokuWiki process to read or write wiki pages accessible to that user. Figure 5-1 shows the flow of an HTTP request in our modified DokuWiki.

One of the key parts of our modified DokuWiki is the login mechanism, which allows the DokuWiki process to obtain a file descriptor to a user's Ufile if it knows the user's password. We implemented this mechanism in a short C program called **dokusu**. **dokusu** accepts a username and password on stdin, checks the username and password against the password database, and if the password matches, it opens the corresponding user's Ufile (listed in the password database) and uses file descriptor passing to pass it back to the caller via stdout (which the caller should have set up as a Unix domain socket). **dokusu** is typically installed as a setuid program with the administrator's UID, and the permissions on all Ufiles for DokuWiki users in `/userfs` and on the password database are such that only the administrator can access them. Thus, to authenticate, DokuWiki spawns **dokusu**, passes it the username and password from the HTTP request, and waits for a Ufile in response.

DokuWiki keeps a copy of the user's password in its HTTP cookie, which makes it easy to authenticate subsequent requests. Cookies that store a session ID could also be supported, by augmenting **dokusu** to keep track of all currently valid session IDs and the corresponding user IDs for each session, and to accept a valid session ID as credentials for the corresponding user.

Making these changes to DokuWiki involved adding approximately 80 lines of PHP code, and implementing the 160-line **dokusu** program, on top of our UserFS PHP and C libraries, respectively. These changes allow the kernel to enforce DokuWiki's security policy, and Section 6.2 shows the effectiveness of this technique.

[1]We changed the first line of DokuWiki's PHP files to allocate a new ephemeral UID for each request, and to switch to that user ID. An alternative approach would be to modify the web server to launch each CGI script under a fresh user ID.

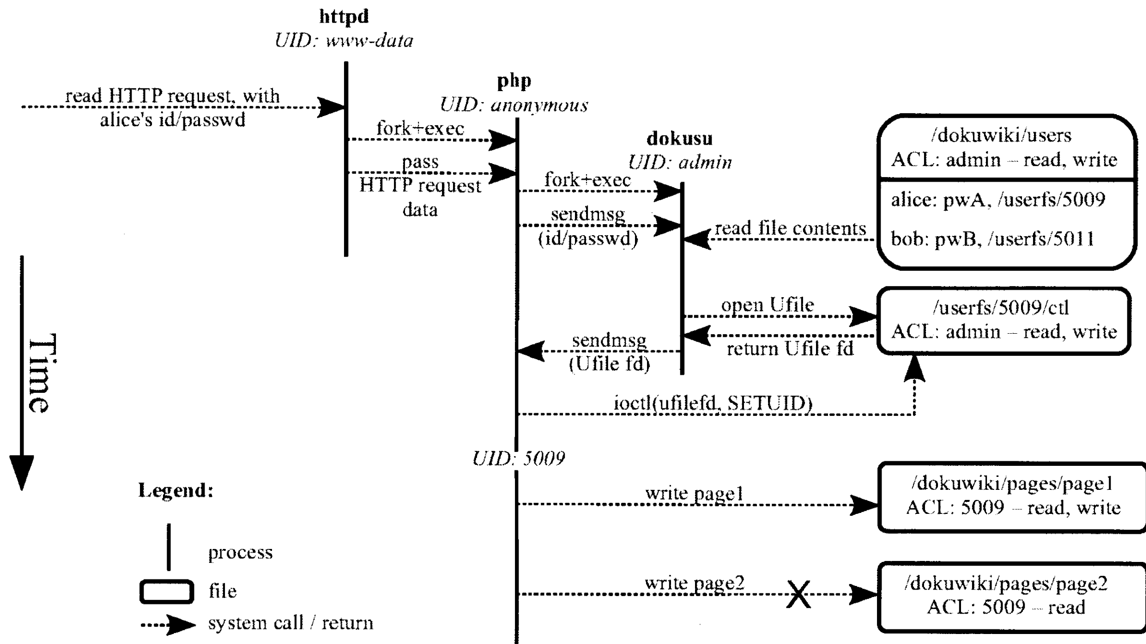

Figure 5-1: Flow of an HTTP request in our modified version of DokuWiki, showing Alice trying to write to two protected pages. Bold labels show process names (httpd, php, and dokusu). Italic labels show process UIDs (www-data, anonymous, admin, and 5009). After reading the users file, dokusu checks the supplied password against the stored password. In this example, Alice can modify page 1 (to which she has read-write access), but cannot modify page 2 (to which she has read-only access). In practice, Alice's UID would be a value between 2^{30} and $2^{31} - 1$, instead of 5009.

