Setuid Demystified*

Hao Chen David Wagner
University of California at Berkeley
{hchen,daw}@cs.berkeley.edu

Drew Dean
SRI International
ddean@csl.sri.com

Abstract

Access control in Unix systems is mainly based on user IDs, yet the system calls that modify user IDs (*uid-setting system calls*), such as *setuid*, are poorly designed, insufficiently documented, and widely misunderstood and misused. This has caused many security vulnerabilities in application programs. We propose to make progress on the setuid mystery through two approaches. First, we study kernel sources and compare the semantics of the uid-setting system calls in three major Unix systems: Linux, Solaris, and FreeBSD. Second, we develop a formal model of user IDs as a Finite State Automaton (FSA) and develop new techniques for automatic construction of such models. We use the resulting FSA to uncover pitfalls in the Unix API of the uid-setting system calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check the proper usage of these calls in programs automatically. Finally, we provide general guidelines on the proper usage of the uid-setting system calls, and we propose a high-level API that is more comprehensible, usable, and portable than the usual Unix API.

1 Introduction

Access control in Unix systems is mainly based on the user IDs associated with a process. In this model, each process has a set of user IDs and group IDs which determine which system resources, such as files and network ports, the process can access[1]. Certain privileged user IDs and groups IDs allow a process to access restricted system resources. In particular, user ID zero, reserved for the superuser *root*, allows a process to access all system resources.

In some applications, a user process needs extra privileges, such as permission to read the password file. By the principle of least privilege, the process should drop its privileges as soon as possible to minimize risk to the system should it be compromised and execute malicious code. Unix systems offer a set of system calls, called the *uid-setting system calls*, for a process to raise and drop privileges. Such a process is called a *setuid process*. Unfortunately, for historical reasons, the uid-setting system calls are poorly designed, insufficiently documented, and widely misunderstood. "Many years after the inception of setuid programs, how to write them is still not well understood by the majority of people who write them" [1]. In short, the Unix setuid model is mysterious, and the resulting confusion has caused many security vulnerabilities.

We approach the setuid mystery as follows. First, we study the semantics of the uid-setting system calls by reading kernel sources. We compare and contrast the semantics among different Unix systems, which is useful for authors of setuid programs. In doing so, we found that manual inspection is tedious and error-prone. This motivates our second contribution: we construct a formal model to capture the behavior of the operating system and use it to guide our analysis. We will describe a new technique for building this formal model in an automated way. We have used the resulting formal model to more accurately understand the semantics of the uid-setting system calls, to uncover pitfalls in the Unix API of these calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check for the proper usage of user IDs in programs automatically.

Formal methods have gained a reputation as being im-

*This research was supported in part by DARPA Contract ECU01-401U subcontract 27-000765 and NSF CAREER 0093337.

[1] In many Unix systems, a process has also a set of *supplementary group IDs*, which are not closely related to the topic of this paper and which will not be discussed.

practical to apply to large software systems, so it may be surprising that we found formal methods so useful in our effort. We will show how our formal model enables many tasks that would otherwise be too error-prone or laborious to undertake. Our success comes from using lightweight techniques to answer a well-defined question about the system; we are *not* attempting to prove that a kernel is correct! Abstraction plays a major role in simplifying the system so that simple analysis techniques are sufficient.

This paper is organized as the follows. Section 2 discusses related work. Section 3 provides background on the user ID model. Section 4 reviews the evolution of the uid-setting system calls. Section 5 compares and contrasts the semantics of the uid-setting system calls in three major Unix systems. Section 6 describes the formal user ID model and its applications. Section 7 analyzes two security vulnerabilities caused by misuse of the uid-setting system calls. Section 8 provides guidelines on the proper usage of the uid-setting system calls and proposes a high-level API to the user ID model.

2 Related Work

Manual pages in Unix systems are the primary source of information on the user ID model for most programmers. See, for example, *setuid(2)* and *setgid(2)*. But unfortunately, they are often incomplete or even wrong (Section 6.4.1). Many books on Unix programming also describe the user ID model, such as Stevens' [2], but often they are specific to one Unix system or release, are outdated, or lack important details.

Bishop discussed security vulnerabilities in setuid programs [3]. His focus is on potential vulnerabilities that a process may be susceptible to once it gains privilege, while our focus is on how to gain and drop privilege confidently and securely. Unix systems have evolved and diversified a great deal since Bishop's work in 1987, and a big problem today is how to port setuid programs securely to various Unix systems.

3 User ID Model

This section provides background on the user ID model. Each user in a Unix system has a unique user ID. The user ID determines which system resources the user can access. In particular, user ID zero is reserved for the superuser *root* who can access all resources.

Each process has three user IDs: the *real user ID* (*real uid*, or *ruid*), the *effective user ID* (*effective uid*, or *euid*), and the *saved user ID* (*saved uid*, or *suid*). The real uid identifies the owner of the process, the effective uid is used in most access control decisions, and the saved uid stores a previous user ID so that it can be restored later. Similarly, a process has three group IDs: the *real group ID*, the *effective group ID*, and the *saved group ID*. In most cases, the properties of the group IDs parallel the properties of their user ID counterparts. For simplicity, we will focus on the user IDs and will mention the group IDs only when there is the potential for confusion or pitfalls. In Linux, each process has also an *fsuid* and an *fsgid* which are used for access control to the filesystem. The *fsuid* usually follows the value in the effective uid unless explicitly set by the *setfsuid* system call. Similarly, the *fsgid* usually follows the value in the effective gid unless explicitly set by the *setfsgid* system call. Since the *fsuid* and *fsgid* are Linux specific, we will not discuss them except when we point out an inconsistency in the handling of them in the Linux kernel.

When a process is created by *fork*, it inherits the three user IDs from its parent process. When a process executes a new file by *exec…*, it keeps its three user IDs unless the set-user-ID bit of the new file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the new file.

Since access control is based on the effective user ID, a process gains privilege by assigning a privileged user ID to its effective uid, and drops privilege by removing the privileged user ID from its effective uid. Privilege may be dropped either temporarily or permanently.

- To drop privilege temporarily, a process removes the privileged user ID from its effective uid but stores it in its saved uid. Later, the process may restore privilege by restoring the privileged user ID in its effective uid.

- To drop privilege permanently, a process removes the privileged user ID from all three user IDs. Thereafter, the process can never restore privilege.

4 History

Bell Laboratories filed a patent application on Dennis Ritchie's invention of a bit to specify that a program should execute with the permissions of its owner, rather than invoker, in 1973. The patent was granted in 1979 [4]. Thus, setuid programs and related system calls have existed through most of Unix history.

4.1 Early Unix

In early Unix systems, a process had two user IDs: the real uid and the effective uid. Only one system call, *setuid*, modified them according to the following rule: if the effective uid was zero, *setuid* set both the real uid and effective uid; otherwise, *setuid* could only set the effective uid to the real uid [1]. This model had the problem that a process could not temporarily drop the root privilege in its effective uid and restore it later. As Unix diverged into System V and BSD, each system solved the problem in a different way.

4.2 System V

System V added a new user ID called the saved uid to each process. Also added was a new system call, *seteuid*, whose rules were:

- If the effective uid was zero, *seteuid* could set the effective uid to any user ID.
- Otherwise, *seteuid* could set the effective uid to only the real uid or saved uid.

seteuid did not change the real uid or saved uid. Furthermore, System V modified *setuid* so that if the effective uid was not zero, *setuid* functioned as *seteuid* (changing only the effective uid); otherwise, *setuid* set all three user IDs.

4.3 BSD

4.2 BSD kept the real uid and effective uid but changed the system call from *setuid* to *setreuid*. Processes could then directly control both their user IDs, under the following rules:

- If the effective uid was zero, then the real uid and effective uid could be set to any user ID.
- Otherwise, either the real uid or the effective uid could be set to value of the other one.

Therefore, the *setreuid* system call enabled a process to swap the real uid and effective uid.

The POSIX standard [5] codified a new specification for the *setuid* call. In an attempt to be POSIX compliant, 4.4 BSD replaced 4.2 BSD's old *setreuid* model with the POSIX/System V style saved uid model. It modified *setuid* so that *setuid* set all three user IDs regardless of whether the effective uid of a process was zero, therefore allowing any process to permanently drop privileges.

4.4 Modern Unix

As System V and BSD influenced each other, both systems implemented *setuid*, *seteuid*, and *setreuid*, although with different semantics. None of these system calls, however, allowed the direct manipulation of the saved uid (although it could be modified indirectly through *setuid* and *setreuid*). Therefore, some modern Unix systems introduced a new call, *setresuid*, to allow the modification of each of the three user IDs directly.

5 Complexity of Uid-setting System Calls

A process modifies its user IDs by the uid-setting system calls: *setuid*, *seteuid*, *setreuid*, and in some systems, *setresuid*. Each of the system calls involves two steps. First, it checks if the process has permission to invoke the system call. If so, it then modifies the user IDs of the process according to certain rules.

In this section, we compare and contrast the semantics of the uid-setting system calls among Linux 2.4.18 [6], Solaris 8 [7], and FreeBSD 4.4 [8]. The behavior of the uid-setting system calls was discovered by a combination of manual inspection of kernel source code and formal methods. We will defer discussion of the latter until Section 6.

