HACS: A Hierarchical Access Control System for User-Level Sandboxing*

Alex Fabrikant, Bryan Fulton, and David Schultz
{alexf,bfulton,dschultz}@cs.berkeley.edu

December 16, 2002

Abstract

We present the detailed design of HACS, a hierarchical control mechanism that extends the traditional user ID mechanism, present in UNIX and elsewhere, to permit users and applications to define fine-grained protection boundaries without the intervention of a system administrator. These facilities can express policies that adhere more closely to the principle of least privilege, and allow untrusted code to be executed more safely without complicated mechanisms layered on top of the operating system. We give the implementation details necessary to integrate such a mechanism into an existing OS, and applications of the scheme to the protection of specific major system resources such as the filesystem and the network.

1 Introduction

A popular access control model in modern operating systems consists of an inflexible two tier system. Specifically, each principal is roughly categorized as either a system administrator or as an ordinary user, and numerous access checks depend on the distinction. Though simple and elegant, this model has an inadequate level of granularity to express several important classes of security policies. For instance, a program that requires any privilege reserved for the administrator, such as raw network socket access, must be given all administrator privileges. If a program wishes to perform operations with a more restrictive set of access rights, it must use an often complicated mechanism to permanently or temporarily revoke all the administrative privileges. Worse, ordinary users cannot employ the operating system to define restrictive protection domains for agents acting on their behalf.

Much prior work has focused on partitioning the system administrator privilege in order to partially address the issue just described ([17], [12], [9]). Although these measures may help protect the integrity of computer systems, they do not prevent attacks against users' accounts. Such attacks, when successful, can lead to leaks or corruption of confidential information or to malicious use of system resources. Moreover, these attacks are becoming increasingly pervasive with the growing popularity of mobile code and complicated network clients that must interpret untrustworthy data.

We have designed and partially implemented HACS, a versatile mechanism by which ordinary users and the applications they run can create restrictive execution environments without the involvement of the system administrator. Under this system, users and applications may, at their discretion, create restricted execution environments in which to run untrustworthy code. The code may come from a trustworthy source but potentially contain exploitable security vulnerabilities, or it may come from an untrustworthy source; its origins might affect the level of trust it is granted. The underlying principle here, and the primary policy goal our system is designed to support, is that of *least privilege* ([14]), which states that an agent should have precisely the set of privileges necessary and no more. We support this notion through the abstraction of a *subuser*, which is effectively a new principal created by an existing principal

*Working title: Bryan-Alex-David Access Subdivision System

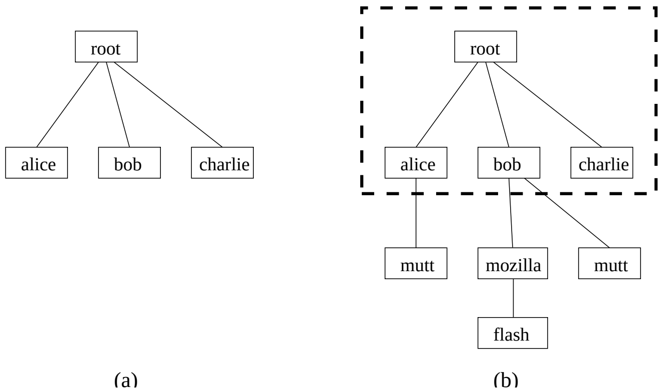

Figure 1: (a) The current UID model. (b) A possible user hierarchy under HACS; the subuser corresponding to the *flash* node is denoted by *bob.mozilla.flash*. Non-HACS-aware programs will not see anything outside the dashed boundary.

on the system. The subuser concept is simple to understand and straightforward to implement on top of current multiuser systems.

We introduce the principles underlying our system in section 2, discuss some significant details of its implementation in section 3, show how this scheme can be used to protect various system resources in section 4, and discuss differences between our system and other approaches to similar problems in section 5.

