The UNIX Process Identity Crisis: A Standards-Driven Approach to Setuid

by

Mark S. Dittmer

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This work revisits the setuid family of calls for privilege management that is implemented in several widely-used operating systems. Three of the four commonly used calls in the family are standardized by POSIX.

The work investigates the current status of setuid and, in the process, challenges some assertions in prior work. It addresses three sets of questions with regards to the setuid family: (1) Is the POSIX standard indeed broken as prior work suggests? (2) Are implementations POSIX-compliant as claimed? (3) Are the wrapper functions that prior work proposes to circumvent issues with setuid calls correct and usable?

Towards (1), the standards are expressed in a precise syntax that lends itself to a rigorous assessment of whether the standards are unambiguous and logically consistent descriptions of well-formed functions. Under some reasonable assumptions, two of the three functions that are standardized fit these criteria, which challenges assertions in prior work regarding the quality of the standard. In cases wherein the standard is broken, the problem is clearly characterized, and suggestions are given for fixing standard, but at the cost of backwards-compatibility.

Towards (2), a state-space enumeration is performed as in prior work, and a discussion of the implications of non-conformance and differences in implementation is presented.

Towards (3), some issues with prior wrappers are identified. The work proposes a new suite of wrapper functions which are designed with a different mindset from prior work, and provides both stronger guarantees with respect to atomicity and a clearer semantics for permanent and temporary changes in process identity.

With a fresh approach, this work is a contribution to a well-established mechanism for privilege management.

Acknowledgements

> All blessings are divine in origin but none can be compared with this power of intellectual investigation and research which is an eternal gift producing fruits of unending delight. [...] Briefly; it is an eternal blessing and divine bestowal, the supreme gift of God to man.
>
> ∼ 'Abdu'l-Bahá

For all my capacities and achievements I give thanks to God.

I would like to thank my wife, Livia, for her loving support in every aspect of my education, this work included. I would also like to thank my supervisor, Dr. Mahesh V. Tripunitara, whose unfailing support and generosity continue to ensure a high standard of quality for my work, and Dr. Vijay Ganesh whose compelling offering of a course on formal logic applications inspired core aspects to my approach of the subject matter. Finally, I would like my league of indefatigable proofreaders. You know who you are.

Dedication

For Tessa, my constant source of inspiriation. Welcome to this wonderful world. May you always engage it with passion, rigor, and precision.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Privilege management is an essential security function provided by modern operating systems. It provides a way to control access to system resources. An underlying reason for managing privileges is so the system as a whole achieves some security property, such as adherence to the principle of least privilege [11]. Via mechanisms provided for privilege management, a process may, for example, temporarily raise its privilege to complete a sensitive task, and then return to its lower privilege level for its remaining tasks.

The setuid family of functions is a POSIX standard [12], and is amongst the oldest mechanisms for privilege management. It has been part of UNIX systems for several decades [16]. In systems in which setuid is used, the privilege of a process is indicated by the user and group identities (IDs) that it possesses. In this way, privilege management with `setuid()`-related functions is a sort of *process identity management*; to modify its privileges, a process assumes a different user and/or group identity. There are customarily three user IDs with which a process is associated: real, effective, and saved. A history of the evolution of these IDs is provided by Chen et al. [7]; a brief description is provided here. The real user ID is the ID of the user that invokes the program that the process executes. The effective user ID is used by the operating system for most authorization checks. In many cases, the effective user ID is the same as the real user ID, because access control is based customarily on the invoker (i.e., the user whose process is attempting access).

However, the effective and the real user IDs may differ in cases wherein, for example, a developer wants authorization to be granted based on the owner of the program rather than the invoker of the program. This is a common way of controlled privilege escalation — a process executed by an unprivileged user is able to perform privileged actions, but only within the confines of what a particular program is created to do. The saved user ID

Table 1.1: The four functions that comprise the setuid family, a brief description of each, and whether each is available on the operating systems under test. The functions `setuid()`, `seteuid()`, and `setreuid()` are standardized by POSIX, whereas `setresuid()` is not.

Name	Purpose	OS
`setuid(uid_t i)`	Sets the effective user ID of the calling process. Leaves the real and saved IDs unchanged, unless the caller is privileged.	1–5
`seteuid(uid_t e)`	Sets the effective user ID of the calling process. If the caller is unprivileged, it may set the effective to the real or saved user IDs.	1–5
`setreuid(uid_t r, uid_t e)`	Sets the real and effective user IDs.	1–5
`setresuid(uid_t r, uid_t e, uid_t s)`	Sets the real, effective, and saved user IDs.	2–4

1: Darwin 13.0.0 2: FreeBSD 9.1 3: Linux 3.8.0 4: OpenBSD 5.4 5: OpenIndiana 5.11

is used, as its name suggests, as scratch space — to save, for example, a user ID of higher privilege so the effective user ID may be set to it when needed.

When a process is created and starts to execute a program, the real, effective, and saved user IDs of the process are instantiated based on the permission settings of the program. These permission settings are stored on disk along with the program as its meta-data. The permission settings may indicate, for example, that a process that executes the program should have, as its initial effective user ID, the ID of the owner of the program and not the invoker [3].

The purpose of setuid functions is to enable a process to change its real, effective, and/or saved user IDs. For example, if the real and effective user IDs are those of an unprivileged user, and the saved user ID is that of a privileged user, then a setuid call can be invoked to set the effective user ID to the saved user ID, and leave the real and saved user IDs unchanged. This has the effect of raising the privilege level of the process. Later, the process can invoke a setuid function to set the effective user ID to be the same as the real user ID, and thereby lower its privilege level. Processes that follow this pattern are often referred to as *setuid-root processes*. Processes in the broader class of programs that invoke setuid functions to switch identities — regardless of whether one of the identities is privileged — are referred to as *setuid processes*.

There are four functions in the setuid family for changing a user ID, listed in Table 1.1.

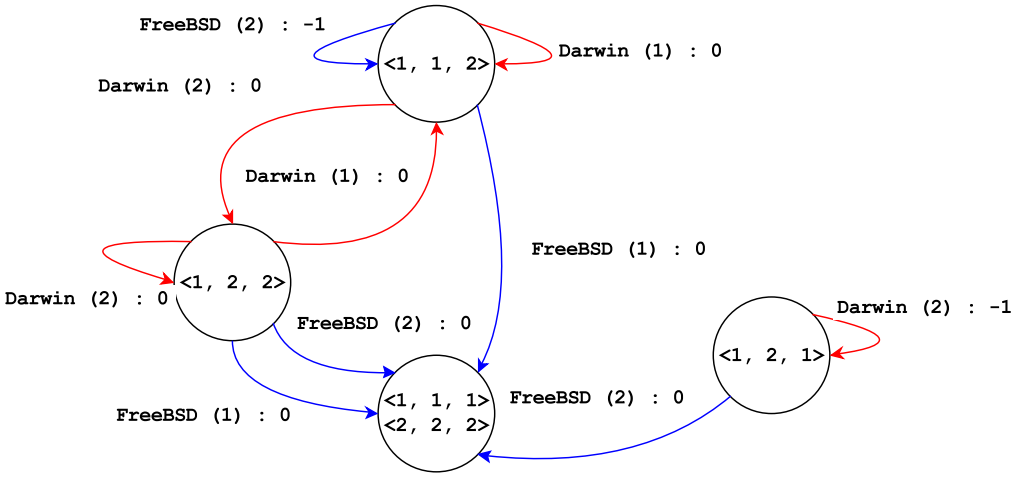

Figure 1.1: A compressed `setuid()` state graph fragment that contrasts Darwin and FreeBSD. The graph fragment contains all `setuid()` calls where: (1) the transitions on the two platforms differ, and (2) the transitions are to and from the three selected states. Nodes are labeled with `<real, effective, saved>` user IDs. Edges are labeled with `Platform (argument): return_value`. The state labeled with both `<1, 1, 1>` and `<2, 2, 2>` is a merger ("compression" — see Chapter 3) of those two states, which are equivalent. For example, in the FreeBSD transition from `<1, 2, 1>`, the actual destination state is `<2, 2, 2>`.

There are other functions in the extended family, such as those for inspecting the user IDs and manipulating group IDs. Some systems have additional, implementation-specific user IDs than just the three (real, effective, saved) discussed here. In addition, some systems implement fine-grained credentials that decouple privileges from user IDs. For example, *Linux capabilities* and *Solaris privileges* allow processes to hold privilege to perform specific actions, regardless of the user IDs associated with the process. Although these additional mechanisms do play an important role in process identity management, only the setuid family is considered here.