The POSIX Specification To understand the semantics of the uid-setting system calls, we begin with the POSIX standard, which has influenced the design of the system calls in many systems. In particular, the behavior of *setuid(newuid)* is defined by the POSIX specification. See Figure 1 for the relevant text.

The POSIX standard refers repeatedly to the term *appropriate privileges*, which is defined in Section 2.3 of POSIX 1003.1-1988 as:

> An implementation-defined means of associating privileges with a process with regard to the function calls and function call options defined in this standard that need special privileges. There may be zero or more such means.

Essentially, the term *appropriate privilege* serves as a wildcard that allows compliant operating systems to use any policy whatsoever for deeming when a call to *setuid* should be allowed. The conditional flag {`_POSIX_SAVED_IDS`} parametrizes the specification, allowing POSIX-compatible operating systems to use either of two schemes (as described in Figure 1). We will see how different interpretations of the term *appropriate privilege* have led to considerable differences in the behavior of the uid-setting system calls between operating systems.

> If {`_POSIX_SAVED_IDS`} is defined:
>
> 1. If the process has *appropriate privileges*, the *setuid*() function sets the real user ID, effective user ID, and the [saved user ID] to *newuid*.
> 2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID or the [saved user ID], the *setuid*() function sets the effective user ID to *newuid*; the real user ID and [saved user ID] remain unchanged by this function call.
>
> Otherwise:
>
> 1. If the process has *appropriate privileges*, the *setuid*() function sets the real user ID and effective user ID to *newuid*.
> 2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID, the *setuid*() function sets the effective user ID to *newuid*; the real user ID remains unchanged by this function call.
>
> (POSIX 1003.1-1988, Section 4.2.2.2)

Figure 1: An excerpt from the POSIX specification [5] covering the behavior of the *setuid* system call.

5.1 Operating System-Specific Differences

Much of the confusion is caused by different interpretations of *appropriate privileges* among Unix systems.

Solaris In Solaris 8, a System V based system, a process is considered to have *appropriate privileges* if its effective uid is zero (root). Also, Solaris defines {`_POSIX_SAVED_IDS`}. Consequently, calling *setuid(newuid)* sets all three user IDs to *newuid* if the effective uid is zero, but otherwise sets only the effective uid to *newuid* (if the *setuid* call is permitted).

FreeBSD FreeBSD 4.4 interprets *appropriate privileges* differently, as noted in Appendix B4.2.2 of POSIX:

> The behavior of 4.2BSD and 4.3BSD that allows setting the real ID to the effective ID is viewed as a value-dependent special case of *appropriate privilege*.

This means that a process is deemed to have *appropriate privileges* when it calls *setuid(newuid)* with *newuid=geteuid()*, in addition to when its effective uid is zero. Also in contrast to Solaris, FreeBSD does not define {`_POSIX_SAVED_IDS`}, although every FreeBSD process does have a saved uid. Therefore, by calling *setuid(newuid)*, a process sets both its real uid and effective uid to *newuid* if the system call is permitted, in agreement with POSIX. FreeBSD also sets the saved uid in all permitted *setuid* calls.

Linux Linux introduces a capability[2] model for finer-grained control of privileges. Instead of a single level of privilege determined by the effective uid (i.e., root or non-root), there are a number of capability bits each of which is used to determine access control to certain resources[3]. One of them, the *SETUID* capability, carries the POSIX *appropriate privileges*. To make the new ca-

[2]Beware: the word “capability” is a bit of a misnomer. In this context, it refers to special privileges that a process can possess, and not to the usual meaning in the security literature of an unforgeable reference. Regrettably, the former usage comes from the POSIX standard and seems to be in common use, and so we follow their convention in this paper.

[3]More accurately, a Linux process has three sets of capabilities, but only the set of *effective capabilities* determine access control. All references to *capabilities* in this paper refer to the effective capabilities.

pability model compatible with the traditional user ID model where *appropriate privileges* are carried by a zero effective uid, the Linux *SETUID* capability tracks the effective uid during all uid-setting system calls: Whenever the effective uid becomes zero, the *SETUID* capability is set; whenever the effective uid becomes non-zero, the *SETUID* capability is cleared.

However, the *SETUID* capability can be modified outside the uid-setting system calls. A process can clear its *SETUID* capability, and a process with the *SETPCAP* capability can remove the *SETUID* capability of other processes (but note that in Linux 2.4.18, no process has or can acquire the *SETPCAP* capability, a change that was made to close a security hole; see Section 7.1 for details). Therefore, explicitly setting or clearing the *SETUID* capability changes the properties of uid-setting systems calls.

5.2 Comparison among Uid-setting System Calls

Next we compare and contrast the uid-setting system calls and point out several unexpected properties and an inconsistency in the handling of *fsuid* in the Linux kernel.

setresuid() *setresuid* has the clearest semantics among the four uid-setting system calls. The permission check for *setresuid()* is intuitive and common to all OSs: for the *setresuid()* system call to be allowed, either the euid of the process must be root, or each of the three parameters must be equal to one of the three user IDs of the process. As each of the real uid, effective uid, and saved uid is set directly by *setresuid*, the programmer knows clearly what to expect after the call. Moreover, the *setresuid* call is guaranteed to have an all-or-nothing effect: if it succeeds, all user IDs are changed, and if it fails, none are; it will not fail after having changed some but not all of the user IDs.

Note that while FreeBSD and Linux offer *setresuid*, Solaris does not. However, Solaris does offer equivalent functionality via the `/proc` filesystem. Any process can examine its three user IDs, and a superuser process can set any of them, in line with the traditional System V notion of *appropriate privilege*.

seteuid() *seteuid* has also a clear semantics. It sets the effective uid while leaving the real uid and saved uid unchanged. However, when the current effective uid is not zero, there is a slight difference in the permission required by *seteuid* among Unix systems. While Solaris and Linux allow the parameter *neweuid* to be equal to any of the three user IDs, FreeBSD only allows *neweuid* to be equal to either the real uid or saved uid; in FreeBSD, the effective uid is not used in the decision. As a surprising result, *seteuid(geteuid())*, which a programmer might intuitively expect to be always permitted, can fail in FreeBSD, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setreuid() The semantics of *setreuid* is confusing. It modifies the real uid and effective uid, and in some cases, the saved uid. The rule by which the saved uid is modified is complicated. Furthermore, the permission required for *setreuid* differs among the three operating systems. In Solaris and Linux, a process can always swap the real uid and effective uid by calling *setreuid(geteuid(), getuid())*. In FreeBSD, however, *setreuid(geteuid(), getuid())* sometimes fails, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setuid() Although *setuid* is the only uid-setting system call standardized in POSIX 1003.1-1988, it is also the most confusing one. First, the required permission differs among Unix systems. Both Linux and Solaris require the parameter *newuid* to be equal to either the real uid or saved uid if the effective uid is not zero. As a surprising result, *setuid(geteuid())*, which a programmer might reasonably expect to be always permitted, can fail in some cases, e.g., when *ruid*=100, *euid*=200, and *suid*=100. On the other hand, *setuid(geteuid())* always succeeds in FreeBSD. Second, the action of *setuid* differs not only among different operating systems but also between privileged and unprivileged processes. In Solaris and Linux, if the effective uid is zero, a successful *setuid(newuid)* call sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD a successful *setuid(newuid)* call sets all three user IDs to *newuid* regardless of the effective uid.

setfsuid() In Linux, each process has also an *fsuid* in addition to its real uid, effective uid, and saved uid. The *fsuid* is used for access control to the filesystem. It normally follows the effective uid unless when explicitly set by the *setfsuid* system call. The Linux kernel tries to maintain the invariant that the *fsuid* is zero only if at least one of the real uid, effective uid, or saved uid is zero, as

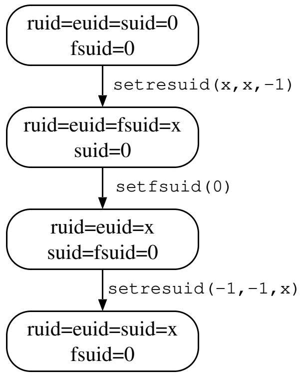


Figure 2: The call sequence shows that the invariant that *the fsuid is zero only if at least one of the ruid, euid, or suid is zero* is violated in Linux. In the figure, x represents a non-zero user ID.

manifested in the comment in a source files. The rationale is that once a process has dropped root privilege in each of its real uid, effective uid, and saved uid, the process cannot have any leftover root privilege in the *fsuid*. Since the *fsuid* is Linux specific, this invariant allows a cross-platform application that is not aware of the *fsuid* to securely drop all privileges.

Unfortunately, we discovered that this invariant may be violated due to a bug in the kernel up to the latest version of Linux (2.4.18, as of this writing). The bug is that while every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call [4]. This causes the call sequence in Figure 2 to violate the invariant. The bug has been confirmed by the Linux community. Section 6.4.3 will describe how we discovered this bug using a formal model.

setgid() and relatives There are also a set of calls for manipulating group IDs, namely, *setgid*, *setegid*, *setregid*, and *setresgid*. They behave much like their setuid counterpart, with only one minor exception (the permission check in *setregid* differs slightly from *setreuid* in Solaris). However, the *appropriate privileges* are always carried by the *euid* in both setuid-like and setgid-like calls. Thus, an effective group ID of zero does not accord any special privileges to change groups. This is a potential source of confusion: it is tempting to assume incorrectly that since *appropriate privileges* are carried by the *euid* in the setuid-like calls, they will be carried by the *egid* in the setgid-like calls, but this is not how it actually works. This misconception caused a mistake in the manual page of *setgid* in Redhat Linux 7.2 (Section 6.4.1).