5.2 Command-line tools

To make it easy for ordinary users to use UserFS, we implemented a command to allocate a new user ID, called `ualloc`, which simply issues `USERFS_IOC_ALLOC` on the Ufile of the current process UID and prints the resulting UID value. To allow users to run code with these newly allocated UIDs, we modified `su` to allow users to be specified by their Ufile pathname instead of by username (in which case `su` relies on Ufile permissions to check if the caller is allowed to run as the target user, since it has no way of authenticating UserFS users by password). These modifications comprised approximately 300 lines of code.

With these changes, users can easily run arbitrary Unix applications with fewer privileges. For example, if a user wants to run a peer-to-peer file sharing program, but wants to avoid the risk of that program sharing private files with the rest of the world, the user can simply run `ualloc` to create a fresh UID for that program, run `su /userfs/`*`newuid`*`/ctl` to open a shell running as that user ID, and run the file sharing program from that shell. The file sharing program will not be able to read any of the user's private files (i.e., files that are not world-readable).

Users can also create processes that are isolated from the user's own account. For instance, `ssh-agent` stores a decrypted version of the user's SSH private key in memory. If an attacker compromises the user's account and finds a running `ssh-agent` process, the attacker can extract the key from memory by debugging `ssh-agent`. To prevent this, a user can allocate a fresh user ID with `ualloc`, run `ssh-agent` as that user ID, change permissions on the agent's socket so that the user can talk to `ssh-agent`[2], and finally change the owner of `ssh-agent`'s Ufile to `ssh-agent`'s UID, so that the user can no longer access it. The only thing the user can do at this point is to communicate with `ssh-agent` via the socket, or kill `ssh-agent` by deallocating the UID. The user cannot access `ssh-agent`'s memory to extract the key, since `ssh-agent` is running under a different UID, and the user cannot gain that UID's privileges, because it cannot open the corresponding Ufile.

[2]We had to make a two-line change to `ssh-agent` to support this, since by default `ssh-agent` refuses connections from other UIDs.

Finally, UserFS makes it easier for users to switch user IDs. With traditional `su`, the user receives a new shell running under the target UID, with a new working directory, new command history, and new environment variables. When the user wants to switch back to their original UID, they again lose their command history and environment variables. To show how UserFS can help, we modified `su` to support an option to pass the resulting Ufile back to the caller via FD passing, instead of running a shell under the resulting user's UID, and likewise modified `bash` to accept the Ufile FD from `su` (much like the design of `dokusu` in the previous subsection) and invoke `USERFS_IOC_SETUID` on it. This allows the user to switch UIDs without having to switch shell processes, improving user convenience.

5.3 User authentication

Many network services run as root in order to authenticate users and to invoke `setuid` to switch to that user's UID afterwards. Unfortunately, these network services are also some of the most vulnerable components in a system, since they are directly exposed to an attacker's inputs from the network, and if they are compromised, the attacker gains root access. With UserFS, network services like ftp, ssh, telnet, or IMAP mail servers can instead run as completely unprivileged processes[3], and perform authentication and login via Unix domain sockets like in DokuWiki above. (Infact, they can reuse the `su` command from the previous subsection, which passes back the authenticated user's Ufile to the caller.) This ensures that if an attacker finds a vulnerability in a network service, they get almost no privileges on the system. To prevent an attacker from subverting subsequent connections to a compromised service, a new service process should be forked, with a fresh non-persistent UID, for each connection.