The three functions, `setuid()`, `seteuid()` and `setreuid()` are standardized by POSIX, although `setreuid()` is marked for deprecation. The last function, `setresuid()`, is non-standard. Despite that POSIX setuid is a decades-old standard, setuid implementations differ from system to system. An example of some `setuid()` implementation differences is given in Figure 1.1. Each edge in the graph fragment is labeled as belonging to either Darwin or FreeBSD. The graph illustrates how FreeBSD may set all three user IDs (real, effective, saved) on successful `setuid()` calls, even if the effective user ID is unprivileged.

Darwin, in contrast, sets only the effective user ID in the unprivileged case. Also notice that, on Darwin, calls to `setuid()` that are passed the current effective ID remain in the same state, but sometimes return 0 and other times -1.

Taken together, setuid implementations across different platforms are fraught with such inconsistencies, some easy to generalize and some not. For example, `seteuid()` behavior on Darwin and FreeBSD are identical, and the same is true for `setreuid()` on Linux and OpenIndiana[1]. However, the implementation differences in `setreuid()` on other platforms is very difficult to characterize succinctly. As discussed in Chapter 2, this sore spot is not surprising given several problems with the standard for `setreuid()`.

1.1 Motivation

Setuid is well-established both in the POSIX standard and in implementations, and several of its problems are well-known. It has even been characterized as "untrustworthy" [21]. Furthermore, some systems have implemented fine-grained privilege management that interacts with setuid (e.g., Linux Capabilities and Solaris Privileges).

Also, devices that rely on setuid number in the tens or even hundreds of millions [2]. Such devices include cloud-computing and e-commerce servers. Furthermore, setuid is invoked by many tools in UNIX systems, such as `mount`. Given the fundamental nature of such setuid-dependent tools, devices' reliance on setuid is unlikely to ease in the foreseeable future.

The behavior of some calls, such as `setuid()`, has changed in some systems recently. For example, `setuid(-1)` behaves differently in Linux 3.2 (released early 2012) than in 3.8 (released early 2013). It is no surprise that work on addressing issues related to setuid continues (see Jain et al. [13] for a recent example).

Therefore, it is meaningful for us to investigate the status of setuid with an intent of seeking improvements. The research community in security has exposed a number of issues with setuid in the past [7, 8, 21]. This work explores whether things have improved with setuid over the past decade or so, particularly since the work of Chen et al. [7], which discusses the issues in some depth.

[1] Unless otherwise indicated, statements about different UNIX platforms refer to the particular platform versions mentioned below Table 1.1.

1.2 Contributions

This work is broadly about investigating the current status of setuid, and challenging some of the assertions from prior work, such as those of Chen et al. [7] and Tsafrir et al. [21]. It poses and answers the following sets of questions. (1) Is the POSIX standard indeed broken as prior work suggests? That is, are the relationships described in the standard unambiguous, and does the standard specify well-formed functions? (2) Do implementations that claim to be POSIX-compliant comply with the latest standard? (3) Do the setuid functions and/or setuid wrappers from prior work provide programmers with an interface that is both correct and usable?

Towards (1), the standards are expressed using quantifier-free first-order logic with an associated semantics. (See Chapter 2 and Appendices.) Constructing this logic was not very challenging, which suggests that the standards are somewhat well-written. Of course, the standards could be encoded in some other precise syntax (e.g., functional). This choice is not critical to the work or the ensuing results.

Towards (2), the precise expression of the standards is combined with a state-space enumeration similar to that of Chen et al. [7] to assess whether five systems that claim to be POSIX-compliant adhere to the latest standard (see Chapter 3).

Towards (3), a discussion of some issues with setuid wrappers proposed in 2002 [7] and 2008 [21] is provided. Further, the aforementioned state-space enumeration is extended to include the functions from 2008 (see Chapter 4). These analyses reveal several usability and correctness problems. Finally a new interface is proposed for managing process identities, and analyzed to establish its correctness and argue its usability.

1.2.1 Summary of Observations

Several observations and results that differ from or build upon prior work are reported. For example, the notion of *appropriate privileges*, which the standard uses to capture implementation differences between standard-compliant systems, does not fully do so. Nevertheless, the presented analysis reveals that the standards for the `setuid()` and `seteuid()` are, under some reasonable assumptions, unambiguous function descriptions. Furthermore, with the exception of FreeBSD, all of the operating systems under test abide by these two standards, and a simple explanation for FreeBSD's deviation from the standard is provided.

Analysis also reveals that the standard for `setreuid()` is both ambiguous and logically inconsistent. The analysis precisely identifies the ways in which the standard is broken,

offers an interpretation of the standard that is consistent with most systems under analysis, and precisely defines the way in which other systems appear to have implemented the function. Also note that POSIX has designated `setreuid()` for deprecation [12].

Analysis of existing setuid wrappers points to several usability and correctness issues. Both wrappers do not perform sufficient sanity checks to ensure that the action implied by the function name is performed correctly. A wrapper proposed in 2002 also has a condition in which it fails silently. Finally, the wrapper proposed in 2008 will, by default, crash the process when it gets stuck in an irreparable state.

1.3 Related Work

Setuid has been part of UNIX systems for several decades [16]. Chen et al. [7] provide a concise history of the evolution of the three user IDs (real, effective and saved) discussed in this work, and the associated family of functions, `setuid()`, `seteuid()`, `setreuid()`, and `setresuid()`. Issues with setuid have been known for a long time. For example, Bishop [3] discusses how to write a program for which it is safe to set the setuid bit, so that it runs with the owner's user ID as its effective user ID, rather than the invoker's.

The work of Chen et al. [7] is the first to systematically study the setuid family, and discuss problems related to ambiguity in the POSIX standard and compliant implementations. It remains the most comprehensive study on the topic. That work considers POSIX 1003.1-1988, which, as that work asserts, standardizes the `setuid()` call only. Since then, and as of the writing of this paper, POSIX has standardized `seteuid()` and `setreuid()` as well. Also, the standard for `setuid()` has not changed in any substantive way since the publication of that paper.

The work of Tsafrir et al. [21] builds upon the work of Chen et al. [7]. Specifically, it points out that the prior work does not consider the role that groups play in an access control decisions. It then provides wrapper-functions over the setuid, setgid, and setgroups calls for a process to, for example, drop its privileges permanently.

This work builds upon the work of Chen et al. [7] and Tsafrir et al. [21], comparing new data from the presented state-space enumeration to the data reported by Chen et al. [7]. Also, the work challenges several assertions in those two pieces of work regarding the quality of the standard, and compliance of implementations. Portions of this work are published elsewhere [9].

Apart from the above pieces of work that deal directly with setuid, there is broader work related to authorization; refer to Bishop [4] for a comprehensive discussion.

1.4 Outline

The remainder of this work is organized as follows. The next chapter discusses a precise encoding of the POSIX standard (formalized in the Appendices) and an implementation-independent description of `setresuid()`, which has not been standardized. The chapter also contains analysis that shows which of these standards and documentation are unambiguous function descriptions. In Chapter 3, five setuid implementations are compared with the standards and documentation. In addition, the chapter contains analysis of the usability and correctness of two process identity management interfaces that wrap setuid. In Chapter 4 two alternative interfaces that wrap setuid are discussed and and a third based on experiments presented in Chapter 3 is described. Finally, Chapter 5 summarizes conclusions drawn from this work and opportunities for future work are identified.

Chapter 2

Setuid Standard

This work assesses the latest POSIX standard [12] for setuid as to whether it provides unambiguous, logically consistent function descriptions. In this context, *unambiguous* means that the standard text does not have more than one reasonable interpretation, *logically consistent* means that there are no contradictions, and *function description* means that the relationships described in the standard define mathematical functions (i.e., a mapping from each possible input to exactly one output). The first criterion, unambiguity, is somewhat subjective; the other two criteria are objective.

Structure of the standard The text in the standard for each setuid function has four parts: (1) the name and signature of the function, (2) a description of function behavior, (3) a description of the return value, and (4) a list of error conditions which may cause the function to fail. This work does not consider other sections, such as *rationale* and *change history*, that discuss the evolution of the standard. The setuid functions standardized by POSIX have three features in common:

- Return a value of `0` to indicate success;
- Return a value of `-1` to indicate failure;
- Fail for one of two reasons:
  - The `EINVAL` error occurs if any function argument is invalid or out-of-range,
  - The `EPERM` error occurs if the process does not have permission to set user IDs to the requested values. This error can only occur when a process is unprivileged.

How the standards were assessed As the standards are written in a natural language, interpreting them is inherently a subjective exercise. The first task was to express the standard for each function using a precise syntax that lends itself to a precise semantics. The standard has been translated using quantifier-free first-order logic, with an associated semantics.

This representation was chosen for two reasons: (1) the sentences of the standard describe *relations* over process UID states and function inputs, (2) these relations are expressed in terms of value equality/inequality. The results from this exercise are presented in the Appendices. Translation of standards from a natural language to logic is not novel. There is work, for example, in translating standards for communication devices to logic for the purpose of verification [1, 19].