In many Unix systems, a process has also a set of *supplementary group IDs* which are modified by the *setgroups* and *initgroups* calls. They are not closely related to the topic of this paper and will not be discussed.

[4] The *seteuid(euid)* call in Linux is implemented as *setreuid(-1, euid)* or *setresuid(-1, euid, -1)*, depending on the version of the C library. Hence, the *seteuid* system call might or might not set the *fsuid* reliably, depending on the C library version.

6 Formal Models

We initially began developing the summary in the previous section by manually reading operating system source code. Although reading kernel sources is a natural method to study the semantics of the uid-setting system calls, it has many serious limitations. First, it is a laborious task, especially when various Unix systems implement the system calls differently. Second, since our findings are based on current kernel sources, they may become invalid should the implementation change in the future. Third, we cannot prove that our findings are correct and that we have not misunderstood kernel sources. Finally, informal specifications are not well-suited to programmatic use, such as automated verification of properties of the operating system or use in static analysis of application programs to check proper usage of the uid-setting system calls. These problems with manual source code analysis motivate the need for more principled methods for building a formal model of the uid-setting system calls.

6.1 Building a Formal Model

Our model of the uid-setting system calls is based on finite state automata. The operating system maintains per-process state (e.g., the real, effective, and saved uids) to track privilege levels, and thus it is natural to view the operating system as implementing a finite state automaton (FSA). A state of the FSA contains all relevant information about the process, e.g., the three uids. Each uid-setting system call leads to a number of possible transitions; we label each transition with the system call that it comes from.

We construct the FSA in two steps: (1) determine its states by reading kernel sources; (2) determine its transitions by simulation. In the first step, we determine the states in the FSA by identifying kernel variables that affect the behavior of the uid-setting system calls. For example, if only the real uid, effective uid, and saved uid can affect the uid-setting system calls, then each state of the FSA is of the form (r, e, s), representing the values of the real, effective, and saved user IDs, respectively.

This is a natural approach. However, the problem one immediately faces is that the resulting FSA is much too large: in Linux, uids are 32-bit values, and so there are $(2^{32})^3 = 2^{96}$ possible states. Obviously, manipulating an FSA of such size is infeasible. Therefore, we need to somehow abstract away inessential details and reduce the size of the FSA dramatically.

Fortunately, we can note that there is a lot of symmetry present. If we have a non-root user ID, the behavior of the operating system is essentially independent of the actual value of this user ID, and depends only on the fact that it is non-zero. For example, the states $(\mathit{ruid}, \mathit{euid}, \mathit{suid}) = (100, 100, 100)$ and $(200, 200, 200)$ are isomorphic up to a substitution of the value 100 by the value 200, since the OS will behave similarly in both cases (e.g., *setuid(0)* will fail in both cases). In general, we consider two states equivalent when each can be mutated into the other by a consistent substitution on non-root user IDs. By identifying equivalent states, we can shrink the size of the FSA dramatically.

Now that we know that there must exist some reasonable FSA model, the next problem is how to compute it. Here we use *simulation*: if we simulate the presence of a pseudo-application that tries every possible system call and we observe the state transitions performed by the operating system in response to these system calls, we can infer how the operating system will behave when invoked by real applications. Once we identify equivalent states, the statespace will be small enough that we can exhaustively explore the entire statespace of the operating system. This idea is made concrete in Figure 3, where we give an algorithm to construct an FSA model using these techniques.

Note that by using simulation to create a model of the uid-setting system calls, we assume that while a process is executing such a call, the user IDs of the process cannot be modified outside the call. In other words, there is no race on the user IDs between a uid-setting system call and other parts of the kernel. This requirement might not hold in multi-threaded programs if multiple threads share the same user IDs. We leave this topic for future work.

Implementation Our implementation follows Figure 3 closely. (Note that the simulator must run as root.) In practice, we extend this basic algorithm with several optimizations and extensions.

GETSTATE():
1. Call `getresuid(&`r`, &`e`, &`s`)`.
2. Return (r, e, s).

SETSTATE(r, e, s):
1. Call `setresuid(`r, e, s`)`.
2. Check for error.

GETALLSTATES():
1. Pick n arbitrary uids $u_1, \ldots, u_n$.
2. Let $U := \{u_1, \ldots, u_n\}$.
3. Let $S := \{(r, e, s) : r, e, s \in U\}$.
4. Let $C := \{$`setuid(`x`)`, `setreuid(`x, y`)`, `setresuid(`x, y, z`)`, $\cdots$ $: x, y, z \in U \cup \{-1\}\}$.
5. Return (S, C).

BUILDMODEL():
1. Let $(S, C) :=$ GETALLSTATES().
2. Create an empty FSA with statespace S.
3. For each $s \in S$, do:
4. For each $c \in C$, do:
5. Fork a child process, and within the child, do:
6. Call SETSTATE(s), and then invoke c.
7. Finally, let $s' :=$ GETSTATE(), pass s' to the parent process, and exit.
8. Add the transition $s \xrightarrow{c} s'$ to the FSA.
9. Return the newly-constructed FSA as the model.

Figure 3: The model-extraction algorithm.

One simple optimization is to use a depth-first search to explore only the reachable states. In our case, the statespace is small enough that the improvement is probably unimportant, and we did not implement this optimization. A more dangerous optimization would be to emulate the behavior of the operating system from user-level by cutting-and-pasting the source code of the setuid system calls from the kernel into our simulation engine. This would speed up model construction, but the performance improvement comes at a severe price: it is hard to be sure that our emulation of the OS is completely faithful. In any case, our unoptimized implementation already takes only a few seconds to generate the model. For these reasons, we do *not* apply this optimization in our implementation.

To ensure maximum confidence in the correctness of our results, we check in two different ways that the call to *setresuid* in line 1 of SETSTATE() succeeds. First, we check the return value from the operating system. Second, we call *getresuid* and check that all three user IDs have been set as desired (see Section 8.1.3).

On Solaris, there are no *getresuid* and *setresuid* system calls. However, we can simulate them using the `/proc` filesystem. We read the three user IDs of a process from its `cred` file, and we modify the user IDs by writing to its `ctl` file (see *proc(4)* for details).

On Linux, we also model the *SETUID* capability bit by adding a fourth dimension to the state tuple. Thus, states are of the form (r, e, s, b) where the bit b is true whenever the *SETUID* capability is enabled. This allows us to accurately model the case where an application explicitly clears or sets its *SETUID* capability bit; though we are not aware of any real application that does this, if we ever do encounter such an application our model will still remain valid.

On all operating systems, we extend our model further to deal with system calls that fail (i.e., when invoking call c in line 6 of BUILDMODEL()). It is sometimes useful to be able to reason about whether a system call has succeeded or failed, and one way is to add a bit to the state denoting whether the previous system call returned successfully or not.

Also, on all operating systems we extend our model to include group IDs. This adds three additional dimensions to the state: real gid, effective gid, and saved gid[5]. In this way, we can model the semantics of the gid-setting system calls. On Linux, we also add a bit to indicate whether the *SETGID* capability is enabled or not.

6.2 Examples of Formal Models

In this section, we show a series of formal models of the uid-setting system calls created using the algorithm in Figure 3. These models differ in their set of user ID values. In other words, they differ in the user ID values picked in step 1 of GETALLSTATES() subroutine in Figure 3.

We start with a simple model where the set of user ID values is $\{0, x\}$ where x is a non-root user ID. Although simple, this model is accurate for many applications that manipulate at most one non-root user ID at a time. For instance, a state like $(100, 200, 100)$ will never appear in such an application. Each state in this simple FSA has three bits, each representing whether the real uid, effective uid, or saved uid is root or not. All together there are eight states in the FSA. In Figure 4 we show graphically the models one obtains in this way for the *setuid* call on Linux, Solaris, and FreeBSD. Note that the models on Solaris and Linux are equivalent, but they differ from the model on FreeBSD. Figure 5 shows the models for the *seteuid*, *setreuid*, and *setresuid* calls on Linux.

A variation of the previous models is shown in Figure 6 where the set of user ID values is $\{x, y\}$ where x and y are distinct non-root user ID values. This model is appropriate for applications that switch between two non-root user IDs (rather than between the root and a non-root user ID). This model is appropriate for analyzing BSD games [9] run under the dungeon master. Foley's work [10] offers a more serious use of this model.