To show this is feasible, we modified the Linux NetKit FTP server [23] to authenticate users using Ufile passing; doing this required 50 lines of code, indicating that

[3]We provide setuid-root binaries to open specific TCP ports below 1024, such as port 80 for the web server, accessible only to the web server's UID.

it is relatively easy to make such changes to existing applications (unlike privilege separation in the style of OpenSSH [41], which is much more invasive). Our modified FTP server uses the `su` program as its authentication agent.

5.4 Chromium browser

One application that is already broken up into many processes is Google's Chromium browser [2], which maintains a separate process for rendering each browser window, and a single browser kernel process responsible for coordinating with the rendering processes. This architecture easily lends itself to privilege separation, by isolating each rendering process. Indeed, Chromium already tries to do this on Windows using tokens [17], although this does not prevent a compromised browser process from accessing the network or world-accessible files.

With UserFS, browser processes can be isolated by allocating a fresh non-persistent UID for each rendering process, `chroot`ing the rendering process into an empty directory, and setting up firewall rules that block all network traffic. Making these changes to Chromium required replacing the `fork` call in Chromium with a call to a UserFS library function called `ufork` that performs precisely the actions mentioned above[4]. All communication between the browser kernel process and the rendering processes happens via sockets, which remain intact, while the kernel's protection mechanisms ensure that a compromised rendering process cannot access any files, signal any processes, or use the network.

[4]We do not provide a more fine-grained lines of code measure for the `ufork` function because it internally relies on most of the other functions provided by the UserFS library.

Chapter 6

Evaluation

To evaluate UserFS, we first discuss its security, then show how UserFS helps prevent attackers from exploiting vulnerabilities in DokuWiki, and then measure the performance overheads associated with UserFS.

6.1 Kernel security

The goal of UserFS is to allow any application to use the kernel's protection mechanisms. This implicitly assumes that the kernel's mechanisms are secure. While security vulnerabilities are found in the kernel from time to time [1], this paper does not attempt to tackle this problem, and assumes that, for the time being, users will continue to run applications on the Linux kernel.

Thus, we mostly focus on the security of any changes that UserFS makes to the Linux kernel. As a first-order measure, UserFS is relatively small—less than 3,000 lines of code—which simplifies the job of auditing our code. The specific mechanisms that UserFS provides that could be misused by adversaries are the `USERFS_IOC_SETUID` ioctl, allowing a process to switch user IDs, and the `chroot` mechanism that allows non-root processes to change their root directory.

We believe the `USERFS_IOC_SETUID` mechanism is secure because it only allows a process to switch user IDs if it has an open file descriptor to the corresponding Ufile. By default, each standard user's Ufile can only be opened by that user (and by root),

making it no different from the current kernel policy. Users can change permissions on Ufiles to allow other processes to open them, but again, a process can only change permissions on a Ufile that they already have access to (i.e. it was initially their UID, or it was granted to them). Applications can potentially make mistakes and leak privileges over a Ufile to another process by forgetting to close a Ufile file descriptor. The UserFS library tries to mitigate this by opening all Ufiles with the `O_CLOEXEC` flag.

The `chroot` mechanism could potentially be used recursively by an adversary to escape from a chroot jail. We believe that we have implemented sufficient safeguards against this, as described in Section 3.2.1, but we have no formal proof of their correctness.

6.2 Application security

Assuming UserFS and the Linux kernel are secure, we wanted to show what security benefits applications could extract from this. To do so, we decided to check whether any previously-reported vulnerabilities for DokuWiki would have been prevented by our changes to enforce the DokuWiki security policy using file system permissions. We found several vulnerabilities for DokuWiki in the past few years that allowed an attacker to compromise DokuWiki [33–38] (as opposed to information disclosure vulnerabilities, such as printing PHP debug information, which might help an attacker in exploiting another attack vector).

Our modified version of DokuWiki (backported to an older version of DokuWiki that contained the above vulnerabilities) was able to prevent exploits of code injection [36–38], directory traversal [34], and insufficient permission check [35] vulnerabilities (5 out of 6), but did not prevent exploits of a cross-site request forgery vulnerability [33]. Although our modified version of DokuWiki contained all of the above vulnerabilities, the vulnerable code was running with limited privileges (either the web server's ephemeral per-request UID, or the UID of a specific wiki user), which prevented the attack from doing any server-side damage.