For each setuid function, there is a set of formulas. The intent is that the formulas are true facts about any invocation of the function. The set of facts for a function together specify when an invocation of the function succeeds, and when it fails. Process state is modeled as a three-tuple of the process' real, effective, and saved user IDs. Setuid function calls are modeled as stateful functions that can manipulate these three values. Each function has a set of explicit arguments as defined by the standard. In addition, the standards contain several implementation-specific parameters. The most conspicuous of these parameters is "appropriate privileges," which is discussed at length by Chen et al. [7]. Implementation-specific parameters are discussed below, after this discussion of the translation. Although these components (state, arguments, and parameters) are sufficient to capture the meaning of statements in the standards, additional intermediate variables are introduced for clarity.

Consider the following passage from the standard for `setuid()` and the corresponding translation to logic. Prior work [7] points to `setuid()` as a particularly problematic function in the setuid family. The example serves to illustrate the translation process, and to convince the reader that the translation process is sound in that a reasonable interpretation of the standard has been adopted.

From the standard:

> If the process does not have appropriate privileges, but *arg_uid*[1] is equal to the real user ID or the saved set-user-ID, *setuid()* shall set the effective user ID to *arg_uid*; the real user ID and saved set-user-ID shall remain unchanged. [12]

[1]The argument name used in the standard is actually *uid*. Its name is changed to *arg_uid* to avoid confusion between propositional variables in logical expressions.

Translation[2]:

$$success\_nap \leftrightarrow \quad rtn = 0 \ \wedge \ new\_ruid = old\_ruid \wedge \\ new\_euid = arg\_uid \ \wedge \ new\_svuid = old\_svuid \tag{2.1}$$

$$success\_nap \leftarrow \quad \neg AP \ \wedge \ (old\_ruid = arg\_uid \vee \\ old\_svuid = arg\_uid) \tag{2.2}$$

The prose applies to the case that the process does not have appropriate privileges. The boolean variable (propositional atom) *success_nap* indicates a successful call by a process that does not have appropriate privileges. Formula (2.1) expresses the effect, and serves to define *success_nap*. Formula (2.2) expresses the conditions under which *success_nap* are expected to occur. Notice that the use of the form, "If [condition], [action]" translates to implication, not equivalence, because the text does not state that *success_nap* should not occur when the given condition is false.

Implementation-specific parameters It can be argued that, for application developers, it is ideal that all setuid functions have well-defined implementation-independent standards. However, in reality, systems developers usually care more about backwards-compatibility than with highly refined function behavior. It can be argued, then, that the ideal of implementation independence is unattainable for POSIX for two reasons: (1) there are setuid implementation differences between some of the earliest UNIX systems, and (2) many modern systems have moved away from the mindset of the *all-powerful root user*, preferring to divide power into smaller units. An example of this is the notion of capabilities in Linux [15]. In Linux, assigning the root user ID to the effective user ID of a process activates all capabilities. However, processes can be given individual capabilities even when their effective user ID is not root. For example, a process may obtain the `CAP_SETUID` capability, which allows the user to "make arbitrary manipulations of process UIDs" [15].

POSIX's way of dealing with this has been to abstract unrestricted setuid privileges in the phrase "appropriate privileges." Prior work has suggested that this abstraction is a cause of confusion about setuid implementation differences [7]. Based on analysis of the standard, appropriate privileges is not to blame for this confusion. Indeed, it is not difficult to give a crisp definition of appropriate privileges on most platforms; for example, on Linux, a process has appropriate privileges if and only if the process has `CAP_SETUID` in its set of effective capabilities.

[2]The given equations are duplicates of A.10 and A.11

Table 2.1: Implementation-dependent parameters in the POSIX setuid standards, as they are defined in the logic system in the Appendices.

Name	Description
AP	Appropriate privileges.
$IsUID(id)$	The value, id, is not a valid user ID on the system.
$RuidIsPermitted(id)$	Given the current privileges of the process, the value of id is permitted by the system as real user ID argument for `setreuid()`.

The notion of appropriate privileges can make the standard easier to read for the following reason. If all implementation-dependent behavior is captured by the notion of appropriate privileges, then the constraints of the standard + a definition of appropriate privileges suffice to precisely define function behavior. Unfortunately, as discussed below, appropriate privileges is not the only implementation-dependent parameter in the setuid standards. A complete list of such parameters is presented in Table 2.1.

Special user IDs It is worth noting that `setreuid()` and `setresuid()` treat an argument value of `-1` specially. Informally, this value indicates, "do not change the corresponding user ID". Unfortunately, the standards for `setuid()` and `seteuid()` make no mention of how an argument value of `-1` should be treated. By this omission, it is presumed that POSIX intends such cases to fall under the purview of implementation-dependent parameters. Security implications of this choice are discussed in Section 3.2.

2.1 Unambiguity of the Standards

This section discusses the first of three properties for the standard: unambiguity. The `setuid()` and `seteuid()` standards are unambiguous. All references to function arguments and state variables have a clear meaning.

However, the standard is ambiguous for `setreuid()`. For example, the standard refers to the real user ID state variable in different instances as "new real user ID", "current real user ID", and "real user ID". It is unclear whether the third term refers to the state

variable at the time the function is invoked, or upon return. The same issue arises in the use of the phrase "either the real, effective, or saved user ID."

Through various iterations of setuid state graph verification (see Chapter 3), the meaning of ambiguous phrases that minimizes the number of systems under test that violate the `setreuid()` standard has been deduced. By "real user ID" the standard appears to be most widely interpreted as the *current* real user ID (i.e., the real user ID value prior to function invocation). By "either the real, effective, or saved user ID" the standard appears to be most widely interpreted as the respective *current* IDs.

2.2 Logical Consistency of the Standards

The second criterion for assessing the standard is logical consistency — the absence of contradictions.

The standards for `setuid()` and `seteuid()` are logically consistent. The standards describe the following four broad categories of behavior: (1) function success for a process with appropriate privileges, (2) function success without appropriate privileges, (3) function failure due to lack of appropriate privileges, and (4) function failure due to invalid input arguments. As such, the class of relations described in the standard can be contradictory in two ways: some set of input arguments may entail (1) different function behaviors (e.g., the function both succeeds and fails), and/or (2) some implementation-dependent parameter is both true and false. In the case of `setuid()` and `seteuid()`, no two behaviors overlap, and that no parameter contradictions are entailed. Therefore, these standards are logically consistent.

The `setreuid()` standard is logically inconsistent. The standard explicitly allows processes without appropriate privileges to set the effective user ID to any one of the current real, effective, or saved user IDs. However, the `EPERM` condition specifies that an attempt by processes without appropriate privileges to set the effective user ID to any value other than the real or saved user ID is an error. As such, the standard contains contradictory success/failure conditions for certain inputs. For example, if `<ruid, euid, svuid> = <0, 1, 2>` and `setreuid(0, 1)` is invoked, the main function description suggests that the call should succeed whereas the errors section suggests that the call should fail.

2.3 Standards as Function Descriptions

The third criterion is whether the standards specify well-formed functions, that is, a mapping from every possible input to a single output.

The standards for `setuid()` and `seteuid()` describe well-formed functions. The four categories of behavior described in the previous section cover all possibilities. Furthermore, each category describes precisely what the function's return value and the process-state should be upon return. Therefore, these standards are complete. Taken together, the standards are indeed function descriptions.

In the case of `setreuid()`, in addition to the logical inconsistencies described in the previous section, the standard does not specify the behavior for all function inputs. Formula 2.3 below[3] is the only one that defines the behavior of the saved user ID when the function returns successfully. The problem is that the formula specifies an "only if" condition, and not an "if and only if". That is, the expression evaluates to $\top$ (i.e., "true"), even if the expression to the right of the inference is false. The new saved user ID is not constrained to a single value when the right-hand side of the formula is false.

$$\begin{aligned}(new\_svuid = new\_euid) \leftarrow & (arg\_ruid \neq -1 \vee \\ & (arg\_euid \neq -1 \wedge \\ & arg\_euid \neq old\_ruid))\end{aligned} \tag{2.3}$$

2.4 Fixing the Standard

Based on observations in the three previous sections the standards are broken in two ways. One is the ambiguity, contradictions, and incompleteness of `setreuid()`. The other is the extraneous implementation-dependent parameters outside the context of appropriate privileges. There seem to be two conflicting perspectives on whether the standards are easily fixed.