2 System Overview

In a HACS-enabled system, any user is given the ability to define *subusers* of himself. Each subuser, in turn, is able to define subusers of himself, and so on, creating a hierarchical structure. A simple sample configuration of such a system is shown in Figure 1(b). In an ideal implementation, i.e. one integrated into an operating system by design, this system would be completely homogeneous, allowing any policy statement that can be made with regard to *bob* to be made with regard to *bob.mozilla* as well. With that in mind, we use the terms *subuser* and *user* interchangeably to refer to any entity in the hierarchy. However, the use of both terms in the same context typically connotes that the subuser is inferior to the user. By convention, we use Latin letters for atomic segments of a subuser name (i.e. nodes in the above hierarchy such as *bob* or *mozilla*), and Greek letters for strings thereof, including full subuser names (such as *bob.mozilla* or *mozilla.flash*)[1].

The set of policies allowed under the HACS scheme is restricted by a single principle:

Principle 1 *A user may not be permitted any access that is denied to his ancestor in the user hierarchy. That is, for instance, user* x.y.z *cannot access any resource that any one of* x.y, x, *or* root *cannot access.*

To enforce this principle, we require that every subuser-aware access control check verifies, when deciding whether a subuser $x_1.\cdots.x_n$ has access to resource r, that $x_1.\cdots.x_i$ has access to r for all $i \leq n$. This immediately removes potential security problems such as the propagation of revokation of x's access to r to x's subusers.

The relationship between two subusers is governed by a second principle:

Principle 2 *A subuser α has full ("root-like") control over all of his subusers, $\alpha.x$, with respect to their naming and permissions. He does not have explicit access to modify the naming and permissions of any other subusers.*

Note that, in addition to the obvious restrictions, the above prohibits mechanisms giving subuser α

[1] This can also include the empty string; in particular, if α is an empty string, it by itself refers to *root*

the ability to explicitly control his "grandchildren", $\alpha.x.y$. This restriction simplifies the lookup procedure for policy decisions without unreasonably restricting the space of policies that may be specified, since if a user does not trust code running as a subuser of his to reasonably restrict the sub-subusers it creates, he can always further restrict the permissions of that subuser.

Finally, backward compatibility is a serious issue when integrating this scheme into an existing operating system. We attempt to address all important categories of resources in this paper, but we also note that protection for each resource can be implemented independently of the others. Meanwhile, the code for existing protection mechanisms for other resources will function without modification. This observation leads to a third constraint:

Principle 3 *All pre-existing code should function correctly and obey pre-existing security assumptions entailed by regarding any action by subuser $x.\alpha$ as being performed by user x.*

3 Implementation Details

We have partially implemented such a system as an extension to FreeBSD 5.0 [5], chosen primarily because it supports the UFS extensions necessary to implement the mechanisms in Section 4.1. Existing or potential resolutions to several interesting issues are presented below.

3.1 Data Structures

In the per-process credential structures and structures used to pass credential information to user programs, subuser IDs are encoded as 0-terminated integer sequences (so *root* is [0] while *bob* might be [102 0], and *bob.mozilla.flash* might be [102 1 2 0]).[2] Given Principle 2, this corresponds to the intuition that, in a sense, *bob.mozilla* may be thought of as *.bob.mozilla.root*, while *root* himself would be *.root*. Code similar to the following is used to replace the UID fields (here, the `cr_uid` field, in `struct ucred`):

```
int cr_uids[MAXSUBUID];
#define cr_uid cr_uids[0]
```

This simple scheme both allows for a minimal slowdown in policy check overhead (with each check being slowed down by effectively not much more than a factor of i for an i'th level subuser) and does not break references to `x.cr_uid` in existing code. This also gives each user a separate namespace to manage his subusers' IDs, and provides for easier adherence to Principle 3 compared to alternative schemes such as compacting subuser IDs into the current single-integer UID fields.

3.2 UID-Relevant System Calls

To minimize the confusion inherent with the current multitude of partially-standardized `setuid`-family system calls ([1]), we introduce a single system call, `setressubuid()`, with semantics parallel to the existing `setresuid()`. In particular, if a process with real, effective, and saved subuser IDs (ρ, ϵ, σ) calls `setressubuid()` with arguments $(\rho', \epsilon', \sigma')$, the IDs are changed if each of ρ', ϵ', and σ' is a prefix (not necessarily proper) of any one of ρ, ϵ, σ. Any subset of the arguments can be set to `NULL` to cause that field to remain unchanged, making it completely general and removing the need for other calls. The call is also constrained to be transactional — if it succeeds, all three are updated with the new values; if any one of the values may not be changed, the call fails without changing anything.