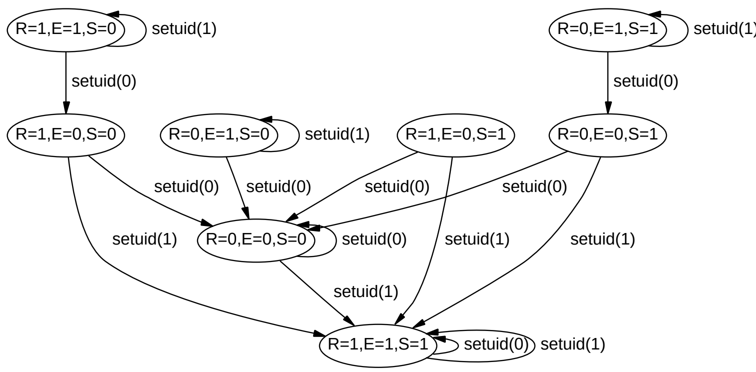
We can easily extend the simple models to include more user ID values, which are appropriate for applications that use more than two user ID values. Figure 7 shows a model where the set of user ID values is $\{0, x, y\}$ where x and y are distinct non-root user ID values. This is the fully general model of Unix user IDs.

6.3 Correctness

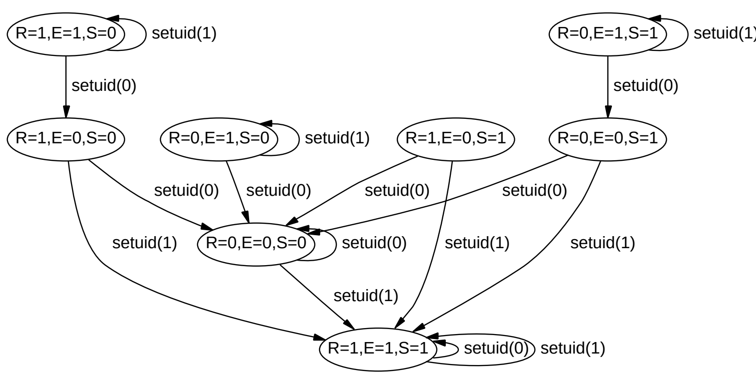
Our model-extraction algorithm (Figure 3) is an instance of a more general schema for inferring finite-state models, specialized by including application-dependent implementations of the GETSTATE(), SETSTATE(), and GETALLSTATES() subroutines. We argue that our algorithm is correct by arguing that the general version is correct. This section may be safely skipped on first reading.

We frame our theoretical discussion in terms of equivalence relations. Let $\mathcal{S}$ denote the set of concrete states (e.g., triples of 32-bit uids) and $\mathcal{C}$ the set of concrete system calls. Write $s \overset{c}{\rightsquigarrow} t$ if the operating system will always transition from state s to t upon invocation of c. We will need equivalence relations $\equiv_{\mathcal{S}}$ on $\mathcal{S}$ and $\equiv_{\mathrm{OS}}$ on $\mathcal{S} \times \mathcal{C}$ that are respected by the operating system: in other words, if $s \overset{c}{\rightsquigarrow} t$ and $s \equiv_{\mathcal{S}} s'$, then there is some state t' and some call c' so that $(s, c) \equiv_{\mathrm{OS}} (s', c')$, $t \equiv_{\mathcal{S}} t'$, and $s' \overset{c'}{\rightsquigarrow} t'$. The intuition is that calling c from s is somehow isomorphic to calling c' from s'. Also, we require that whenever $(s, c) \equiv_{\mathrm{OS}} (s', c')$ holds, then $s \equiv_{\mathcal{S}} s'$ does, too.

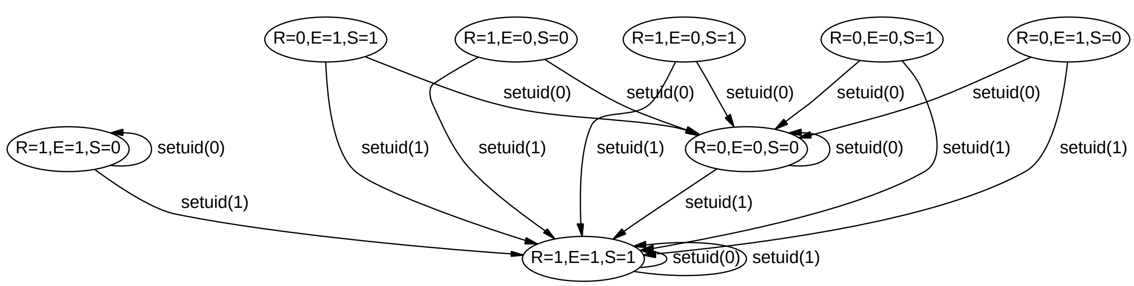
[5] We don't currently model supplemental groups, though this would be straightforward to correct. Note that this omission does not affect the correctness of our model, as supplemental groups are only used in access control checks and never affect the behavior of the *setgid*-like calls.



(a) An FSA describing *setuid* in Linux 2.4.18

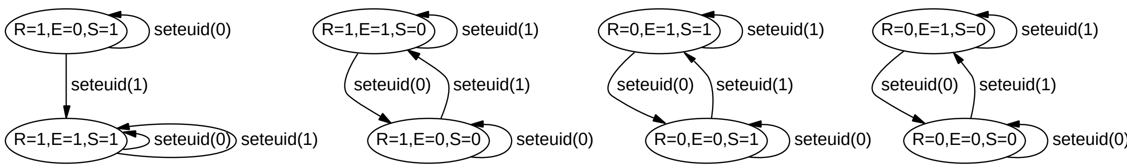


(b) An FSA describing *setuid* in Solaris 8

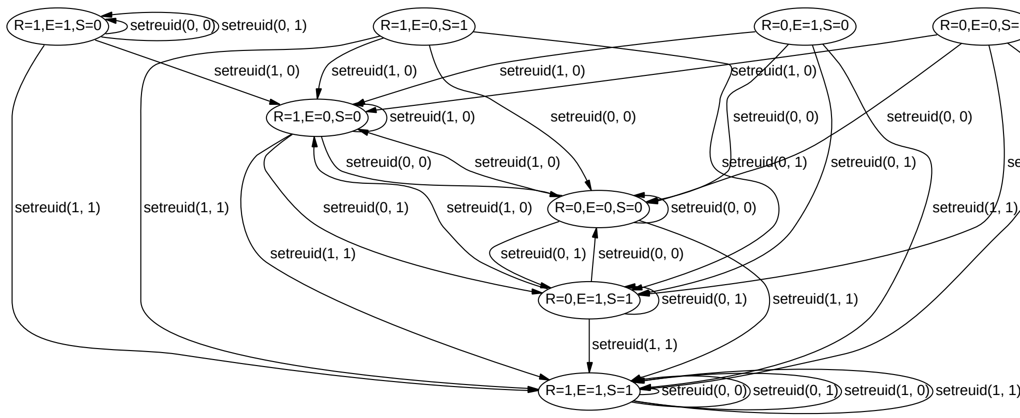


(c) An FSA describing *setuid* in FreeBSD 4.4

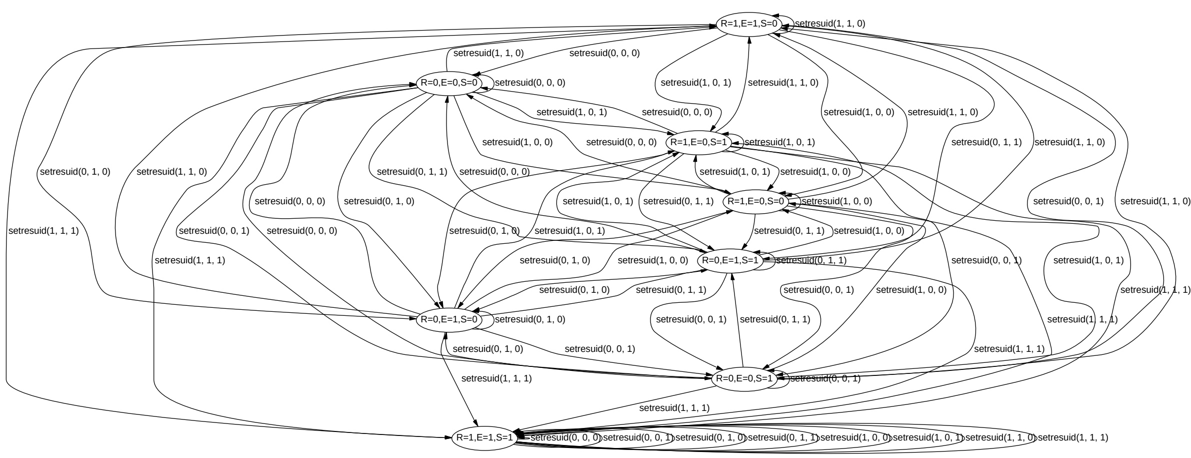
Figure 4: Three finite state automata describing the *setuid* system call in Linux, Solaris, and FreeBSD, respectively. Ellipses represent states of the FSA, where a notation like "R=1,E=0,S=1" indicates that $euid = 0$ and $ruid = suid \neq 0$. Each transition is labelled with the system call it corresponds to. To avoid cluttering the diagram, we omit the error states and (in Linux) the capability bits that otherwise would appear in our deduced model.



(a) An FSA describing *seteuid* in Linux



(b) An FSA describing *setreuid* in Linux



(c) An FSA describing *setresuid* in Linux

Figure 5: Three finite state automata describing the *seteuid*, *setreuid*, *setresuid* system calls in Linux respectively. Ellipses represent states of the FSA, where a notation like "R=1,E=0,S=1" indicates that $euid = 0$ and $ruid = suid \neq 0$. Each transition is labelled with the system call it corresponds to.