On one hand, these problems are seemingly easy to fix. A previous section offered an interpretation of ambiguous statements that maximizes compliance of five modern UNIX-like systems. Contradictions in `setreuid()` can be resolved by adopting one of the two descriptions for a valid *arg_euid* argument, and its functional description can be completed by changing the saved user ID requirements to read "if and only if." Finally, extraneous

[3] Formula 2.3 is a simplified form of Formula A.21 in the Appendix.

implementation-dependent parameters can be dropped by specifying: (1) *arg_ruid* values for `setreuid()` in the unprivileged case that are acceptable, and (2) user IDs that should trigger the `EINVAL` error condition for each function.

On the other hand, however, these changes would break standards compliance for several implementations. At least three `setreuid()` implementations (FreeBSD, Linux, and OpenIndiana), and two `setuid()` and `seteuid()` implementations (Darwin, and OpenIndiana) would no longer comply.

2.5 Setresuid

As mentioned in Chapter 1, POSIX standardizes only three of the setuid functions under investigation: `setuid()`, `seteuid()`, and `setreuid()`. Three of the five operating systems under test additionally support `setresuid()` (see Table 1.1 in Chapter 1). A careful study of `setresuid()` in prior work [7, 21] and the documentation for it in the three systems that support it [10, 14, 17] reveals that all of them specify the same semantics for it with respect to processes with and without appropriate privileges. That is, even where the specifications for the function's behavior may differ, they do not differ for success and failure conditions for a process with appropriate privileges, or for a process without appropriate privileges. Any difference is only in what constitutes appropriate privileges. This work refers to their common specification as the *consensus standard* for `setresuid()`.

The consensus standard for `setresuid()` is an unambiguous, logically consistent function description. All terms have a clear and consistent meaning, the input conditions for each output behavior are non-overlapping, and every input maps to an output. This clarity with `setresuid()` has been pointed out by prior work [7, 21], and the investigation presented in Chapter 3 verifies that the three `setresuid()` implementations under test conform to the consensus standard. Nevertheless, cross-platform applications cannot rely on `setresuid()` alone, making other analyses in this work crucial to such applications, because, as mentioned above, (1) it is not standardized, and (2) not all systems support it.

Chapter 3

Setuid Implementations

To evaluate setuid standards compliance a state-space enumeration similar to the one described by Chen et al. [7] was performed on the five operating systems mentioned in Table 1.1. The procedure systematically explores all possible user ID states of a process, and the state-transitions from each state to other states, under certain assumptions. The procedure assumes that, with the exception of `uid = -1` — implicitly or explicitly cast to `uid_t`, discussed below — all unprivileged user IDs have the same restrictions placed on their setuid calls. It further assumes that no new user IDs other than those listed as function arguments are introduced in the resulting state.

As noted in Table 1.1, Darwin and OpenIndiana do not implement `setresuid()`. As one might expect, they similarly do not implement `getresuid()`. To look up the saved user ID on Darwin, the state-space explorer invokes `proc_pidinfo()`, which fills out a `struct` containing, among other things, the current saved user ID of the process. To look up the same datum on OpenIndiana, the program reads it from the `/proc/[pid]/cred` file.

A point of curiosity during this exercise was how the `EINVAL` error is triggered in each system. Before completing a full state-space enumeration, two hypotheses were manually tested: (1) systems will return `EINVAL` when a negative user ID is applied to any setuid function (with the exception of passing `-1` to `setreuid()` and `setresuid()`), and (2) systems will return `EINVAL` when `-1` is passed to `setuid()` and `seteuid()`. Hypothesis (1) proved to be false; this is not surprising, given the `uid_t` is `unsigned int` on all five platforms. Hypothesis (2) holds for Linux and OpenIndiana, but not for Darwin, FreeBSD, and OpenBSD. In fact, Hypothesis (2) accounts for *all* instances of `EINVAL` in the enumeration.

The above-mentioned observations led to two key differences between the state-space exploration presented here (henceforth referred to *this enumeration*) and that of Chen et al. [7]. This enumeration investigates five systems, whereas the prior work investigated three. Also, Chen et al. [7] do not present findings regarding transitions between non-root users. This enumeration is performed over eight user IDs: -1, 0, and 1 through 6. User IDs -1 (only supported by some systems) and 0 (the super-user) are treated specially. Six additional unprivileged user IDs are included to cover state transitions such as: $\langle 1, 2, 3\rangle \rightarrow$ `setresuid(4, 5, 6)` $\rightarrow \langle ?, ?, ?\rangle$. Constructing this larger graph ensures that, under the above-mentioned assumptions, the enumeration explores all possible states. The output of the state-space enumeration is a graph, G_{OS} as follows:

$$
\begin{aligned}
&U = \{-1, 0, 1, 2, 3, 4, 5, 6\} \\
&G_{OS} = \langle V, E\rangle \\
&V = \langle \text{ruid}, \text{euid}, \text{svuid}\rangle \\
&E = \langle \text{function, arguments, return\_value}\rangle \\
&\text{function} \in \{\texttt{setuid()}, \texttt{seteuid()}, \texttt{seteuid()}, \\
&\qquad\qquad \texttt{setreuid()}, \texttt{setresuid()}\} \\
&\text{every argument} \in U
\end{aligned}
$$

The assumptions suggest that the graph could be compressed; one could, for example, deem that `<1,2,2> = <6,4,4>`. This compression would save space and require that edges be labeled with a remapping of canonical user IDs. Figure 1.1 shows a fragment of a "difference" graph (generated automatically) that contains such a compression. The graph fragment shows some of the differences between Darwin and FreeBSD for `setuid()`.

3.1 Standards Compliance

Once the setuid state graphs for all five systems were constructed, the logic of the standards[1] was reified as C++ code that inspects state transitions for standards compliance. Then, standards compliance of the five graphs was checked against the reified logic. The results of this experiment are reported in Table 3.1. As a follow up experiment, constraints were relaxed in the standard to better understand the nature of non-compliant implementations.

[1]In the case of `setreuid()`, the more permissive interpretation of acceptable *arg_euid* values from the DESCRIPTION portion of the standard is applied. Naturally, applying the unmodified standard in its logically inconsistent state would fail.

Checking for contradictions related to function output was straightforward: the proposition $(success \ \wedge \ \neg fail) \vee (fail \ \wedge \ \neg success)$ must hold for all edges.

Checking implementation-dependent parameters was more nuanced. In the case of $\neg IsUID(.)$, observe that if a call to function, f, with arguments, A, triggers an `EINVAL` error in one context, it is because $\exists\, a \in A \,.\, \neg IsUID(a)$. Therefore, if $f(A)$ triggers `EINVAL` in one context, then it must trigger `EINVAL` in any context. To check the consistency of `EINVAL`, a list of $\langle f, A \rangle$ pairs that trigger the error was maintained. After visiting the entire graph once, the graph was revisited to confirm that no such pair triggers the error in one context but not another. Note that this proves a weak condition on the full set of function parameters. It does not prove which argument value triggers `EINVAL`, or whether the offending argument is treated inconsistently by the system. As discussed below, the identity and consistency of invalid user IDs become self-evident when each system's instances of `EINVAL` are manually inspected.

To check AP (appropriate privileges), the formulas constructed from the standard were manipulated to derive facts that necessarily entail either AP or $\neg AP$. Then the fact $\neg(AP \ \wedge \ \neg AP)$ was checked on all edges. In the case of `setreuid()`, where such facts contain the implementation-dependent condition $RuidIsPermitted(arg\_ruid)$, the weak condition that $\neg(AP \ \wedge \ \neg AP)$ holds for either $RuidIsPermitted(arg\_ruid) = \top$ or $RuidIsPermitted(arg\_ruid) = \bot$ (where $\top$ and $\bot$ indicate `true` and `false`, respectively) was checked. The same procedure was used to check $RuidIsPermitted(arg\_ruid)$ with this weak condition applied to AP.

3.1.1 Results

FreeBSD's `setuid()` implementation is out of date. In previous revisions of the POSIX standards, POSIX introduced a parameter called `_POSIX_SAVED_IDS`, which was to be set to either "defined" or "undefined". POSIX now mandates that implementations follow the behavior for which `_POSIX_SAVED_IDS` is defined, but FreeBSD's `setuid()` function implements the `_POSIX_SAVED_IDS`-undefined case in the old standard.

Darwin and OpenBSD's `setreuid()` implementations violate the standard for setting the saved user ID. Rather than following the rule expressed by Formula 2.3 from Section 2.3 (or, in the complete standard logic, Formula A.21 from the Appendices), Darwin and OpenBSD follow the rule expressed by Formula 3.1 and Formula 3.2, respectively. On Darwin, a successful `setreuid()` call either leaves the saved user ID unchanged, or sets it to the previous effective user ID. On OpenBSD, a successful `setreuid()` call either leaves the saved user ID unchanged, or sets it to the new real user ID.