The existing system calls that deal with passing around UIDs, such as `fork()`, do not need to be changed, since they just copy the entire credential structure already. However, issues arise for some exotic corner cases of the existing `setuid` family calls. Suppose a program running as *bob.mozilla* executes a non-subuser-aware setuid root program, thus giving it an effective UID of *root* and real and saved UIDs of *bob.mozilla*, and suppose that the latter decides to revert the effective UID to *bob.mozilla* by calling `seteuid(getuid())`.[3] To retain compatibility with existing applications, the `getuid()` interface can only return a single integer, even though the canonical rep-

[2]The actual storage is allocated with a fixed-length array, with the maximum length currently set to 8 since we do not foresee any interesting applications for user hierarchies of depth greater than 3 or 4.

[3]For more information on the distinction among the real, effective, and saved user IDs in UNIX, the reader is referred to [1].

resentation of user IDs in our system is as sequences of integers.

One possible workaround would be to make `getuid()` return the first integer in the sequence. In the example above, the return value would be *bob*. However, this means that calls to `setuid()` can be ambiguous. Technically the process in the example above has appropriate privilege to set its UID to *bob*, but `setuid()` would need to understand that the correct meaning is *bob.mozilla*. Such magic is dangerous, and impossible to get right in general. For instance, what does `setuid()` do if the process's real UID is *bob.mozilla* and the saved UID is *bob.mutt*?

Schemes that rely upon keeping state in the kernel about `getuid()` calls are also destined for failure. Even if we resolve the ambiguity by mapping *bob.mozilla* onto a unique integer, we risk namespace conflicts, and we will not be able to track uses of the integer outside of the process for which it was created.

Due to these inherent limitations of the `getuid()` interface, we decided to take the safest approach and require that programs that use the old interface and are invoked by subusers be converted to the new interface. Presumably it will usually be unnecessary for subusers to invoke these programs in the first place, so the decision should result in little burden. This restriction can be enforced by either causing the system call to fail, or by killing the offending process when `setuid()` or a related call is made.[4] The latter approach is safer, since many programs, such as older versions of sendmail, neglect to check the return value of `setuid()` and expect the system to fully comply with POSIX semantics. However, we acknowledge that killing the process may be too draconian, so we give the administrator the choice of semantics as a kernel option.

It may also be appropriate to limit the execution of setuid binaries to narrow the interface through which a subuser might exploit a security hole; even binaries converted to the new interface could be buggy. A toggle can be provided on a per-subuser basis, and, once disabled, should eliminate most issues with `setuid`-family calls since they are rarely made outside code that was not run from a setuid executable.

[4]Note that even maximally stringent schemes for allowing *some* `setuid` calls to go through, such as only allowing them if all of the UID fields are root or a first-level user, can be problematic. If a process with euid *root*, ruid *bob.mozilla* runs `oldu=getuid()`, then for some reasons changes all its UIDs to *root* and then calls `seteuid(oldu)`, it will end up with higher permissions than it expects.

3.3 User-Space Support

Parallel to the existent system-wide `/etc/passwd` file, we allow each subuser, in his home directory, to have a similarly-formatted file named `.passwd`, which specifies name-to-UID mapping for his immediate subusers. Also in this file we must have a "home directory" for the subuser (primarily for locating his `.passwd`), as well as, optionally, any per-subuid settings needed to control any specific resource-protection mechanisms present in a particular implementation.[5]

The default settings for a user such as resource limits and environment variables specified in `login.conf` are set by the program handling the login, such as `ssh` or `login`. Similarly, a userland program called `subexec`, executed as user α, "logs in" a subuser $\alpha.x$ by parsing α's `.passwd` file, setting the effective, real, and saved subuser ids appropriately, applying any subuser-specific restrictions to the process[6], and executing a command of α's choice in the newly-restricted environment.