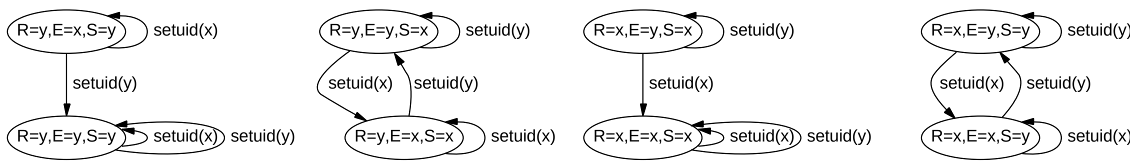


Figure 6: A finite state automaton describing the *setuid* system call in Linux. This FSA considers only two distinct non-root user ID values x and y. Ellipses represent states of the FSA, where a notation like "R=x,E=y,S=x" indicates that *euid* $= y$ and *ruid* $=$ *suid* $= x$. Each transition is labelled with the system call it corresponds to.

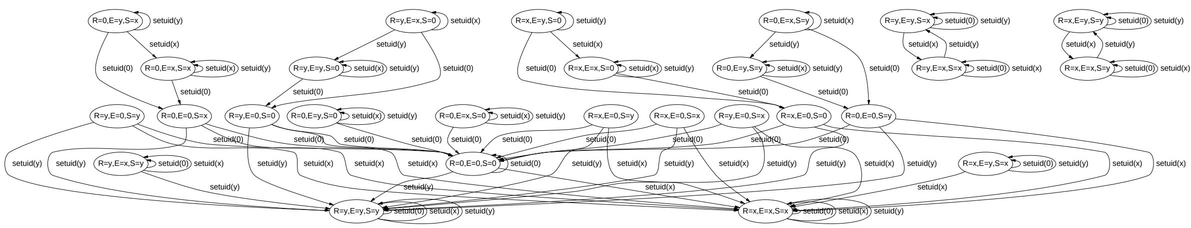


Figure 7: A finite state automaton describing the *setuid* system call in Linux. This FSA considers three user ID values: the root user ID and two distinct non-root user ID values x and y. Ellipses represent states of the FSA, where a notation like "R=0,E=x,S=y" indicates that *ruid* $= 0$, *euid* $= x$ and *suid* $= y$. Each transition is labelled with the system call it corresponds to.

A critical requirement is that the operating system must behave *deterministically* given the equivalence class of the current state. More precisely, if $s \stackrel{c}{\leadsto} t$ and $s' \stackrel{c'}{\leadsto} u$ where $(s, c) \equiv_{\text{OS}} (s', c')$, then we require $t \equiv_{\mathcal{S}} u$. The intuition is that the behavior of the operating system will depend only on which equivalence class we are in, and not on any other information about the state. For instance, the behavior of the operating system cannot depend on any global variables that don't appear in the state s; if it does, these global variables must be included into the statespace $\mathcal{S}$. As another example, a system call implementation that attempts to allocate memory and returned an error code if this allocation fails will violate our requirement, because the success or failure of the memory allocation introduces non-determinism, which is prohibited. We can see that this requirement is non-trivial, and it must be verified by manual inspection of the source code before our algorithm in Figure 3 can be safely applied; we will return to this issue later.

Next, there are three requirements on the instantiation of the GETSTATE(), SETSTATE(), and GETALLSTATES() subroutines. First, the GETSTATE() routine must return (a representative for) the equivalence class of the current state of the operating system. Note that it is natural to represent equivalence classes internally by singling out a unique representative for each equivalence class and using this value. Second, the SETSTATE() procedure with parameter s must somehow cause the operating system to enter a state s' in the same equivalence class as s (the implementation may freely choose one). Finally, the GETALLSTATES() function must return a pair (S, C) so that S contains at least one representative from each equivalence class of $\equiv_{\mathcal{S}}$ and so that every equivalence class of $\equiv_{\text{OS}}$ contains some element (s, c) with $c \in C$.

When these general requirements are satisfied, the BUILDMODEL() algorithm from Figure 3 will correctly infer a valid finite-state model for the underlying operating system. The proof is easy. We will write $[x]$ for the equivalence class containing x, e.g., $[s] = \{t \in \mathcal{S} : s \equiv_{\mathcal{S}} t\}$. If $s \xrightarrow{c} t$ appears in the final FSA output by BUILDMODEL(), then there must have been a step at which, for some $s' \in [s]$, $t' \in [t]$, and c' with $(s, c) \equiv_{\text{OS}} (s', c')$, we executed c' in state s' at line 6 and transitioned to state t'. (This follows from the correctness of SETSTATE() and GETSTATE().) The latter means that $s' \stackrel{c'}{\leadsto} t'$, from which it follows that $s \stackrel{c}{\leadsto} t''$ for some $t'' \in [t]$, since the OS respects $\equiv_{\text{OS}}$. Conversely, if $s' \stackrel{c'}{\leadsto} t'$ for some s', c', t', then by the correctness of GETALLSTATES(), there will be some s and c satisfying $(s, c) \equiv_{\text{OS}} (s', c')$ so that we enter line 6 with s, c, and thanks to the deterministic nature of the operating system we will discover the transition $s \xrightarrow{c} t$ for some $t \equiv_{\mathcal{S}} t'$. Thus, the FSA output by BUILDMODEL() is exactly what it should be. Consequently, all that remains is to check that these requirements are satisfied by our instantiation of the schema.

We argue this next for the implementation shown in

Figure 3. Let $\mathcal{U}$ denote the set of concrete uids (e.g., all 32-bit values), so that $\mathcal{S} = \mathcal{U} \times \mathcal{U} \times \mathcal{U}$. Say that a map $\sigma : \mathcal{U} \to \mathcal{U}$ is a *valid substitution* if it is bijective and fixes 0, i.e., $\sigma(0) = 0$. Each such substitution can be extended to one on $\mathcal{S}$ by working component-wise, i.e., $\sigma(r, e, s) = (\sigma(r), \sigma(e), \sigma(s))$, and we can extend it to work on system calls by applying the substitution to the arguments of the system call, e.g., $\sigma(\texttt{setreuid}(r, e)) = \texttt{setreuid}(\sigma(r), \sigma(e))$. We define our equivalence relation $\equiv_{\mathcal{S}}$ on $\mathcal{S}$ as follows: two states $s, s' \in \mathcal{S}$ are equivalent if there is a valid substitution σ such that $\sigma(s) = s'$. Similarly, $(s, c) \equiv_{\mathrm{OS}} (s', c')$ holds if there is some valid substitution σ so that $\sigma(s) = s'$ and $\sigma(c) = c'$.

The correctness of GETSTATE() and SETSTATE() is immediate. Also, so long as $n \geq 6$, GETALLSTATES() is correct since the choice of uids $u_1, \ldots, u_n$ is immaterial: every pair $(s, c) \in \mathcal{S} \times \mathcal{C}$ is equivalent to some pair $(s', c') \in S \times C$, since we can simply map the first six non-zero uids in (s, c) to $u_1, \ldots, u_6$ respectively, and there can be at most six non-zero uids in (s, c). Actually, we can see that the algorithm in Figure 3 comes from a finer partition than that given by $\equiv_{\mathrm{OS}}$: for example, (u_1, u_1, u_1) and (u_2, u_2, u_2) are unnecessarily distinguished. This causes no harm to the correctness of the result, and only unnecessarily increases the size of the resulting FSA. We gave the variant shown in Figure 3 because it is simpler to present, but in practice our implementation does use the coarser relation $\equiv_{\mathcal{S}}$.

All that remains to check is that the operating system respects and behaves deterministically with respect to this equivalence class. We verify this by manual inspection of the kernel sources, which shows that in Linux, FreeBSD, and Solaris the only operations that the uid-setting system calls perform on user IDs are equality testing of two user IDs, comparison to zero, copying one user ID to another, and setting a user ID to zero. Moreover, the operating system behavior does not depend on anything else, with one exception: Linux depends on whether the *SETUID* capability is enabled for the process, so on Linux we add an extra bit to each state indicating whether this capability is enabled. Thus, our verification task amounts to checking that user IDs are treated as an abstract data type with only four operations (equality testing, comparison to zero, and so on) and that the side effects and results of the system call do not depend on anything outside the state S. In our experience, verifying that the operating system satisfies these conditions is much easier than fully understanding its behavior, as the former is an almost purely mechanical process.

This completes our justification for the correctness of our method for extracting a formal model to capture the behavior of the operating system.

6.4 Applications

The resulting formal model has many applications. We have already discussed in Section 5 the semantics of the setuid system calls and pointed out pitfalls; this relied heavily on the FSA formal model. Next, we will discuss several additional applications: verifying documentation and checking conformance with informal specifications; identifying cross-platform semantic differences that might indicate potential portability issues; detecting inconsistency in the handling of user IDs within an OS kernel; and checking the proper usage of the uid-setting system calls in programs automatically.