Table 3.1: POSIX setuid standard [12] compliance for five modern operating systems. "Yes" indicates standard compliance, and "No" indicates deviation from the standard.

System\Function	`setuid()`	`seteuid()`	`setreuid()`
Darwin	Yes	Yes	**No**
FreeBSD	**No**	Yes	Yes
Linux	Yes	Yes	Yes
OpenBSD	Yes	Yes	**No**
OpenIndiana	Yes	Yes	Yes

$$svuid\_success \leftrightarrow \quad new\_svuid = old\_svuid \ \vee \ new\_svuid = old\_euid \tag{3.1}$$

$$svuid\_success \leftrightarrow \quad new\_svuid = old\_svuid \ \vee \ new\_svuid = new\_ruid \tag{3.2}$$

Note that the above two rules are not a complete description of the behavior of `setreuid()` on the saved user ID. The exact rules for Darwin and OpenBSD are considerably more complex. Given that the `setreuid()` standard is an incomplete description of a function that has been marked for deprecation by POSIX, the above two rules are presented in the interest of clarity.

3.2 Security Implications

This section discusses implications of the presented findings to the security of a system that uses setuid. One issue is that any setuid program that assumes that a platform complies with the latest standard, when it does not, may be vulnerable to attack. This appears to be the case with FreeBSD's implementation of `setuid()`, and with Darwin and OpenBSD's implementations of `setreuid()` (see Table 3.1 and Section 3.1.1).

Implementation differences with respect to when the error `EINVAL` is returned are likely also problematic. A specific case of this is the mismatched semantics of `-1` as an argument (implicitly or explicitly cast to `uid_t`) between the pairs of functions `setuid()` and `seteuid()`, and `setreuid()` and `setresuid()`. The mismatched semantics can, for example, cause a process to reasonably assume that it has dropped privileges when it has not.

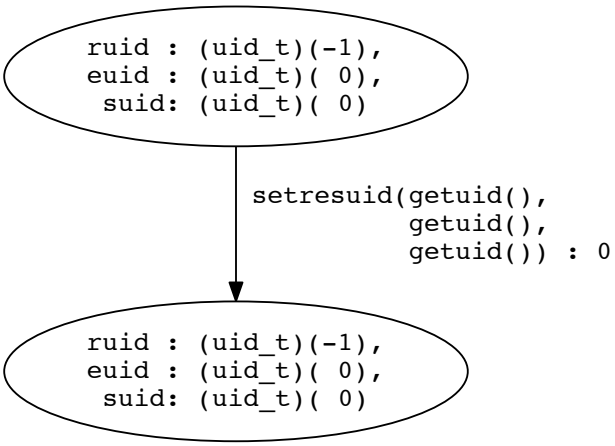

Figure 3.1: An example of implementation differences for the `EINVAL` error being potentially problematic. It is reasonable to expect the `setresuid()` call in the figure to cause the process to downgrade itself to the real user ID.

An example is shown in Figure 3.1. In the figure, a process is in the state in which its real, effective, and saved user IDs are $\langle -1, 0, 0 \rangle$, each typecast to `uid_t`. Now, a call to `setresuid()` as shown in the figure does not downgrade the process to $\langle -1, -1, -1 \rangle$. One would reasonably expect an invariant to be respected that when the real user ID of the process is non-root, and such a call is made, the process is permanently downgraded to its real user ID.

Chapter 4

Alternative Interfaces

Prior work [7, 21] has suggested that UNIX process identity management functions are: (1) often misused, suggesting usability problems, and (2) designed to support temporarily dropping/restoring privileges and permanently dropping privileges. In light of these observations, the prior work proposes two alternative interfaces, one that wraps the setuid family of functions and one that wraps an extended family user and group ID management functions. To extend the analysis of setuid these two wrappers are analyzed for usability and correctness. The issues found through this analysis motivated the implementation of a different wrapper interface (presented in Section 4.1).

Chen et al. [7] provide two versions of their interface; one for use with systems that support `setresuid()`, and another for systems that do not. The issues discussed below manifest consistently in the former implementation.

Both Chen et al. [7] and Tsafrir et al. [21]'s interfaces do not perform sufficient sanity checks on function arguments. There is no check to ensure that the argument to `drop_priv_perm()` (which is used to drop privileges permanently) or `drop_priv_temp()` (drop privileges temporarily) cause the process to indeed drop privileges. In Chen et al. [7]'s case, if a privileged process invokes one of these functions with an argument value of `-1` or `0`, then the privilege-dropping functions return `0` without dropping appropriate privileges. The same probelem arises in Tsafrir et al. [21]'s privilege-dropping functions for an argument value of `0`, though a sanity check is performed for a value of `-1`.

Chen et al. [7]'s functions that rely on `setresuid()` contain another bug. If a process calls `restore_priv()`, perhaps mistakenly, after calling `drop_priv_perm()`, `restore_priv()` erroneously returns `0`, implying that privileges have been restored.

Tsafrir et al. [21]'s interface also contains a serious usability issue. The interface is written to "do its best" to switch to the requested user identity. When `setresuid()` is not available this can entail making two setuid-like function calls (the same goes for setgid-like functions). Unfortunately, the second call may fail, leaving the process in a potentially irreversible intermediate state. The interface authors' way of dealing with this problem is to call `abort()` on any system call failure, crashing the process[1]. While this approach is secure, it is also highly unusable.

4.1 Proposed Interface

This section proposes an alternate interface for user ID management. The interface is similar to Chen et al. [7]'s and Tsafrir et al. [21]'s insofar as it is designed with the notions of "temporary" and "permanent" identity changes in mind. Indeed, the implementation shares some code with theirs [20]. However, unlike Tsafrir et al. [21]'s interface, this implementation deals exclusively with user IDs, and not groups. An extension to properly manage groups is left as future work. The implementation will soon be available for download [18].

The hallmark usability consideration of the interface design is the following observation: a setuid-like interface should be usable for root-setuid processes as well as non-root-setuid processes. Nevertheless, wrappers from prior work use the language of dropping and restoring privileges. In light of this observation, the proposed interface comprises the two functions: `change_identity_permanently(uid)` and `change_identity_temporarily(uid)`. This puts the onus on the developer to correctly select a privileged or unprivileged identity. The function names do not imply that privileges will be dropped or restored.

The interface is designed to provide the following guarantees: (1) internally consistent success or failure[2]: return `0` on success, or return `-1` and set `errno` appropriately without changing any user identity state on failure; (2) in the `_permanent` case, upon success, set all three user IDs to the specified value; and (3) in the `_temporary` case, upon success, set the effective user IDs to the specified value and store the previous effective ID in either the real or saved ID location.

[1] Tsafrir et al. [21] provide a C-style `#define`, LIVING_ON_THE_EDGE, to allow users of the interface to get a return of `-1` instead of crashing, but this does not solve the intermediate state problem.

[2] The implementation can be compiled with or without threading support, which wraps interface functions in a basic locking mechanism to prevent concurrent identity changes on different threads. Naturally, this does not guarantee correct behavior in multi-threaded environments where setuid functions are manually invoked, or where interface functions are invoked within interrupt handlers.

The greatest challenge in implementing such an interface is dealing with systems that do not support `setresuid()`. In such cases, the implementation must rely on potentially multiple platform-dependent system calls to change identities. In light of the divergent — sometimes non-standard — setuid implementations discovered during the experiments presented in Chapter 3, the proposed interface exploits a platform-specific call graph to correctly execute the desired behavior.

The implementation uses a shortest path matrix to efficiently look up relevant details of the call graph. This matrix can be automatically computed offline for each version of each operating system where the implementation is to be deployed. The computation repeatedly invokes Boost's implementation of Dijkstra's shortest path algorithm [5], which outputs a *predecessor map* for a given starting vertex. The resulting predecessor maps, one for each choice of source vertext, are stored as a shortest path matrix for quickly looking up the sequence of predecessors to get from any source vertex to any destination vertex. When a change-identity function is called, the implementation first uses the shortest path matrix to determine the feasibility of the identity change. If the change is feasible, the implementation employs the matrices to follow the shortest path to the desired state; if not, it returns an error without touching the process' current privileges.

The proposed change-identity algorithms are described in Algorithms 1, 2, and 3. Note that **ChangeIdentityPermanently** and **ChangeIdentityTemporarily** are analogous to `change_identity_permanently(uid)` and `change_identity_temporarily(uid)`, respectively. Also note that the **Path** function looks up a *shortest* path using a predecessor map. Both algorithms have an implicit input parameter, G, that represents the user ID state graph.