Additional wrappers can easily be created to simplify the process of automating the execution of a particular application under as a particular subuser, and to allow for safe editing of `.passwd` files.

4 Protectable Resources

Once a HACS scheme is in place on a system, it can be used to protect any subset of the system's resource types. Principle 3 guarantees that the existence of such protections is orthogonal between different resources. We present the mechanisms needed to add HACS-aware protection to most major resources under FreeBSD 5.0.

[5]The `.passwd` files need not be world readable. If they cannot be read by others, the users they contain will be seen as numerical user ids.

[6]Neither of which actions, note, require `subexec` to be executed by anything more privileged than α. For this reason, we do not simply overload the semantics of the `su` utility, which ordinarily runs with elevated privileges.

```
user::rwx
user:alice:r-x
user:charlie:r-x
group::---
mask::r-x
other::---
```

Figure 2: A typical ACL following the POSIX draft.

4.1 Filesystem

The filesystem is arguably the most central and the most complicated resource that must be protected. It is important not only to deny access to unauthorized principals, but also to provide expressive mechanisms to facilitate sharing where appropriate. Moreover, we must consider the effects of our filesystem changes not only on systems that implement the new model, but also on kernels that do not support the features.

4.1.1 Semantics

Although the task of extending the filesystem to support subuids sounds relatively straightforward, several tricky issues must be dealt with in order to give users enough expressive power to define reasonable security policies. Moreover, it is convenient to have a prototype that does not break filesystem compatibility with existing kernels. The standard three-entry access control list traditionally used by UNIX systems is insufficient to even permit a user to delegate access rights to his subusers.

We assume a UFS filesystem with an implementation of fine-grained access control lists as specified in draft 17 of the withdrawn POSIX 1003.1e standard [12], as well as IRIX-style extended attributes as implemented in FreeBSD. The POSIX draft describes a mechanism that allows each user to give access permissions to specific other users. An access control list associated with a file owned by *bob* might look like Figure 2. That ACL specifies that *bob*, the owner of the file, has read, write, and execute permissions on it. The users *alice* and *charlie* have read and execute access but not write access, and members of *bob*'s group and the rest of the world have no access. The mask entry is a convenience for specifying ACLs; it is bitwise-ANDed with each ACL entry except the one for the file owner.

This mechanism can easily be extended to include basic support for subusers. In addition to an entry for *alice*, we could have one for *alice.mozilla*. Alice might have such an entry when delegating a subset of her privileges to the subuid she intends to use for running an untrusted program. But what happens when *alice* gives *bob* access to, say, an HTML document, and *bob* wants to view it in his web browser? He needs to delegate his access to the file to *bob.mozilla*; however, traditionally only the owner of a file (or a user with a superset of the owner's privileges) may specify access permissions.

To address this problem, our system should, in general, allow users to delegate access to subusers for all files for which they have appropriate access rights, even if their access is read-only. We employ the following specific rules:

- The owner of a file may add ACL entries referring to arbitrary subusers in the system.
- Any subuser with non-zero access to a file may add ACL entries referring to his children.

The first rule is a consequence of our intent to not make arbitrary distinctions between first-level users and subusers lower in the hierarchy. We do not want to restrict x from making policy decisions about $y.\alpha$, even though we do not believe this is a common thing to do. The second rule allows *bob* to delegate access to *bob.mozilla* as he would like to do, without giving him the unchecked ability to delegate access to *alice*'s files to other users. Lampson has suggested that the owner of a file should be allowed to define who has the capability to make the latter class of delegations ([8]).

We believe that it is also useful to include wildcard support in ACLs. The original POSIX 1003.1e ACLs need only have the notion of "all users", which is encoded by leaving the second field of an ACL entry blank. However, our model should be able to express classes such as "all direct children of *bob*". We allow an asterisk to be used for this purpose, as the last component of a subuser sequence, so the previous example would be encoded as *bob.∗*. We do not provide a way to refer to "all grandchildren of *bob*" as a matter of simplicity of mechanism. Such a feature could be added at a later time, but, pursuant to Principle 2 we believe that in a well-abstracted policy, *bob*'s children should be responsible for specifying the access permissions of *bob*'s grandchildren.