6.4.1 Verifying Accuracy of Manual Pages

Manual pages are the primary source of information for Unix programmers, but unfortunately they are often incomplete or wrong. FSAs are useful in verifying the accuracy of manual pages of uid-setting system calls. For each call, if its FSA is small and its description in manual pages is simple, we check if each transition in the FSA agrees with the description by hand. Otherwise, we build another FSA based on the description and compare this FSA to the original FSA built by simulation. Differences between the two FSAs indicate discrepancies between the behavior of the system call and its description in manual pages.

The following are a few examples of problematic documentation that we have found using our formal model:

- The man page of *setuid* in Redhat Linux 7.2 fails to mention the *SETUID* capability, which affects the behavior of *setuid*.
- The man page of *setreuid* in FreeBSD 4.4 says:

  > Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

  However, this is incorrect. Swapping the real uid and effective uid does not always succeed, such as when ruid=100, euid=200, suid=100, contrary to what the man page suggests. The correct description is “Unprivileged users may change the real user

ID to the real uid or saved uid, and change the effective uid to the real uid, effective uid, or saved uid."

- The man page of *setgid* in Redhat Linux 7.2 says

  > The setgid function checks the effective gid of the caller and if it is the superuser, all process related group ID's are set to gid.

  In reality, the effective **uid** is checked instead of the effective **gid**.

6.4.2 Identifying Implementation Differences

Since various Unix systems implement the uid-setting system calls differently, it is difficult to identify their semantic differences via reading kernel sources. We can solve this problem by creating an FSA of the user ID model in each Unix system and contrasting the FSAs. For example, Figure 4 shows clearly that the semantics of *setuid* in Solaris is different from that in FreeBSD and Linux.

The approach can be further formalized by taking the symmetric difference of FSAs. In particular, if M, M' are two FSAs for two Unix platforms with the same state-space, we can find portability issues as follows. Compute the parallel composition $M \times M'$, whose states are pairs (s, s') with s a state from M and s' a state from M'. Then, mark as an accepting state of $M \times M'$ any pair (s, s') where $s \neq s'$. Now any execution trace that starts at a non-accepting state and eventually reaches an accepting state indicates a sequence of system calls whose semantics is not the same on both operating systems. This indicates a potential portability issue, and all such differences can be computed via a simple reachability computation (e.g., depth-first search).

6.4.3 Detecting Inconsistency within an OS Kernel

An OS kernel maintains many invariants which both the kernel itself and many application programs depend on. Violation of the invariants may cause vulnerabilities in both the OS and applications. Therefore, it is important to detect any violation of the invariants.

The Linux kernel tries to maintain the invariant that the *fsuid* is zero only if at least one of the real uid, effective uid, or saved uid is zero. To verify this invariant, we extend the formal model of user IDs with the *fsuid* and automatically create an FSA of the model on Linux. From the FSA, we discovered that the invariant does not always hold, because the state where $fsuid = 0$ and $ruid \neq 0$, $euid \neq 0$, $suid \neq 0$ is reachable. For example, the call sequence in Figure 2 will violate the invariant. The problem results from an inconsistency in the handling of the *fsuid* in the uid-setting system calls. While every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call. The problem has been confirmed by the Linux community.

6.4.4 Checking Proper Usage of Uid-setting System Calls

The formal model is also useful in checking proper usage of uid-setting system calls in programs. We model a program as an FSA, called the *program FSA*, which represents each program point as a state and each statement as a transition. We call the FSA describing the user ID model a *model FSA*. By composing the program FSA with the model FSA, we get a *composite FSA*. Each state in the composite FSA is a pair (s, s') of one state s from the model FSA (representing a unique combination of the values in the real uid, effective uid, and saved uid) and one state s' from the program FSA (representing a program point). Thus, a reachable state (s, s') in the composite FSA indicates that the state s in the model FSA is reachable at the program point s'. Figure 8(b) shows the program FSA of the program in Figure 8(a). Figure 8(c) shows the composite FSA obtained by composing the model FSA in 4(a) with the program FSA in Figure 8(b).

This method is useful for checking proper usage of uid-setting system calls in programs, such as:

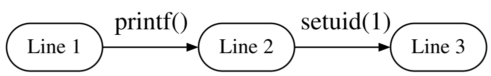
- Can a uid-setting system call fail? If any error state in the model FSA is reachable at some program point, it shows that a uid-setting system call may fail there.

- Can a program fail to drop privilege? If any state that contains a privileged user ID in the model FSA is reachable at a program point where the program should be unprivileged, it shows that the program may have failed to drop privilege at an earlier program point.

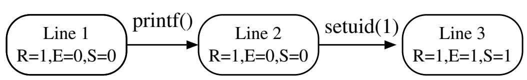
- Which part of the program may run with privilege? To answer this question, we first identify all states that contain a privileged user ID in the model FSA.

```
   // ruid=1, euid=0, suid=0
1: printf("drop priv");
2: setuid(1);
3: execl("/bin/sh", "sh",NULL);
```

(a) A program segment



(b) Program FSA of the program in Figure 8(a)



(c) Composite FSA of the model FSA in Figure 4(a) and the program FSA in Figure 8(a)

Figure 8: Composing a model FSA with a program FSA

Then, we identify all program points where any of those states are reachable. The program may run with privilege at these program points.

A full discussion is out of the scope of this paper, and we refer the interested reader to a companion paper for details [11].

6.5 Advantages

The formal model holds several advantages over trying to understand the behavior of the kernel through manual code inspection. First, our formal model makes it easier to describe the properties of the uid-setting system calls. While we still need to read kernel code to determine the kernel variables that affect the uid-setting system calls, the majority of the workload, determining their actions, is done automatically by simulation. Second, the formal model is reliable because it is created from the same environment where application programs run. The formal model has corrected several mistakes in the user ID model that we created manually. Third, the formal model is useful in identifying semantic differences of uid-setting system calls among Unix systems. Fourth, the formal model is useful in detecting inconsistency in an OS kernel. Finally, the formal model is useful in checking proper usage of uid-setting system calls in programs automatically.

7 Case Studies of Security Vulnerability

Misuses of uid-setting system calls have caused many security vulnerabilities, which are good lessons in learning the proper usage of the system calls. We will analyze two such incidents in older versions of sendmail.

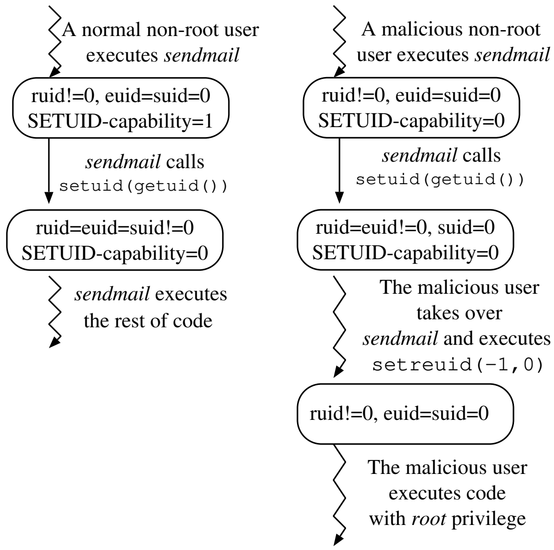
Sendmail [12] is a commonly used Mail Transmission Agent(MTA). It runs in two modes: (1) as a daemon that listens on port 25 (SMTP), and (2) via a Mail User Agent to submit mail to the mail queue. In the first case, all three user IDs of the sendmail process are typically zero, as it is run by the superuser *root* in the boot process. In the second case, however, sendmail is run by an ordinary user. As the mail queue is not world writable, sendmail requires privilege to access the mail queue.

7.1 Misuse of Setuid

Next we describe a vulnerability that was caused by a misuse of setuid [13]. Sendmail 8.10.1 installed the *sendmail* binary as a setuid-root executable. When it was executed by a non-root user, the real uid of the process was the non-root user while both the effective uid and saved uid were zero. This gave *sendmail* permission to write to the mail queue since its effective uid was zero. To minimize risks in the event that an attacker takes over *sendmail* and executes malicious code with root privilege, *sendmail* permanently dropped root privilege before doing potentially dangerous operations requested by an user. This was done by calling *setuid(getuid())*, which sets all three user IDs to the non-root user.

POSIX specifies that if a process has *appropriate privileges*, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, *setuid(newuid)* only sets the effective uid to *newuid* (if *newuid* is equal to the real uid or saved uid). In Linux, *appropriate privileges* are carried by the *SETUID* capability. Furthermore, after any uid-setting system call, the Linux kernel sets or clears the *SETUID* capability bit, if necessary, to establish a simple post-condition: the *SETUID* capability should be set if and only if the effective uid is zero.

However, prior to version 2.2.16 of Linux, there was a bug in the kernel that made it possible for a process to



(a) A normal execution of sendmail by a non-root user

(b) An execution of sendmail by an attacker

Figure 9: A vulnerability in sendmail due to a misuse of *setuid*. Note the failure: the programmer assumed that *setuid(getuid())* would always succeed in dropping all privilege, but by disabling the *SETUID* capability, the attacker is able to violate that expectation.

clear its *SETUID* capability bit even when its effective uid was zero. In this case, calling *setuid(getuid())* only modified the effective uid, and under these conditions, *sendmail* would only drop root privilege from its effective uid but not its saved uid. Consequently, any malicious local user who could take over *sendmail* (e.g., with a buffer overrun attack) could restore root privilege in the effective uid by calling *setreuid(-1, 0)*. In other words, an attacker could ensure *sendmail*'s attempt to drop all privileges would fail, thereby raising the risk of a root attack on sendmail. Figure 9 illustrates the vulnerability.