Operation	Time without UserFS	Time with UserFS
Allocate new UID	—	0.022 ms
Check generation number [1]	—	0.003 ms
Run `sudo ls`	10.943 ms	10.946 ms
Fetch page from DokuWiki	45 ms	61 ms

Figure 6-1: Time taken to perform several operations with and without UserFS.

6.3 Performance

Performance of applications running on Linux with UserFS depends on two factors: overheads imposed by UserFS on system calls, and overheads associated with privilege-separating the application to make use of UserFS. In most cases, UserFS imposes no overheads on system calls, because the kernel executes the same exact access control checks based on UIDs with or without UserFS. One exception to this is the invocation of setuid binaries, for which UserFS checks the generation number of the setuid binary against the latest generation number for that UID. Applications that are modified to take advantage of UserFS incur two additional sources of overhead: the cost to invoke UserFS mechanisms, such as ioctls to allocate or change UIDs, and the cost of privilege-separating the application into separate Unix processes.

To evaluate these three sources of overhead, we used microbenchmarks to measure the cost of system calls affected by UserFS, and we used DokuWiki to measure the cost of privilege-separating an application with UserFS. Figure 6-1 shows the results of these experiments on a 2.8GHz Intel Core i7 system with 8GB RAM running a 64-bit Linux 2.6.31 kernel. As can be seen from the figure, UserFS imposes minimal overheads for both user allocation and for checking generation numbers on setuid binaries (which is dwarfed by the cost of forking a setuid program in the first place). In the case of DokuWiki, the performance overhead of privilege separation is largely dominated by the cost of spawning the **dokusu** authentication agent; we expect that having a long-running authentication agent that accepts requests over Unix domain sockets would significantly reduce the cost of running DokuWiki with UserFS. However, the

[1] It only applies to setuid executable programs

costs of privilege-separation are not specific to UserFS, and have been studied before extensively [2, 3, 5–7, 25, 27, 41].

Chapter 7

Related Work

The principle of least privilege [42] is generally recognized as a good strategy for building secure systems, and has been used by many applications in practice, including qmail [3], OpenSSH [41], OKWS [25], a number of web browsers [2, 19, 43], and others. Current Unix protection mechanisms make it difficult for non-root applications to follow the principle of least privilege, by not allowing them to create less-privileged principals. This requires developers that want less privileges to actually have more privileges by running as root, and UserFS directly addresses this problem.

It is well-known that reasoning about the safety of a computer system in the presence of setuid programs is difficult [22, 28], and there are many pitfalls in implementing safe setuid programs [4, 8]. For example, the privilege escalation vulnerabilities of the sudo setuid program have been reported every year since its first release in 1999 [39]. At the lowest level, UserFS does not make it any easier to write a correct setuid program. However, we hope that UserFS makes it possible for programs that currently run as root, including setuid-root programs, to run under a less privileged UID instead, mitigating the damage from any vulnerability.

Krohn argued that applications must be given mechanisms to reduce their privileges [26], and ServiceOS [44] similarly argues for support for application-level principals in the OS kernel. Capability-based systems like KeyKOS [6, 21], and DIFC systems like Asbestos [12] and HiStar [49], allow users to create new protection domains of their own, at the cost of requiring a new OS kernel. Flume [27] shows how these

ideas can be implemented on top of a Linux kernel to avoid the cost of re-implementing a new OS kernel, but Flume does not allow users to apply its protection mechanisms to unmodified existing applications. UserFS shows how the idea of egalitarian protection mechanisms can be realized in a standard Linux kernel, in a way that cleanly applies to most existing applications, and achieves many of the goals suggested by Krohn [26] and Wang [44].