In order to use the user ID state graphs, the actual user IDs at runtime must be normalized to the IDs used in the graph (hence the **UIDMap** and **Normalized*** functions). Two user IDs, `0` and `-1`, are not normalized. As mentioned in Chapter 3, all other users are treated as having interconvertible privileges. As such, the implementation converts each non-zero-and-non-negative-one "actual ID" to a "normalized ID" within the range presented in Chapter 3. When setuid functions are called, IDs are converted back to their actual values before the functions are invoked (hence the *uidMap* parameter for **InvokeSetuidFunction**).

If a system does not support user ID `-1`, then any attempt to lookup a user ID state containing `-1` will fail. In such cases, the implementation returns the `EINVAL` error. In this way, invoking the interface with a user ID of `-1` can succeed only on systems that support

setting the ID to `-1`, and will fail on all other systems by construction.

Algorithm 1: **ChangeIdentityPermanently**(uid, G)

$cState \leftarrow$ **CurrentUIDs**(uid)
$uids \leftarrow \{x \in cState\} \cup \{uid\}$
$uidMap \leftarrow$ **UIDMap**($uids$)
$ncState \leftarrow$ **NormalizedState**($cState$, $uidMap$)
$nUID \leftarrow$ **NormalizedUID**(uid, $uidMap$)
$ntState \leftarrow \langle nUID, nUID, nUID \rangle$
return GoToState($ncState$, $ntState$, $uidMap$, G)

Algorithm 2: **ChangeIdentityTemporarily**(uid, G)

$cState \leftarrow$ **CurrentUIDs**(uid)
$uids \leftarrow \{x \in cState\} \cup \{uid\}$
$uidMap \leftarrow$ **UIDMap**($uids$)
$nUIDSet \leftarrow$ **NormalizedUIDSs**($uidMap$)
$ncState \leftarrow$ **NormalizedState**($cState$, $uidMap$)
$nUID \leftarrow$ **NormalizedUID**(uid, $uidMap$)
$\langle a, b, c \rangle \leftarrow ncState$
$ntStates \leftarrow \{\langle x, y, z \rangle : y = uid \wedge ((x = b \wedge z \in \{a, b, c\}) \vee (x \in \{a, b, c\} \wedge z = b))\}$
$ntState \leftarrow x : x \in ntStates, \quad \forall_{y \in ntStates} |\textbf{Path}(ncState, x)| \leq |\textbf{Path}(ncState, y)|$
return GoToState($ncState$, $ntState$, $uidMap$, G)

Algorithm 3: **GoToState**($ncState$, $ntState$, $uidMap$, G)

if *not* ***StateExists***($ntState$, G) **then**
    $errno \leftarrow$ EINVAL
    return -1
if *not* ***PathExists***($ncState$, $ntState$, G) **then**
    $errno \leftarrow$ EPERM
    return -1
$path \leftarrow$ **Path**($ncState$, $ntState$, G)
foreach $functionCall \in path$ **do**
    InvokeSetuidFunction($functionCall$, $uidMap$)
return 0

In **ChangeIdentityTemporarily**, a state with the shortest available path is chosen for the sake of efficiency.

The runtime of **ChangeIdentityPermanently** and **ChangeIdentityTemporarily** is $O(|\textbf{Vertices}(G)|)$. This is clear from the following observations: (1) $|uidMap| \leq 4$, making normalization and normalized ID lookup $O(1)$; (2) $|ntStates| \leq 5$; (3) $|path| \leq |\textbf{Vertices}(G)|$, and the same applies to all shortest paths in G. In practice, shortest paths of interest — when they exist — contain two edges or fewer. The worst case is starting in a state with with an unprivileged user ID, and requesting a change to a different unprivileged user ID that appears nowhere in the current user ID state. When such a change is possbile, it requires one call to elevate privilege and a second call to set the effective user ID (and possibly the other IDs) to the requested value.

Resilience to change During data collection on different machines it was observed that Linux has changed its behavior for user ID `-1` at least as recently as the last three years. Darwin and FreeBSD have a common BSD ancestry, but FreeBSD supports `setresuid()` whereas Darwin, which was more recently introduced into the UNIX/BSD ecosystem, does not. Suffice it to say that setuid behaviors do change. One of the advantages to the proposed user ID call graph approach is its resilience to change. The graph can be generated for any version of any POSIX-compliant kernel and, with virtually no code changes[3], the interface will work properly.

Converting existing code It is reasonable to expect that switching to the proposed interface is straightforward for developers familiar with their existing code. In such instances, developers can readily identify whether setuid calls constitute attempts to change identity permanently or temporarily. Unfortunately, automating this process may prove difficult due to the strange ways in which setuid invocations are sometimes used (e.g., checking the return value of `setuid()` to test whether privileges have been permanently dropped [6]).

[3] A `#define` is used in the generator code to declare which platforms do not support `setresuid()`. Naturally, legacy systems may also require changes to `#include`s as well.

Chapter 5

Conclusions and Future Work

Prior work has suggested that standardized setuid functions are broken, and that the non-standard `setresuid()` has superior (i.e., more straightforward) semantics. This investigation has revealed problems with both the standard itself and standard compliance. The standards for `setuid()` and `seteuid()` are, for the most part, well-written. The standard for `setreuid()` is ambiguous, logically inconsistent, and incomplete as a description of a well-formed function. Given that the standard requires "appropriate privileges" (or an abstraction like it) to allow for some flexibility, this work concludes that the biggest overall problem with how the standards are written is that this abstraction does not entirely encapsulate implementation-defined parameters of the setuid family of functions. These issues can be fixed with straightforward changes to the standard, but such changes would make at least five current function implementations non-compliant.

With respect to standards compliance, FreeBSD has not updated its `setuid()` implementation to abide by the latest standard, and Darwin and OpenBSD do not abide by the prescription for `setreuid()` saved user ID behavior. Also, Darwin, FreeBSD, and OpenIndiana do not treat `-1` as an invalid ID, leading to problematic differences between `setuid()` and `seteuid()`, and `setreuid()` and `setresuid()`.

In terms of usability, missing implementations for `setresuid()` on Darwin and OpenIndiana complicates the construction of an interoperable interface for process identity management. Through evaluation of two such interfaces this work confirms that building something that is both usable and correct on top of setuid is not straightforward. Nevertheless, preliminary testing of the proposed interface suggests such an interface is possible to construct when platform-specific behaviors are integrated into the implementation in a consistent manner.

The investigation presented here suggests several directions for future work. The current implementation does not use a compressed graph, which could, if implemented, reduce space requirements. Also, the state graph-based approach could be extended to additionally manage group IDs. Finally, the interface could employ platform-specific credential management systems, specifically Linux capabilities and Solaris privileges, when state graph lookup suggests the requested identity change is infeasible. These research directions may serve to further unify the fragmented interface for POSIX process identity management.

APPENDICES

The tables below describe setuid function standards, a representation in quantifier-free first-order logic, and an associated semantics. The standard texts are from POSIX 1003.1 Base Specification Issue 7 [12]. Some input argument names are changed for clarity, and formula numbers are referenced in the standard text (in parentheses) to indicate the passage(s) from which a particular formula is derived. The following assumptions, which may not be self-evident in the text, are made:

1. Function success entails both a return value of 0 and a change in user process user IDs consistent with the "DESCRIPTION" section of the function standard.
2. Function failure entails not only a return value of -1 and appropriate setting of `errno`, but also no change in the state of the process user IDs.
3. The term "any value", when applied to a user ID, is interpreted to mean "any valid user ID value".

Table A.1: Semantic definitions of syntactic elements in logic formulas.

Syntactic Element(s)	**Element Type**	**Semantics**
0	Integer constant	The number 0; either the root/superuser UID or a function return vaule that indicates success (depending on the context).
-1	Integer constant	The number -1; a function return value that indicates failure or a "do not change UID" input argument value.
Continued on next page		

Table A.1 – continued from previous page

Syntactic Element(s)	**Element Type**	**Semantics**
arg_uid	Integer Variable (Int. Var.)	The only input argument to `setuid()` and `seteuid()`.
arg_ruid	Int. Var.	The first input argument to `setreuid()`.
arg_euid	Int. Var.	The second input argument to `setreuid()`.
old_ruid	Int. Var.	The real UID of the process before function invocation.
old_euid	Int. Var.	The effective UID of the process before function invocation.
old_svuid	Int. Var.	The saved UID of the process before function invocation.
new_ruid	Int. Var.	The real UID of the process directly following function invocation.
new_euid	Int. Var.	The effective UID of the process directly following function invocation.
new_svuid	Int. Var.	The saved UID of the process directly following function invocation.
rtn	Int. Var.	The return value of the invoked function.
AP	Boolean Variable (Bool. Var.)	An implementation-dependent parameter. True if and only if the process has *appropriate privileges*.
success	Bool. Var.	Indicates whether or not the function succeeded with no errors.
fail	Bool. Var.	Indicates whether or not the function failed due to an error.
einval	Bool. Var.	Indicates that the `EINVAL` error occurred.
eperm	Bool. Var.	Indicates that the `EPERM` error occurred.
ruid_success	Bool. Var.	Indicates correct behavior of the real UID for `setreuid()` function success.
euid_success	Bool. Var.	Indicates correct behavior of the effective UID for `setreuid()` function success.
svuid_success	Bool. Var.	Indicates correct behavior of the saved UID for `setreuid()` function success.