The vulnerability was caused by the overloaded semantics of *setuid*. Depending on whether a process has the *SETUID* capability, setuid sets one user ID or all three user IDs, but it returns a success code in both cases. The vulnerability can be avoided by replacing *setuid(newuid)* with *setresuid(newuid, newuid, newuid)* if available, or with *setreuid(newuid, newuid)* otherwise.

7.2 Interaction between User IDs and Group IDs

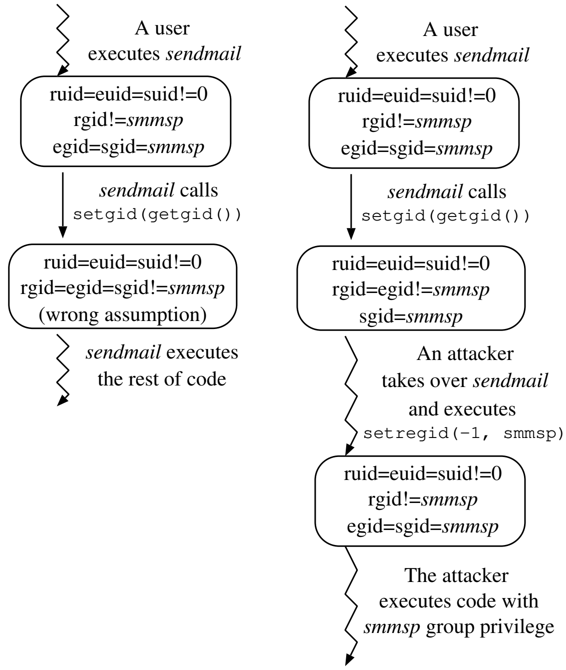
Another vulnerability in Sendmail was caused by an interaction between the user IDs and the group IDs [14]. To further reduce the risk from a malicious user taking over *sendmail*, as of version 8.12.0 Sendmail no longer installed *sendmail* as a setuid-root program. To give *sendmail* permission to write to the mail queue, the mail queue was configured to be writable by group *smmsp*, and *sendmail* was installed as setgid-*smmsp*. Therefore, when *sendmail* was executed by a non-root user, the real gid of the process was the primary group of the user, but the effective gid and saved gid were *smmsp*.

For the same reason that it permanently dropped root privilege in previous versions, now *sendmail* permanently dropped *smmsp* group privilege before executing potentially malicious directives from a user. Similar to the use of *setuid(getuid())* to permanently drop root privilege, *sendmail* called *setgid(getgid())* to permanently drop *smmsp* group privilege. However, since *sendmail* no longer had *appropriate privileges* because its effective uid was not zero anymore, *setgid(getgid())* only dropped the privileged group ID *smmsp* from the effective gid but left it in the saved gid. Consequently, any malicious user who found some way to take over sendmail (e.g., by a buffer overrun) could restore the *smmsp* group privilege in the effective gid by calling *setgid(-1, smmsp)*. This is illustrated in Figure 10.

The vulnerability was caused by an interaction between the user IDs and group IDs since changing user IDs may affect the property of *setgid*. To avoid the vulnerability, we can replace *setgid(newgid)* with *setresgid(newgid, newgid, newgid)* if available, or *setregid(newgid, newgid)* otherwise. The vulnerability also shows that if both user IDs and group IDs are to be modified, the modification should follow a specific order (Section 8.1.2).

8 Guidelines

We provide guidelines on the proper usage of the uid-setting system calls. First, we discuss general guidelines that apply to all setuid programs. Then, we focus on applications that use the uid-setting system calls in a specific way. We propose a high-level API for these applications to manage their privileges. The API is easier to understand and to use than the Unix API.



(a) The programmer's mental model of an expected execution trace

(b) Real execution of sendmail by a malicious user

Figure 10: A vulnerability in sendmail due to interaction between user IDs and group IDs. The failure occurs because the programmer has overlooked that she has already dropped root privilege and hence no longer has the *appropriate privileges* to drop all group privileges in the *setgid* call.

8.1 General Guidelines

8.1.1 Selecting an Appropriate System Call

Since *setresuid* has a clear semantics and is able to set each user ID individually, it should always be used if available. Otherwise, to set only the effective uid, *seteuid(new_euid)* should be used; to set all three user IDs, *setreuid(new_uid, new_uid)* should be used.

setuid should be avoided because its overloaded semantics and inconsistent implementation in different Unix systems may cause confusion and security vulnerabilities for the unwary programmer. As described in Section 5.2, in Linux or Solaris, if the effective user ID is zero, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD *setuid(newuid)* sets all three user IDs to *newuid* regardless of the effective user ID. We envision the following scenarios where *setuid* may be misused:

- If a setuid-root program temporarily drops root privilege with *seteuid(getuid())* and later calls *setuid(getuid())* with the intention of permanently dropping all root privileges, the program does not get the intended behavior on Linux or Solaris, because the saved user ID remains root. (However, the program does receive the intended behavior on FreeBSD.)
- Also on Linux or Solaris, in a setuid-root program, calling *setuid(getuid())* permanently drops root privileges; however, in a setuid-non-root program (e.g., a program that is setuid-Alice where Alice is a non-root user), calling *setuid(getuid())* will not permanently drop Alice's privileges, because the saved user ID remains Alice. This is particularly confusing, because the way setuid-root programs permanently drop privileges does not work in setuid-non-root programs on Linux or Solaris.

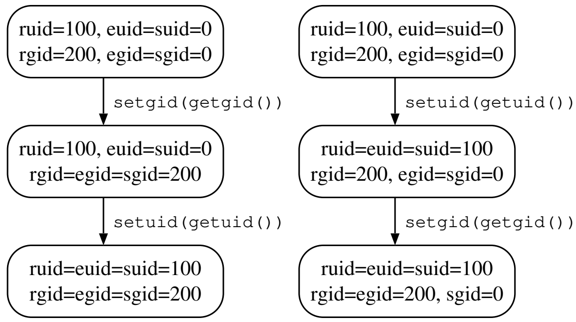
8.1.2 Obeying the Proper Order of System Calls

The POSIX-defined *appropriate privileges* affect the actions of both system calls that set user IDs and that set group IDs. Since often *appropriate privileges* are carried by the effective uid, a program should drop group privileges before dropping user privileges permanently. Otherwise, after permanently dropping user privileges, the program may be unable to permanently drop group privileges. For example, the program in Figure 11(a) is able to permanently drop both user and group privileges because it calls *setgid* before *setuid*. In contrast, since the program in Figure 11(b) calls *setuid* before *setgid*, it fails to drop group privileges permanently.

8.1.3 Verifying Proper Execution of System Calls

Since the semantics of the uid-setting system calls may change, e.g., when the kernel changes or when an application is ported to a different Unix system, it is imperative to verify successful execution of these system calls.

Checking Return Codes The uid-setting system calls return zero on success and non-zero on failure. A process should check the return codes to verify the successful execution of these calls. This is especially important when



(a) A program correctly drops both user and group privileges permanently by calling *setgid(getgid())* before *setuid(getuid)*

(b) A program fails to drop group privileges permanently because it calls *setuid(getuid())* before *setgid(getgid())*

Figure 11: Proper order of dropping user and group privileges. Figure (a), on the left, shows proper usage; figure (b) shows what can go wrong if one gets the order backwards.

a process permanently drops privilege, since such an action usually precedes operations that, if executed with privilege, may compromise the system.

Be aware that the Linux-specific *setfsuid* system call returns the previous *fsuid* from before the call and does not return any error message to the caller on failure. This is one motivation for our next guideline.

Verifying User IDs However, checking return codes may be insufficient for uid-setting system calls. For example, in Linux and Solaris, depending on the effective uid, *setuid(newuid)* may either (1) set all three user IDs (if the effective uid is zero), or (2) set only the effective uid (if it is non-zero), but the system call returns the same success code in both cases. The return code does not indicate to the process which case has happened, and thus checking return codes is not enough to guarantee successful completion of the uid operation in some cases. Moreover, checking the return code is infeasible for the *setfsuid* call since it does not return any error message on failure.

Therefore, after each uid-setting system call, a program should verify that each of its user IDs are as expected. A process may call *getresuid* to check all three user IDs if it is available, as in Linux and FreeBSD, or use the `/proc` filesystem on Solaris. Otherwise, the process may call *getuid* and *geteuid* to check the real uid and effective uid, if none of these are available. In Linux, a process must examine its *fsuid* via the `/proc` filesystem since Linux does not offer a *getfsuid* call.

```
// drop privilege
setuid(getuid());

// verify the process cannot restore privilege
if (setreuid(-1, 0) == 0)
  return ERROR;
```

Figure 12: An example of a program that verifies that it has properly dropped root privileges. The verification is achieved by checking that unpermitted uid-setting system calls will fail. Note that a full implementation should also check the return code from *setuid* and verify that all three user IDs are as expected after the call to *setuid*.