The semantics of the traditional UNIX permission bits pose a small problem. Ordinarily, many standard system files are marked world readable, or, in the case of the system temporary directory, world writable. According to [12], setting the permission bits overrides any specification in a fine-grained ACL. In some situations it may be desirable to restrict a subuser's access even to files that are customarily designated as world-readable. For instance, a system might have a database that should be accessible by all local users, but off-limits to an untrusted program running under a subuser id. For files in this category, we suggest that the owner add an ACL entry giving access to `*`. As explained above, this specification gives permission to top-level users, but allows nested subusers access only by explicit delegation.

4.1.2 Disk Structures

To support subusers, we must modify the filesystem to store sequences of subuids wherever it presently uses single user ids. Several organizations are possible. For example, we could store a bijective mapping from a flat user ID namespace to a hierarchical one or create a new filesystem with native support for the desired features. However, one of the design goals for our prototype implementation is that it be compatible with existing kernels. A production implementation might sacrifice this convenience for efficiency.

All principals in the system, including subusers, may create files in any directories where they have write access. In most cases, the default owner of a newly-created file is the principal who created it, so subusers should be capable of owning files. To maintain compatibility with the existing filesystem format, we store all extra metadata in *extended attributes*. Extended attributes are extensible repositories for arbitrary metadata associated with files, and fine-grained access control lists are typically implemented on top of them.[7] The traditional top-level owner of a file is still stored in the inode for compatibility, and any additional subusers are stored in extended attributes. Since dereferencing an extended attribute block is is extremely expensive, we store a flag in the inode indicating whether a file has such an extended attribute. In doing so, we avoid the inefficiency in a common case. The cached in-memory format of the inode contains all of the subuser ids, which makes successive references to the file less expensive.

We also need to be able to add ACL entries for subusers. ACLs are stored in extended attributes in FreeBSD 5.0. Rather than using a new format for all ACLs, we represent ACL entries for top-level users in the existing format, and use a separate extended attribute for other ACL entries. On kernels that do not support the new features, the ACL entries for nested subusers are automatically ignored.

One unfortunate limitation of the extended attributes implementation we are using is that the total space usable for per-file metadata is bounded.[8] Consequently, we need to take care that a user cannot monopolize this space on a file he does not own. Fortunately, quotas on the order of a hundred bytes are usually sufficient to allow users enough expressive power. A per-file metadata quota applied to any user must also apply to all of that user's descendents in order to prevent abuse through the creation of numerous subusers. Quotas for top-level users should be large enough to permit an adequate degree of expressive power, while still reserving sufficient metadata space for others.

In addition to the aforementioned modifications, a series of straightforward extensions must be made to the virtual filesystem interface in order to communicate subuser ids between the filesystem and the kernel proper. In particular, several additional HACS-aware ACL-related system calls must be created, which would enforce the above policies.

4.2 Network

Another resource that should be protected from unauthorized access is the network. Many fine-

[7] Since these attributes do not have a fixed size, as does the inode, we do not have to worry about hard-coding any restrictions on subuid depth into the filesystem.

[8] The limit happens to be two filesystem blocks, which is 32 kilobytes on a typical filesystem today. In an alternative organization, which was used by earlier extended attributes implementations for FreeBSD and IRIX, a full-fledged file backs the metadata, allowing for potentially unbounded capacity. But regardless of the scheme chosen, disk space is still a finite resource and must be managed carefully.

grained restriction mechanisms are possible. For example, per-subuser firewall rules could be used to restrict communications to certain network ports, or to enforce bandwidth limitations. Facilities for matching packets based on user ID and applying said restrictions already exist, and need only be extended to understand subuser IDs ([4]) Presumably an incoming packet destined for user $x_1. \cdots .x_n$ would need to be accepted by the system firewall rules designated by the administrator, then by all rules specified for $x_1. \cdots .x_{i+1}$ by $x_1. \cdots .x_i$, in order[9]. The algorithm is identical for outgoing packets. However, it is nontrivial to allow ordinary users to enter firewall rules into the kernel without making denial of service attacks trivially possible. Even if we only permit BPF-style rules such that infinite recursion in the ruleset is not possible, the rules for each subuser would need to persist in the kernel as long as they might be needed, and the extent of that state is difficult to define.