Continued on next page

Table A.1 – continued from previous page

Syntactic Element(s)	**Element Type**	**Semantics**
$arg\_euid\_success$	Bool. Var.	Relates $arg\_euid\_is\_valid$ (see below) with AP and the case that the *euid* argument is -1.
$arg\_euid\_is\_valid$	Bool. Var.	Indicates that the arg_euid argument passed to `setreuid()` is permissible for a process without appropriate privileges.
$arg\_euid\_is\_invalid$	Bool. Var.	Indicates that the arg_euid argument passed to `setreuid()` is not permissible for a process without appropriate privileges.
$einval$	Bool. Var.	Indicates that an `EINVAL` error occurs in a call to `setreuid()`.
$eperm$	Bool. Var.	Indicates that an `EPERM` error occurs in a call to `setreuid()`.
$= \neq$	Binary Predicates	True if and only if the value on the left and right are equal, or not equal (respectively).
$RuidIsPermitted(.)$	Unary Predicate	True if and only if its integer argument is a real UID that is "permitted by the implementation".
$IsUID(.)$	Unary Predicate	True if and only if its integer argument is a valid UID in the system.
$\neg \wedge \vee \rightarrow \leftrightarrow$	Boolean Connectives	The Boolean connectives: *negation*, *and*, *or*, *impliciation*, *equivalence* (respectively).

Table A.2: Formulas common among `setuid()`, `seteuid()`,`setreuid()`, and `setresuid()`.

Logic Expressions	
$(success \ \wedge \ \neg fail) \ \vee \ (fail \ \wedge \ \neg success)$	(A.1)
$fail \leftrightarrow \ \ rtn = -1 \ \wedge \ new\_ruid = old\_ruid \wedge$ $new\_euid = old\_euid \ \wedge \ new\_svuid = old\_svuid$	(A.2)
$fail \leftrightarrow \ \ einval \ \vee \ eperm$	(A.3)
$IsUID(old\_ruid) \ \wedge \ IsUID(old\_euid) \ \wedge \ IsUID(old\_svuid)$	(A.4)
$IsUID(new\_ruid) \ \wedge \ IsUID(new\_euid) \ \wedge \ IsUID(new\_svuid)$	(A.5)

Table A.3: Function definition and formulas for `setuid()`.

Annotated Standard Text / Logic Expressions

setuid(arg_uid)

DESCRIPTION

If the process has appropriate privileges, *setuid()* shall set the real user ID, effective user ID, and the saved set-user-ID of the calling process to *arg_uid* (A.7, A.9).

If the process does not have appropriate privileges, but *arg_uid* is equal to the real user ID or the saved set-user-ID, *setuid()* shall set the effective user ID to *arg_uid*; the real user ID and saved set-user-ID shall remain unchanged (A.8, A.10).

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error (A.1, A.2, A.6).

ERRORS

The *setuid()* function shall fail if:

[EINVAL] The value of the *arg_uid* argument is invalid and not supported by the implementation (A.7, A.8, A.3, A.12).

[EPERM] The process does not have appropriate privileges and *arg_uid* does not match the real user ID or the saved set-user-ID (A.3, A.7, A.8, A.13).

$$success \leftrightarrow \ (success\_ap \ \vee \ success\_nap) \tag{A.6}$$

$$AP \rightarrow \ success\_ap \ \vee \ fail \tag{A.7}$$

$$\neg AP \rightarrow \ success\_nap \ \vee \ fail \tag{A.8}$$

$$success\_ap \leftrightarrow \ rtn = 0 \ \wedge \ new\_ruid = arg\_uid \ \wedge \ new\_euid = arg\_uid \ \wedge \ new\_svuid = arg\_uid \tag{A.9}$$

$$success\_nap \leftrightarrow \ rtn = 0 \ \wedge \ new\_ruid = old\_ruid \ \wedge \ new\_euid = arg\_uid \ \wedge \ new\_svuid = old\_svuid \tag{A.10}$$

$$success\_nap \leftarrow \ \neg AP \ \wedge \ (old\_ruid = arg\_uid \ \vee \ old\_svuid = arg\_uid) \tag{A.11}$$

Continued on next page

Table A.3 – continued from previous page

Annotated Standard Text / Logic Expressions
$$einval \leftrightarrow \ \neg IsUID(arg\_uid) \quad \text{(A.12)}$$
$$eperm \leftrightarrow \ \neg AP \ \wedge \ arg\_uid \neq old\_ruid \ \wedge \ arg\_uid \neq old\_svuid \quad \text{(A.13)}$$

Table A.4: Function definition and formulas for `seteuid()`.

Annotated Standard Text / Logic Expressions
seteuid(arg_uid) **DESCRIPTION** If *arg_uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, seteuid() shall set the effective user ID of the calling process to *arg_uid*; the real user ID and saved set-user-ID shall remain unchanged (A.14, A.15). The seteuid() function shall not affect the supplementary group list in any way. **RETURN VALUE** Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and errno set to indicate the error (A.1, A.2, A.14). **ERRORS** The *seteuid()* function shall fail if: [EINVAL] The value of the *arg_uid* argument is invalid and not supported by the implementation (A.3, A.16). [EPERM] The process does not have appropriate privileges and *arg_uid* does not match the real user ID or the saved set-user-ID (A.3, A.17).
$$success \leftrightarrow \ rtn = 0 \ \wedge \ new\_ruid = old\_ruid \wedge \ new\_euid = arg\_uid \ \wedge \ new\_svuid = old\_svuid \quad \text{(A.14)}$$
$$success \leftarrow \ arg\_uid = old\_ruid \ \vee \ arg\_uid = old\_svuid \ \vee \ AP \quad \text{(A.15)}$$
$$einval \leftrightarrow \ \neg IsUID(arg\_uid) \quad \text{(A.16)}$$
Continued on next page

Table A.4 – continued from previous page

Annotated Standard Text / Logic Expressions
$eperm \leftrightarrow \neg AP \ \wedge \ arg\_uid \neq old\_ruid \ \wedge \ arg\_uid \neq old\_svuid$ (A.17)

Table A.5: Function definition and formulas for `setreuid()`.

Annotated Standard Text / Logic Expressions
setreuid(arg_ruid, arg_euid) **DESCRIPTION** The *setreuid()* function shall set the real and effective user IDs of the current process to the values specified by the *arg_ruid* and *arg_euid* arguments. If *arg_ruid* or *arg_euid* is -1, the corresponding effective or real user ID of the current process shall be left unchanged (A.19, A.20). A process with appropriate privileges can set either ID to any value (A.27). An unprivileged process can only set the effective user ID if the *arg_euid* argument is equal to either the real, effective, or saved user ID of the process (A.22, A.23). If the real user ID is being set (*arg_ruid* is not -1), or the effective user ID is being set to a value not equal to the real user ID, then the saved set-user-ID of the current process shall be set equal to the new effective user ID (A.21). It is unspecified whether a process without appropriate privileges is permitted to change the real user ID to match the current effective user ID or saved set-user-ID of the process. **RETURN VALUE** Upon successful completion, 0 shall be returned (A.18). Otherwise, -1 shall be returned and errno set to indicate the error (A.1, A.2).
Continued on next page

Table A.5 – continued from previous page

Annotated Standard Text / Logic Expressions

ERRORS

The *setreuid()* function shall fail if:

[EINVAL] The value of the *arg_ruid* or *arg_euid* argument is invalid or out-of-range (A.3, A.25).