Verifying Failures Once an attacker takes control of a process, the attacker may insert arbitrary code into the process. Therefore, for further assurance on security, the process should ensure that all unpermitted uid-setting system calls will fail. For example, after dropping privilege permanently, the process should verify that attempts to restore privilege will fail. This is shown in Figure 12.

8.2 An Improved API for Privilege Management

Although the general guidelines in Section 8.1 can help programmers to use the uid-setting system calls more securely, programmers still have to grapple with the complex semantics of the uid-setting system calls and their differences among Unix systems. The complexity is partly due to a mismatch between the low-level semantics of the system calls, which describes how to modify the user IDs, and the high-level goals of the programmer, which represent a policy for when the application should run with privilege. We propose to resolve this tension by introducing an API that is better matched to the needs of application programmers.

8.2.1 Proposed API

In many applications, privilege management can typically be broken down into the following tasks:

- Drop privilege temporarily, in a way that allows the privilege to be restored later.

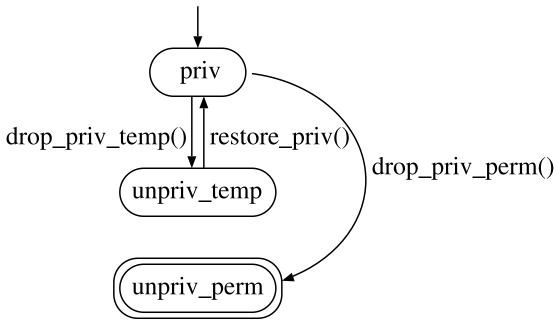


Figure 13: An FSA showing the behavior of a process when calling the functions of the new API.

- Drop privilege permanently, so that it can never be restored.
- Restore privilege.

We propose a new API that offers the ability to perform each of these tasks directly and easily. The API contains three functions:

- *drop_priv_temp(new_uid)*: Drop privilege temporarily. Move the privileged user ID from the effective uid to the saved uid. Assign *new_uid* to the effective uid.
- *drop_priv_perm(new_uid)*: Drop privilege permanently. Assign *new_uid* to all the real uid, effective uid, and saved uid.
- *restore_priv*: Restore privilege. Copy the privileged user ID from the saved uid to the effective uid.

By raising the level of abstraction, we free programmers to think more about their desired security policy and less about the mechanism of implementing this policy. Figure 13 illustrates the action of these functions pictorially with a simple state diagram.

8.2.2 Implementation

We implement the new API as wrapper functions to the uid-setting system calls. The implementation uses *setresuid* if available since it has the clearest semantics and it is able to set each of the user IDs independently, as shown in Figure 14. If *setresuid* or its equivalent is not available, the implementation uses *seteuid* and *setreuid*, as shown in Figure 15.

```
int drop_priv_temp(uid_t new_uid)
{
  if (setresuid(-1, new_uid, geteuid()) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int drop_priv_perm(uid_t new_uid)
{
  uid_t ruid, euid, suid;
  if (setresuid(new_uid, new_uid, new_uid) < 0)
    return ERROR_SYSCALL;
  if (getresuid(&ruid, &euid, &suid) < 0)
    return ERROR_SYSCALL;
  if (ruid != new_uid || euid != new_uid ||
      suid != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int restore_priv()
{
  int ruid, euid, suid;
  if (getresuid(&ruid, &euid, &suid) < 0)
    return ERROR_SYSCALL;
  if (setresuid(-1, suid, -1) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != suid)
    return ERROR_SYSCALL;
  return 0;
}
```

Figure 14: A possible implementation of the high-level API for systems with *setresuid*.

To use this implementation, an application must meet the following requirements:

- When the process starts, its effective uid contains the privileged user ID. This is true in most circumstances. When a process is run by a privileged user, all three user IDs contain the privileged user ID. If the process is run as a privileged user, i.e., its executable is setuid'ed to the privileged user and is run by an unprivileged user, both the effective uid and saved uid of the process contain the privilege user ID.
- If the privileged user ID is not zero, then the unprivileged user ID must be stored in the real uid when the process starts. This requirement enables the process to replace the privileged user ID in the effective uid with the unprivileged user ID in *drop_priv_temp* and *drop_priv_perm*. This is the case when a non-root user executes an executable that is setuid'ed to another non-root user. On the other hand, if the privileged user ID is zero, then there is no such requirement, since the process can set its user IDs to

```
uid_t priv_uid;

int drop_priv_temp(uid_t new_uid)
{
  int old_euid = geteuid();

  // copy euid to suid
  if (setreuid(getuid(), old_euid) < 0)
    return ERROR_SYSCALL;
  // set euid as new_uid
  if (seteuid(new_uid) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != new_uid)
    return ERROR_SYSCALL;
  priv_uid = old_euid;
  return 0;
}

int drop_priv_perm(uid_t new_uid)
{
  uid_t suid;
  if (setreuid(new_uid, new_uid) < 0)
    return ERROR_SYSCALL;
  // OS specific way of reading suid
  suid = read_suid_from_proc_filesystem();
  if (getuid() != new_uid ||
      geteuid() != new_uid ||
      suid != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int restore_priv()
{
  if (seteuid(priv_uid) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != priv_uid)
    return ERROR_SYSCALL;
  return 0;
}
```

Figure 15: A possible implementation of the high-level API for systems without *setresuid*.

arbitrary values.

- The process does not make any uid-setting system calls that change any of the three user IDs. Such a call may cause the process to enter a state not covered by the FSA in Figure 13, on which the high-level API and the implementation are based.

The implementation has the following beneficial properties:

- It does not affect the real uid.
- It guarantees that all transitions in Figure 13 succeed.
- It verifies that the user IDs are as expected after each uid-setting system call.
- It does the right thing even in cases where root is not involved, i.e., where the privileged user ID is not the superuser.

We can extend this basic implementation to include stronger safeguards against programming errors or OS inconsistency. To prevent a program from restoring a wrong privilege, we can let the function *restore_priv* take a parameter and check that the parameter matches the privilege stored in the saved user ID (Figure 14) or in the variable *priv_uid* (Figure 15). Another improvement is to let the function *drop_priv_perm* verify that an attempt to regain privilege will fail, as described in Section 8.1.3.

8.2.3 Evaluation

To evaluate the high-level API, we replaced every uid-setting system call in OpenSSH 2.5.2 with functions from the new API. OpenSSH contains fifteen uid-setting system calls in eight tasks. Of the eight tasks, four are to drop privilege permanently, two are to drop privilege temporarily, and two are to restore privilege. We are able to implement all these tasks with the new API.

One known limitation of our API is that it does not address group privileges. We leave this for future work.

9 Future Work

We plan to study how the uid-setting system calls affect other properties of a process, such as the ability to receive signals and to dump cores. We may also study how to extend the formal models for multi-threaded programs. Topics to investigate include in-kernel races and how the user IDs are inherited during the creation of new threads in different Unix systems.

10 Conclusion

We have studied the proper usage of the uid-setting system calls by two approaches. First, we documented the semantics of the uid-setting system calls in three major Unix systems (Linux, Solaris, and FreeBSD) and identified their differences. We then showed how to formalize this problem using formal methods, and we proposed a new algorithm for constructing a formal model of the semantics of the uid-setting system calls. Using the resulting formal model, we identified semantic differences of the uid-setting system calls among Unix systems and discovered inconsistency within an OS kernel. Finally, we provided guidelines for proper usage of the uid-setting system calls and proposed a high-level API for managing user IDs that is more comprehensible, usable, and portable than the usual Unix API.

Acknowledgment

We thank Monica Chew, Solar Designer, Peter Gutmann, Robert Johnson, Ben Liblit, Zhendong Su, Theodore Ts'o, Wietse Venema, Michal Zalewski, and the anonymous reviewers for their valuable comments.

References

[1] Chris Torek and Casper H.S. Dik. Setuid mess. `http://yarchive.net/comp/setuid_mess.html`.

[2] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, 1992.

[3] Matt Bishop. How to write a setuid program. *;login:*, 12(1):5–11, 1987.

[4] Dennis M. Ritchie. Protection of data file contents. United States Patent #4,135,240. Available from `http://www.uspto.gov`.

[5] *IEEE Standard 1003.1-1998: IEEE standard portable operating system interface for computer environments*. Institute of Electrical and Electronics Engineers, 1988.

[6] `http://www.kernel.org`.

[7] `http://www.sun.com/software/solaris/`.

[8] `http://www.freebsd.org`.

[9] dm(8). 4.4 BSD System Manager's Manual.

[10] Simon N. Foley. Implementing Chinese walls in Unix. *Computers and Security Journal*, 16(6):551–563, December 1997.

[11] Hao Chen, David Wagner, and Drew Dean. An infrastructure for examining security properties of software. manuscript in preparation.

[12] `http://www.sendmail.org/`.

[13] Sendmail Inc. Sendmail workaround for linux capabilities bug. `http://www.sendmail.org/sendmail.8.10.1.LINUX-SECURITY.txt`.

[14] Michal Zalewski. Multiple local sendmail vulnerabilities. `http://razor.bindview.com/publish/advisories/adv_sm812.html`.