The use of Ufile file descriptors to represent privileges over UIDs is inspired by capability systems [29]. Unlike traditional capability systems, which use capabilities to control access to all resources, UserFS only uses file descriptors to track the set of Ufiles currently held open by a process, and to pass Ufiles between processes. Initial access to Ufiles for opening the file descriptor, as well as access to all other resources, is controlled by Unix file permissions and other Unix mechanisms. One common problem facing capability systems is revocation of access. UserFS uses generation numbers to ensure that, once a UID has been reused, leftover file descriptors cannot gain access to that UID, since their generation numbers do not match the UID's generation number.

Although current Unix protection mechanisms are not egalitarian, many systems have used them to achieve privilege separation, at the cost of requiring some part of their system to run as root. For example, OKWS [25] shows how to build a privilege-separated web server by running a launcher as root, and Android [16] similarly uses Linux user IDs to isolate different applications on a cell phone. If these platforms start running increasingly more complex applications inside them, those applications will not have the benefit of running as root and creating their own protection domains. UserFS would address this problem.

Similarly, there have been a number of tools that help programmers privilege-separate their existing applications [5, 7, 41, 46]. The resulting privilege-separated applications often require root privileges to actually set up protection domains, and UserFS could be used in conjunction with these tools to run privilege-separated applications without root access.

System call interposition [15] could, in principle, implement any policy that a kernel could implement. By relying on the kernel's protection mechanisms, UserFS avoids

some of the pitfalls associated with system call interposition [14] and avoids runtime overhead for most operations. More importantly, UserFS illustrates what *interface* could be used by applications to allocate and manage their protection domains and set policies; the same interface could be implemented by a system call interposition system.

Bittau et al [5] propose a new kernel abstraction called an *sthread* that can execute certain pieces of an application's code in isolation from the rest of that application. The key contribution of *sthreads* was in providing a mechanism that has relatively low overhead for fine-grained isolation of process memory, and that can be used by any processes in the system. UserFS, on the other hand, provides persistent UIDs that can be used to control access to data in the file system, and to control interactions between multiple processes in an operating system.

The Linux kernel supports several security mechanisms in addition to traditional user ID protection, such as SELinux [30], Linux-vserver [40] and `seccomp` [18], but none of these mechanisms allow users to create their own protection domains and use them to protect system resources like files and devices. One protection mechanism that *is* available to users on Linux is running code in a virtual machine such as qemu. Unfortunately, this is often too coarse-grained and heavy-weight for most applications.

Taint tracking in an operating system can be used to implement certain application-level security policies; for example, SubOS [24] shows how this can be implemented on OpenBSD. Unfortunately, these mechanisms are much more invasive and impose more runtime overhead than UserFS, which simply exposes existing mechanisms in the OS kernel.

The protection mechanisms in Windows differ from those found in Unix systems. Windows protection is centered around the notion of tokens [32]. Users can create tokens that grant almost no privileges, and this is used by applications such as Chromium to sandbox untrusted code [17]. However, there is no way to create tokens with a fresh user ID (without administrative privileges to create a new user), which makes it difficult to implement controlled sharing of system resources (as opposed to complete isolation in a sandbox). Windows tokens can be passed between processes,

similar to how UserFS allows passing file descriptors for Ufiles. The Windows firewall allows associating firewall rules with executables. UserFS associates firewall rules with user IDs, and inherits firewall rules on user ID creation, which ensures that a user cannot escape firewall rules by creating and running a new executable.

Chapter 8

Limitation and Future Work

While UserFS helps applications run code with fewer privileges, it is not a panacea. Running untrusted code on a system often exposes a wider range of possibly-vulnerable interfaces than if we were simply interacting with the attacker over the network. For example, an attacker may try to exploit bugs in the kernel or in other applications running on the same machine. Nonetheless, if it is necessary to run untrusted or partially-trusted applications on a machine, UserFS helps improve security with respect to system resources.

UserFS, much like Linux itself, currently assumes that all file systems are always mounted on the same machine, and does not have a plan for translating UIDs from a file system that was originally mounted on a different machine. One possible approach to dealing with this problem may be to maintain a globally unique name of each UID (perhaps a public key), and to store on each file system a mapping table between file system UIDs and the globally unique names for those UIDs.