The only currently plausible solution we are aware of that allows per-subuser rules as described would be to force all firewall activity pertinent to rules defined by non-*root* users to be carried out by user-space programs. However, the cost of handling network traffic in userland is likely to outweigh the potential benefits of such a scheme, at least for busy servers. On small systems with lower network load, paying the extra context switches and extra protection boundary crossings for each packet may be acceptable.

Alternatively, if a pseudo-filesystem such as AccessFS ([3]) that allows processes to control their network permissions via filesystem calls is extended to include ACL and extended attributes structures exploited by the filesystem access control mechanism above, the latter can be homogeneously extended to provide network access control as well. Furthermore, connection-oriented protocols would be more efficient than in the firewall-based approach because access checks would be done at the time of a connection attempt, rather than for each network packet. However, some policies (e.g. bandwidth limitation) are difficult to express under this framework.

Since it is not clear which model is the most useful in practice, we advocate a simpler approach, leaving a more expressive one as a future extension. We note that many programs require no network access at all, so a simple binary model will often suffice. Therefore, we add a per-process flag specifying whether network and inter-process communication is possible for the process and any of its descendents. By associating this bit with processes, we avoid the need to manage per-subuser state in the kernel. As with resource limits, userland management tools can optionally drop network access when they descend into a subuser.

[9]It is also technically possible for a user α to impose firewall rules upon himself, perhaps as an explicit way of blocking unwanted traffic.

4.3 Interaction with the X Window System

Several times in this paper we have mentioned web browsers and other applications that typically require a graphical user interface. The most popular GUI programming interfaces for UNIX applications employ the X Window System protocol. In this system, applications are clients that connect to a server, often on the same machine. The server displays windows as specified by the clients and notifies the clients about relevant user input events. This model is problematic if we wish to sandbox an application with a GUI, because the server trusts all clients equally.

Fortunately, this problem has already been solved. It is possible to indirect communication between the client and server through a proxy to enforce restrictions on clients ([10], [7]). The proxies we are aware of enforce geometrical restrictions, which would allow users to confine each subuser to a subset of the screen. We have not yet integrated this functionality into our implementation, but we intend to do so.

4.4 Groups

In keeping with Principle 1, no subuser can be allowed to belong to any group that all of his ancestors do not belong to. While the current implementation does not allow a user to change the GID or groups membership of his subusers, a simple further refinement can allow this. In that case, a user would be allowed to, for each subuser, specify in his `.passwd` (or a `.groups` file in the same directory) a set of groups to which the latter belongs. Then, the `subexec` program, when switching to the environment of a subuser, would delegate to the new process only those groups that are both allowed for that subuser and are already in the credentials of the calling process. Note that this would require an expansion to the current `setgroups()` system call which in the current implementation does not allow non-*root* users to do anything whatsoever. By allowing `setgroups()`, when called by a non-*root* user, to set the groups to a subset of the current groups, we allow the above operation to be performed by any user without introducing any means of breaching existing security policies.[10]

This technique can even be extended to allow a user to create a process with no associated groups. That feature would require the creation of a special GID value (say, -1) corresponding to the complete absence of group membership, and teaching the kernel about the special case.

