SUB-IDENTITIES:
A HIERARCHICAL IDENTITY MODEL FOR PRACTICAL CONTAINMENT

A Thesis

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science and Engineering

by

Philip Snowberger, B. A., B. S.

Douglas Thain, Director

Graduate Program in Computer Science and Engineering

Notre Dame, Indiana

April 2007

SUB-IDENTITIES:
A HIERARCHICAL IDENTITY MODEL FOR PRACTICAL CONTAINMENT

Abstract

by

Philip Snowberger

The operation of modern security systems demands too much sophistication from the average user. Further, in the face of increasingly common malware and spyware, users are not empowered to protect themselves. To address these issues, I propose sub-identities, a simple abstract identity model for practical containment in the operating system. In this model, user identities form a hierarchy, and each user may create sub-identities at any time without the help or approval of an administrator. While this model does not provide the fine-grained security available with more intrusive or comprehensive systems, it provides a significant measure of drop-in security that is more accessible to the average user. In this work, I demonstrate an implementation of the abstract model of sub-identity in the form of a user-space toolkit, Pluggable Authentication Module, and user-space filesystem, as well as several applications of sub-identity and a continuum of disciplines for using sub-identity.

CONTENTS

TABLES

FIGURES

ACKNOWLEDGMENTS

I would like to thank my wonderful fiancée, Karen Chan, for providing support, editing suggestions, and insights; my advisor, Dr. Douglas Thain, for his guidance and intuitions; and all my friends and family for their constant support and understanding.

In addition, I would like to thank the National Science Foundation for their generous grant, number CNS 05-49087, which made this work possible.

CHAPTER 1

INTRODUCTION

Users of modern computer systems find themselves awash in a sea of malware, macro viruses, Trojan horses, and spyware. Faced with the choice between a rich experience through downloading and running new programs and a dull experience through software teetotalism, many users elect the former and suffer the consequences, perhaps unknowingly. Security breaches occur in many ways; for instance, poorly-written software can inadvertently or ineptly clobber private files or expose them to public view, and malicious software can vandalize home directories or send sensitive documents to remote hosts.

> *You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you using untrusted code.* - Ken Thompson [39]

Most users lack the expertise or the time necessary to inspect every line of code of every piece of software that they run. Consequently, users cannot be assured of the security of their data unless they take measures to compartmentalize or contain potentially rogue code.

Any security measure must be simple and intuitive, or it will be seldom used or used incorrectly [15, 42]. Containment mechanisms are no different; if a user configures a containment measure poorly because he doesn't know how to use it properly, it should be no surprise that his private data could suddenly become public knowledge. Comprehensive security systems that provide very powerful access control models are in no short supply [13, 20, 25, 26, 34]. However, while quite powerful and a formidable defense when properly deployed, these systems require a great deal of technical expertise to operate correctly. Few users and systems need security that would satisfy the United States Department of Defense — many can achieve an acceptable level of security with a simpler mechanism. Further, presenting an unnecessarily complicated interface to the user, programmer, or administrator is a good way to have it be abused or ignored.

Users are most likely to embrace and effectively use a security system which presents an intuitive interface. Most users have an intuitive understanding of the family hierarchy, inasmuch as they have a sense that parents have absolute control over their children. For instance, parents have the authority and the power to "snoop" in their children's rooms, but the reverse is not true. Because users can identify with this concept, a containment system based on a hierarchy is thus intuitive. There already exist containment techniques that give some impression of modeling a hierarchy, such as traditional sandboxing [19] and virtual machines [12, 40], however these approaches require a high level of user sophistication to operate correctly.

Clearly, something more accessible to the average user is necessary. Users are already familiar with the concept of accounts, if only because they are required to log in with a user name and password, both on their home system and on myriad online services. With this familiarity comes the realization that users cannot damage or read other users' private information. How then could a user protect his files from a potentially malicious piece of code? He could do so by creating a new identity in which to run the code, relying on the operating system's standard protections that keep users from compromising each other. However, administrators of conventional operating systems are presented with a conundrum: because the operation of assuming another identity is a privileged operation, a user cannot protect himself from an untrusted program unless he becomes the super-user first, which introduces its own set of risks and complications. Note especially that, in order to *restrict* one's own privilege, one must first be *elevated* to maximum privilege, exposing the entire system to risk. This offends the principle of least privilege [31].

To remedy this situation, I propose an abstract model of *sub-identity*, and provide an implementation of the model that can be "dropped in" to existing Unix-style operating systems. In an operating system that implements the abstract model of sub-identity, every user can create new protection domains, each of which can have a meaningful name and can be used to enforce access control, perform auditing, or simply isolate sub-processes from one another.

This work is organized as follows: Chapter 2 presents the abstract model of sub-identities as it would be implemented in a hypothetical operating system. Chapter 3 outlines the place of the abstract model of sub-identity in the context of related work. Chapter 4 explores how sub-identities can be approximated in existing operating systems with a drop-in identity toolkit. Chapter 5 gives some applications of the sub-identity model including creating a safe X window, untrusted web browsing, untrusted web serving, and securing remote execution. Chapter 6 illustrates a continuum of disciplines for using sub-identity, and how various points along the continuum vary in security and convenience. Chapter 7 draws some final conclusions.