When a user ID is deallocated, it may be difficult to remove non-empty directories owned by that UID in the file system without root's intervention. While we have not yet implemented a solution to this problem, we imagine a system call or a setuid-root program that, upon request, recursively garbage-collects files or sub-directories owned by de-allocated UIDs from a given directory, as long as the caller has write permission on that directory.

UserFS only protects resources managed by the operating system, such as files, processes, and devices. Web applications often use databases to store their data, which UserFS cannot protect directly. In the future, we hope to explore the use of OS UIDs in a database to implement protection of data at a finer granularity (perhaps at the row level).

Our current prototype allocates user IDs, but does not separately allocate group IDs. We believe it is best to have only one kind of dynamically allocated principal, such as the 32-bit integer called the UID in UserFS. These principals can then be used to represent either users or groups, depending on the application's requirements. The GID and grouplist associated with every Unix process could then be used to represent a process that has the privileges of multiple principals at once. To support this, UserFS could provide a `USERFS_IOC_ADDGROUP` ioctl, which would add the Ufile's UID to the grouplist of the calling process. To avoid conflicts with existing groups, this ioctl should be only allowed for dynamically-allocated UIDs. In terms of file permissions, we also believe that POSIX ACLs [20] are a better alternative to the Unix user-group-other permission bits.

UserFS relies on the kernel to support 32-bit UIDs, as opposed to 16-bit UIDs from the original Unix design. Linux has supported 32-bit UIDs since kernel version 2.3.39 (January 2000), but UserFS cannot support older file systems that can only keep track of a 16-bit UID, such as the original Minix filesystem.

Our prototype faces several limitations because it is implemented as a loadable kernel module, and avoids making any extensive changes to the Linux kernel. For example, the `chroot` system call on Linux always rejects calls from non-root users, requiring UserFS to provide an alternative way of invoking `chroot`. Performing privileged operations in the kernel also requires UserFS to sometimes change the current UID of the calling process. While we believe our prototype does so safely, being able to change permission checks inside the core kernel code would be both simpler and more secure in the long term.

If UserFS was integrated into the Linux kernel, we would hope to extend our `chroot` mechanism to also allow arbitrary users to use the Linux file system namespace

mechanism (a generalization of the mount table). In particular, we want to allow any process to invoke *clone* with the `CLONE_NEWNS` flag to create a new namespace, and allow a process to change its namespace using `mount --bind` if it's running as the same UID that invoked `clone(CLONE_NEWNS)`, along with restrictions on setuid binaries similar to `chroot`. Similar support could also be added to allow users to manage the system V IPC namespace (`CLONE_NEWIPC`).

Finally, if UserFS was integrated into the Linux kernel, we would also like to replace our firewall mechanism with a per-process `iptables` firewall ruleset, inherited by child processes across `fork` and `clone`. To specify new firewall rules, applications would specify a new flag to the *clone* system call to start the child process with a fresh `iptables` ruleset. To ensure that a child cannot escape from the parent's firewall rules, the child's ruleset would be chained to the parent's.

Chapter 9

Conclusion

This paper presented UserFS, the first system to provide egalitarian OS protection mechanisms for Linux. UserFS allows any user to use existing OS protection mechanisms, including Unix user IDs, `chroot` jails, and firewalls. This both allows applications to reduce their privileges, and in many cases avoids the need for root privileges altogether.

One key idea in UserFS is representing user IDs as files in a `/proc`-like file system. This allows applications to manage user IDs much like they would any other file, without the need to introduce any new user ID management mechanisms. UserFS maintains a hierarchy of user IDs for accountability and resource revocation purposes, but allows child user IDs in the hierarchy to be made inaccessible to parent user IDs, in order to protect sensitive processes like `ssh-agent` from outside interference. To cope with a limited 32-bit user ID namespace, UserFS introduces per-UID generation numbers that disambiguate multiple instances of a reused 32-bit UID value. Finally, UserFS implements security checks that make it safe to allow non-root users to invoke `chroot`, without allowing users to escape out of existing `chroot` jails or abuse setuid executables.