[EPERM] The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation (A.3, A.24, A.26).

$$success \leftrightarrow \ rtn = 0 \ \wedge \ ruid\_success \ \wedge \ euid\_success \ \wedge \ svuid\_success \ \wedge \ arg\_euid\_success \tag{A.18}$$

$$ruid\_success \leftrightarrow \ (arg\_ruid = -1 \ \wedge \ new\_ruid = old\_ruid) \ \vee (arg\_ruid \neq -1 \ \wedge \ new\_ruid = arg\_ruid) \tag{A.19}$$

$$euid\_success \leftrightarrow \ (arg\_euid = -1 \ \wedge \ new\_euid = old\_euid) \ \vee \ (arg\_euid \neq -1 \ \wedge \ new\_euid = arg\_euid) \tag{A.20}$$

$$svuid\_success \leftrightarrow \ (new\_svuid = arg\_euid) \ \leftarrow \ (arg\_ruid \neq -1 \ \vee \ (arg\_euid \neq -1 \ \wedge \ arg\_euid \neq old\_ruid)) \tag{A.21}$$

$$arg\_euid\_success \leftrightarrow \ arg\_euid\_is\_valid \ \leftarrow \ (\neg AP \ \wedge \ arg\_euid \neq -1) \tag{A.22}$$

$$arg\_euid\_is\_valid \leftrightarrow \ arg\_euid = old\_ruid \ \vee \ arg\_euid = old\_euid \ \vee \ arg\_euid = old\_svuid \tag{A.23}$$

$$arg\_euid\_is\_invalid \leftrightarrow \ \neg(arg\_euid = -1 \ \vee \ arg\_euid = old\_ruid \ \vee \ arg\_euid = old\_svuid) \tag{A.24}$$

$$einval \leftrightarrow \ \neg(\ (arg\_ruid = -1 \ \vee \ IsUID(arg\_ruid)) \wedge (arg\_euid = -1 \ \vee \ IsUID(arg\_euid))) \tag{A.25}$$

Continued on next page

Table A.5 – continued from previous page

Annotated Standard Text / Logic Expressions
$eperm \leftrightarrow \neg AP \wedge (arg\_euid\_is\_invalid \vee \neg RuidIsPermitted(arg\_ruid))$ (A.26)
$AP \rightarrow success \vee \neg IsUID(arg\_ruid) \vee \neg IsUID(arg\_euid)$ (A.27)

Table A.6: Function definition and formulas for `setresuid()`. The function definition is the consensus standard, derived from three platform-specific `setresuid()` manual pages [10, 14, 17]. Ellipses indicate areas where platform-specific details appear in manual pages.

Annotated Standard Text / Logic Expressions
setresuid(arg_ruid, arg_euid, arg_svuid)
DESCRIPTION setresuid() sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process. Unprivileged user processes may change the real UID, effective UID, and saved set-user-ID, each to one of: the current real UID, the current effective UID or the current saved set- user-ID (A.33, A.34, A.35, A.36, A.37). Privileged processes [...] may set the real UID, effective UID, and saved set-user-ID to any value (A.32). If one of the arguments equals -1, the corresponding value is not changed (A.29, A.30, A.31). [...]
RETURN VALUE On success, zero is returned (A.28). On error, -1 is returned, and errno is set appropriately (A.38, A.39).
Continued on next page

Table A.6 – continued from previous page

Annotated Standard Text / Logic Expressions
ERRORS [...] [EINVAL] The value of the *arg_ruid*, *arg_euid*, or *arg_svuid* argument is invalid and not supported by the implementation (A.3, A.38). [EPERM] The calling process is not privileged and tried to change the IDs to values that are not permitted.

$$success \leftrightarrow \; rtn = 0 \; \wedge \; ruid\_success \; \wedge \; euid\_success \; \wedge \; svuid\_success \tag{A.28}$$

$$ruid\_success \leftrightarrow \; (arg\_ruid = -1 \; \wedge \; new\_ruid = old\_ruid) \; \vee \; (arg\_ruid \neq -1 \; \wedge \; new\_ruid = arg\_ruid) \tag{A.29}$$

$$euid\_success \leftrightarrow \; (arg\_euid = -1 \; \wedge \; new\_euid = old\_euid) \; \vee \; (arg\_euid \neq -1 \; \wedge \; new\_euid = arg\_euid) \tag{A.30}$$

$$svuid\_success \leftrightarrow \; (arg\_svuid = -1 \; \wedge \; new\_svuid = old\_svuid) \; \vee \; (arg\_svuid \neq -1 \; \wedge \; new\_svuid = arg\_svuid) \tag{A.31}$$

$$AP \rightarrow \; success \; \vee \; (arg\_ruid \neq -1 \; \wedge \; \neg IsUID(arg\_ruid)) \; \vee \; (arg\_euid \neq -1 \; \wedge \; \neg IsUID(arg\_euid)) \; \vee \; (arg\_svuid \neq -1 \; \wedge \; \neg IsUID(arg\_svuid)) \tag{A.32}$$

$$arg\_ruid\_is\_valid \leftrightarrow \; arg\_ruid = old\_ruid \; \vee \; arg\_ruid = old\_euid \; \vee \; arg\_ruid = old\_svuid \tag{A.33}$$

$$arg\_euid\_is\_valid \leftrightarrow \; arg\_euid = old\_ruid \; \vee \; arg\_euid = old\_euid \; \vee \; arg\_euid = old\_svuid \tag{A.34}$$

Continued on next page

Table A.6 – continued from previous page

Annotated Standard Text / Logic Expressions	
$arg\_svuid\_is\_valid \leftrightarrow \quad arg\_svuid = old\_ruid \vee$ $arg\_svuid = old\_euid \vee$ $arg\_svuid = old\_svuid$	(A.35)
$new\_uids\_are\_valid \;\leftrightarrow\; arg\_ruid\_is\_valid \wedge$ $arg\_euid\_is\_valid \;\wedge\; arg\_svuid\_is\_valid$	(A.36)
$success \leftarrow \quad \neg AP \;\wedge\; new\_uids\_are\_valid$	(A.37)
$einval \leftrightarrow \neg(\quad (arg\_ruid = -1 \;\vee\; IsUID(arg\_ruid)) \wedge$ $(arg\_euid = -1 \;\vee\; IsUID(arg\_euid)) \wedge$ $(arg\_svuid = -1 \;\vee\; IsUID(arg\_svuid)))$	(A.38)
$eperm \leftrightarrow \quad \neg AP \;\wedge\; \neg new\_uids\_are\_valid$	(A.39)

References

[1] Abu Nasser Mohammed Abdullah, Behzad Akbarpour, and Sofiène Tahar. Error Analysis and Verification of an IEEE 802.11 OFDM Modem using Theorem Proving. In *Proceedings of the First Workshop on Formal Methods for Wireless Systems (FMWS 2008)*, pages 3–30, 2009.

[2] Alexander Löhner. LiCo - The New LinuxCounter Project. http://linuxcounter.net/. Last accessed: Nov. 12, 2013.

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

[4] Matt Bishop. *Computer Security — Art and Science*. Addison-Wesley, 2003.

[5] dijkstra_shortest_paths. http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/dijkstra_shortest_paths.html. boost C++ libraries. Last accessed: Aug. 14, 2014.

[6] CERT. POS37-C. Ensure that privilege relinquishment is successful. https://www.securecoding.cert.org/confluence/display/seccode/POS37-C.+Ensure+that+privilege+relinquishment+is+successful, June 2013.

[7] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, August 2002.

[8] Drew Dean and Alan J. Hu. Fixing Races for Fun and Profit: How to Use access(2). In *Proceedings of the 13th USENIX Security Symposium*, pages 195–206, August 2004.

[9] Mark S. Dittmer and Mahesh V. Tripunitara. The UNIX Process Identity Crisis: A Standards-Driven Approach to Setuid. In *Proceedings of the 21st ACM conference on Computer and communications security*. ACM, 2014.

[10] getresgid, getresuid, setresgid, setresuid – get or set real, effective and saved user or group ID. FreeBSD System Calls Manual, April 2001.

[11] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[12] IEEE and The Open Group. POSIX.1-2008, 2013. Available from http://pubs.opengroup.org/onlinepubs/9699919799/.

[13] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. Practical Techniques to Obviate Setuid-to-root Binaries. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 8:1–8:14, New York, NY, USA, 2014. ACM.

[14] setresuid, setresgid - set real, effective and saved user or group ID. Linux Programmer's Manual, July 2007.

[15] Capabilities - overview of Linux capabilities. Linux Programmer's Manual, August 2009.

[16] Dennis M. Ritchie. Protection of data file contents. US Patent 4135240, January 1979.

[17] getresgid, getresuid, setresgid, setresuid - get or set real, effective and saved user or group ID. OpenBSD Programmer's Manual, August 2013.

[18] Mark S. Dittmer and Mahesh V. Tripunitara. unix-process-identity. https://github.com/mdittmer/unix-process-identity, 2014.

[19] A. Souari, S. Tahar, and A. Gawanmeh. Formal error analysis and verification of a frequency domain equalizer. In *IEEE 10th International New Circuits and Systems Conference (NEWCAS)*, pages 189–192, 2012.

[20] Dan Tsafrir, Dilma Da Silva, and David Wagner. Change Process Identity. Available from https://code.google.com/p/change-process-identity/. Last accessed May 2014.

[21] Dan Tsafrir, Dilma Da Silva, and David Wagner. The Murky Issue of Changing Process Identity: Revising "Setuid Demystified". *USENIX; login:*, 33(3), June 2008.