4.5 Miscellaneous Resources

In addition to the above, access to a number of other resources can be modified to allow per-subuser restrictions by relatively minor modifications to the kernel and/or the `subexec` wrapper. We list a representative sample below, but mechanisms to account for other resources may follow similar patterns as well:

Signal passing: Without modifying existing signal-handling system calls to support HACS, any process owned by $x.\gamma$ will be able to send signals to any process owned by[11] $x.\delta$. The `kill` system call (in particular, the `cr_cansignal` access control checker) can be easily fixed to only allow signals to be sent by α's processes to $\alpha.\beta$'s processes.[12] A separate field in the `.passwd` may also be introduced to allow a particular subuser $\alpha.x$ to send signals to processes owned by α (and, consequently, $\alpha.\beta$); such a field may also allow this to be enabled on a per-signal basis, so that a user may, for instance, only allow a certain subuser to send SIGUSR1 and SIGUSR2 signals to his processes to establish inter-process communications. To implement this restriction, we would need to expand the process credential structure to include a signal-subuser-ID (and a per-signal mask if desired), set by the `subexec` wrapper, potentially based on the `.passwd`, to the prefix α such that the new process and its children can only send signals to processes owned by subusers of form $\alpha.\beta$. The `kill` system call would then need to check process credentials against that. The same access controls should apply to process tracing and debugging facilities as well.

Resource limits: As with the existing `login.conf` mechanism, per-process resource limits such as maximum memory usage and maximum CPU time can be set based on per-subuser configuration in `.passwd` or elsewhere by the `subexec` wrapper. The `setrlimit` mechanism already prohibits a process from setting limits higher than its own, so Principle 1 is enforced automatically. Per-UID resource limits (currently the process count and socket buffer space consumed) must be handled differently. A `uidinfo` structure, currently created once for each user, must be created for each subuser, and any per-user resource charged to user α must also be charged to all of his ancestors (to prevent resource abuse via creation of multiple subusers). This requires just a backlink-to-superuser field in `uidinfo`, and a simple modification to the accounting code that will slow it down by a factor of not much larger than the depth of the current subuser. As a special case, we exempt the superuser from being charged for every user's resource usage in order to preserve existing semantics.

Setuid program execution: To address problems due to interaction with `setuid`-family calls, we can, as mentioned above, allow a user to disallow his subusers from running setuid programs. This would just require an extra bit in a process's credential structure, filled out by `subexec` based on a new field in `.passwd`, and checked by `exec`.

[10] For this to be sound with respect to current security policies, we must assume that groups are used as capabilities, and not as mechanisms for defining *ex*clusion sets. We believe this is a reasonable assumption given current practices.

[11] There are non-trivial restrictions on which of the three UIDs correspond to the two occurrences of "owned by" in this relationship, but the resulting distinctions are orthogonal to the discussion here.

[12] As with the existing scheme, a process should, in all circumstances, still be allowed to send SIGCONT any other process in the same session, independent of credentials; this is harmless by design.

5 Other Approaches

There are several well-studied methods for sandboxing untrusted programs, including virtual machines, safe languages and code verification, system call reference monitors, and information labeling, each of which has certain drawbacks that are addressed by HACS.

One of the most popular approaches to access control is the use of a virtual machine abstraction. Examples such as Java ([2]) and VMWare ([15]) are problematic for a variety of reasons. While Java is a comprehensive system that addresses safety and security, it does not provide security for legacy software, as our model does. Both Java and VMWare carry a performance penalty that makes them unsuitable for many applications. In particular, applying our nested access control model to virtual machines is likely to result in abysmal performance. Both effectively build an operating system on top of another operating system in order to address various deficiencies in the underlying system. We argue that at least the problems relating to security can be fixed without adding another layer of indirection and the associated complexity.

Another model is the use of reference monitors such as Janus ([16]) and Systrace ([13]), which implement security policies at the system call level. While these systems provide practical and simple methods of application sandboxing, canonicalization issues arise because security policies are not tightly bound to the objects they are designed to protect. We believe that HACS permits policy specifications to be more expressive. Moreover, it was designed with a hierarchical structure in mind from the beginning. With Systrace, on the other hand, it is difficult to define a policy that allows one instance of the monitor to be embedded within another, since the monitor requires privileges that could potentially be used to escape from the protected environment.