An important goal of the UserFS design is compatibility with existing applications, interfaces, and kernel components. Porting applications to use UserFS requires only tens to hundreds of lines of code, and prevents attackers from exploiting application-level vulnerabilities, such as code injection or missing ACL checks in a PHP-based wiki

web application. UserFS requires minimal changes to the Linux kernel, comprising of a single 3,000-line kernel module, and incurs no performance overhead for most operations.

Bibliography

[1] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[2] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Google Inc., 2008.

[3] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007.

[4] Matt Bishop. How to write a setuid program. *;login: The Magazine of Usenix & Sage*, 12(1):5–11, January/February 1987.

[5] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation*, pages 309–322, San Francisco, CA, April 2008.

[6] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[7] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Usenix Security Symposium*, pages 57–72, San Diego, CA, August 2004.

[8] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.

[9] Michael Dalton, Nickolai Zeldovich, and Christos Kozyrakis. Nemesis: Preventing authentication and access control vulnerabilities in web applications. In *Proceedings of the 18th Usenix Security Symposium*, pages 267–282, Montreal, Canada, August 2009.

[10] DokuWiki. `http://www.dokuwiki.org/dokuwiki`.

[11] DokuWiki. Access control lists. `http://www.dokuwiki.org/acl`.

[12] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 2005.

[13] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[14] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[15] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2004.

[16] Google, Inc. Android: Security and permissions. http://developer.android.com/guide/topics/security/security.html.

[17] Google, Inc. Chromium sandbox. http://dev.chromium.org/developers/design-documents/sandbox.

[18] Google, Inc. Seccomp sandbox for linux. http://code.google.com/p/seccompsandbox.

[19] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 402–416, Oakland, CA, 2008.

[20] Andreas Grünbacher. POSIX access control lists on Linux. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX track*, pages 259–272, San Antonio, TX, June 2003.

[21] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating System Review*, 19(4):8–25, October 1985.

[22] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[23] David A. Holland. linux-ftpd. In *Linux NetKit*. ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/linux-ftpd-0.17.tar.gz.

[24] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan Smith. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, September 2002.

[25] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[26] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, M. Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart,

and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

[27] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 321–334, Stevenson, WA, October 2007.

[28] Tim Levin, Steven J. Padilla, and Cynthia E. Irvine. A formal model for UNIX setuid. In *Proceedings of the 10th IEEE Symposium on Security and Privacy*, pages 73–83, Oakland, CA, May 1989.

[29] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[30] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 29–40, June 2001. FREENIX track.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[32] Microsoft Corp. Access tokens (windows). `http://msdn.microsoft.com/en-us/library/aa374909%28VS.85%29.aspx`.

[33] MITRE Corporation. DokuWiki cross-site request forgery vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2010-0289.

[34] MITRE Corporation. DokuWiki directory traversal vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2010-0287.

[35] MITRE Corporation. DokuWiki insufficient permission checking vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2010-0288.

[36] MITRE Corporation. DokuWiki php code inclusion vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2009-1960.

[37] MITRE Corporation. DokuWiki php code injection vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2006-4674.

[38] MITRE Corporation. DokuWiki php code upload vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2006-4675.

[39] MITRE Corporation. Sudo group privilege escalation vulnerability. In *Common Vulnerabilities and Exposures (CVE) database.* CVE-2011-0010.

[40] Herbert Pötzl. *Linux-VServer Technology*, 2004. `http://linux-vserver.org/Linux-VServer-Paper`.

[41] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Usenix Security Symposium*, Washington, DC, August 2003.

[42] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[43] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle web browser. In *18th USENIX Security Symposium*, August 2009.

[44] Helen J. Wang, Alexander Moshchuk, and Alan Bush. Convergence of desktop and web applications on a multi-service OS. In *4th Usenix Workshop on Hot Topics in Security*, August 2009.

[45] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, Boston, MA, August 2007.

[46] Robert N. M. Watson, Jonathan Anderson, Kris Kennaway, and Ben Laurie. Capsicum: practical capabilities for unix. In *19th USENIX Security Symposium*, August 2010.

[47] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.

[48] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[49] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.