Language-based security is another approach to the problems of access control and system security. Examples of this model are Java and Proof-Carrying Code ([11]). PCC is a technique by which the code consumer can verify that code provided by an untrusted code producer satisfies a predefined set of safety properties. These rules are chosen by the code consumer in such a way that they are sufficient guarantees for safe behavior of programs. Yet this model is also not an option for legacy software. Furthermore, it appears likely that many non-trivial formal resource usage bounds proofs will be severely intractable, incurring an excessive performance overhead. HACS is subject to neither of these restrictions.

Perhaps the approach most similar to ours is SubOS, with which we share the term "subuser", albeit used here in a different context. SubOS is a non-hierarchical system based around the concept of information labeling. Data are labeled with immutable tags that bind sets of access restrictions to them, and processes inspecting those data are subject to the restrictions.[13] The system allows users to control their processes in the spirit of Orange Book-style mandatory access control, as opposed to our approach, which can be viewed as an extension to existing discretionary access control interfaces. The SubOS design as presented in [6] breaks down when multiple programs access the same data in different contexts. Also, although the designers did not specify how access rights are bound to subuids, it appears that their approach is similar to [13] and [16] and thus inherits the canonicalization problems of those approaches.

6 Conclusion

We detailed the design of HACS, a versatile and easy-to-understand mechanism to enable unprivileged users and applications to define protection domains. We have also implemented the kernel framework to support the system in the FreeBSD 5.0 kernel; however, we note that the main ideas of the design are applicable to many multi-user operating systems. Our model addresses a problem that has not been solved in most widely-deployed systems — namely, the lack of a simple OS-based mechanism for confining untrusted code. Using HACS, users can safely run mobile code and guard their accounts against attacks that take advantage of buggy applications. Finally, we have shown that the HACS framework allows for sufficient reverse-compatibility with the traditional UNIX protection model that it should

[13] SubOS addresses only the filesystem; however, other information labeling approaches such as [17] also cover data paths such as IPC.

be relatively simple to deploy in existing systems.

Future work in this area should include, naturally, a full implementation of such a mechanism with all relevant resources protected, as well as measurements of its impact on system performance under various conditions. Also, there are a number of interesting questions surrounding the obstacles preventing efficient HACS-based protections around certain resources. For instance, this includes possible approaches to allowing user-definable firewall rules without neither the substantial performance drops associated with processing network traffic outside the kernel nor the potential for denial-of-service attacks by malicious users on the kernel entailed by allowing users to add arbitrarily complex rules to a kernel firewall. Beyond that, it naturally remains open whether a system like this is more applicable in practical settings than the current alternatives such as SubOS, or whether better approaches to user-level sandboxing can be found.

References

[1] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *Proceedings of the Eleventh Usenix Security Symposium*, San Francisco, CA, 2002.

[2] D. Dean, E. W. Felten, and D. S. Wallach. From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[3] O. Dietsche. AccessFS: Permission Filesystem for Linux. http://home.t-online.de/-home/olaf.dietsche/linux/accessfs/, 2002.

[4] Firewalls. In FreeBSD Handbook, sec. 10.7, 2002.

[5] FreeBSD. http://www.freebsd.org/.

[6] S. Ioannidis and S. Bellovin. Sub-Operating Systems: A New Approach to Application Security. In *Proceedings of the SIGOPS European Workshop*, September 2002.

[7] B. L. Kahn. Safe Use of X Window System Protocol Across a Firewall. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, June 1995.

[8] B. Lampson. Protection. *ACM Operating Systems Rev.*, 8(1):18–24, January 1974.

[9] Linux Capabilities FAQ 0.2. Available at ftp://ftp.de.kernel.org/pub/linux/libs/-security/linux-privs/kernel-2.2/capfaq-0.2.txt.

[10] D. Matic. XNest. Available in the X11R6 source, and via public ftp from ftp.cs.umass.edu.

[11] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[12] POSIX 1003.1e Draft Standard 17 (withdrawn). POSIX, 1997.

[13] N. Provos. Improving Host Security with System Call Policies. Technical Report 02-3, Center for Information Technology Integration, November 2002.

[14] J. Saltzer and M. Schröder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[15] VMWare. http://www.vmware.com/.

[16] D. A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, UC Berkeley, CS Division, 12, 1999.

[17] R. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28, June 2001.