CHAPTER 2

ABSTRACT MODEL OF SUB-IDENTITY

In this chapter I describe an abstract model of sub-identities, assuming that all aspects of the operating system are open to modification. It should be noted that the operations presented here for interacting with the model need not be user-facing; they could also be wrapped in an intuitive graphical metaphor, or used solely "behind the scenes" by applications to protect the good name of the application by preventing its subversion. An implementation adapted to an existing system might deviate from this model in certain ways; Chapter 4 discusses one such implementation.

In traditional systems, the identity space is flat, or else the identities have a fixed form to them. For such systems, this model is sufficient, because identities are only created by an administrator and there is usually no need to create a hierarchy of identities. A hierarchical model of identity can give users the power to manage their identity domain in much the same way that they have traditionally had power over their home directories, a chunk of secondary storage.

Figure 2.1 shows how sub-identities might be used in a Unix-like system. Each edge indicates the creation of a user, running from the *superior* user to the *inferior* user. In this example, the root of the identity hierarchy is the *root* user, which is superior to its three inferior users *alice*, *www*, and *kerberos*. The fully-qualified

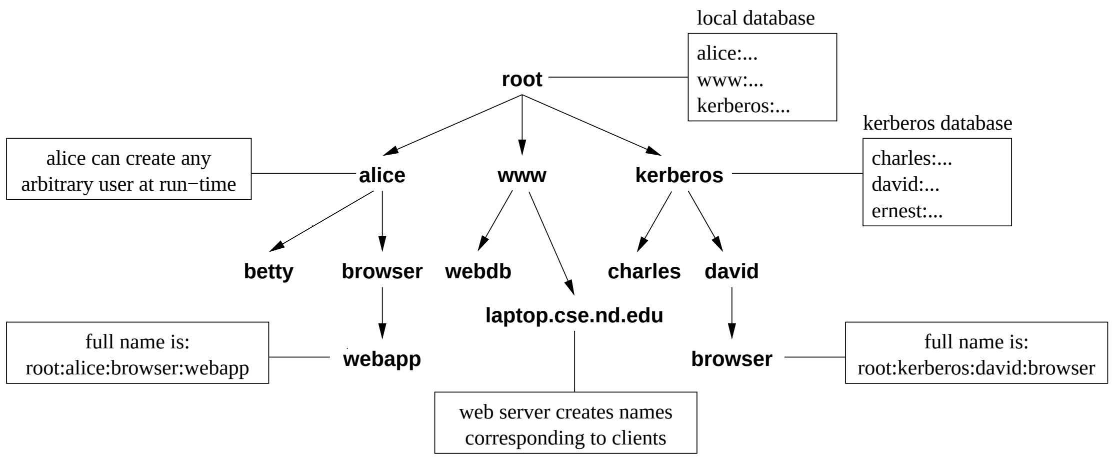

local database
alice:...
www:...
kerberos:...
root
alice can create any arbitrary user at run−time
alice
www
kerberos
kerberos database
charles:...
david:...
ernest:...
betty
browser
webdb
charles
david
laptop.cse.nd.edu
full name is: root:alice:browser:webapp
webapp
browser
full name is: root:kerberos:david:browser
web server creates names corresponding to clients

Figure 2.1: Example of an Identity Hierarchy. This figure shows a variety of users that might be employed on a system with sub-identities. The *root* user starts services and accepts ordinary logins, consulting a local user database before granting access. *root:alice* creates a variety of identities for her personal use. *root:www* is used to run the web server, and safely operates each subsystem in a sub-identity. *root:kerberos* also accepts logins and creates new sub-identities corresponding to users that appear in the remote Kerberos database.

name of *betty* in the hierarchy depicted in Figure 2.1 is *root:alice:betty.* This fully-qualified name also permits distinction between identities in different branches of the tree: *root:alice:browser* is distinct from *root:kerberos:david:browser.*

2.1 Operations on identities

The programmatic interface to sub-identities is simple. A process may obtain its identity by calling *getuser().* A process running as the identity x may call *subuser(y)* in order to change the identity of the current process to *x:y.*

It is important to note that the programmatic interface does not dictate how superior users make authentication decisions. *subuser(x)* is roughly analogous to the *setuid* system call in Unix; it simply modifies the identity of the current process. Each level of the hierarchy can have its own authentication method. For example, the *root* user can employ the traditional user database in */etc/passwd* in order to validate passwords and admit new users. This database could be made to only reflect the users in the second level of the tree, instead of being a global database of users. The local Kerberos service need not consult */etc/passwd*; it relies entirely on the remote Kerberos service to decide what users to admit. Other ordinary users may implement their own authentication schemes and simply invoke *subuser(n)* as they see fit. Every inferior process retains the ability to perform *subuser(n)*, but this is safe because *subuser(n)* does not allow any process to elevate its privilege. Note that in any implementation, some kind of limit will need to be placed on the depth and width of the identity hierarchy so that runaway processes don't perform a denial of service on the system.

2.2 Semantics of the ideal model

In the ideal identity model, a superior user is "effectively *root*" with respect to its inferior users; a superior user may send signals to its inferiors, debug their processes, modify their data, and perform any other activity necessary to ensure the safety and correctness of their operation. Naturally, an inferior user has no such power with respect to its superiors. A consequence of this is that there is very little reason for any user to assume the *root* identity, except perhaps to modify the operating system kernel or install a device driver. Further, this follows the principle of least privilege, since users can give up privileges without first gaining more.

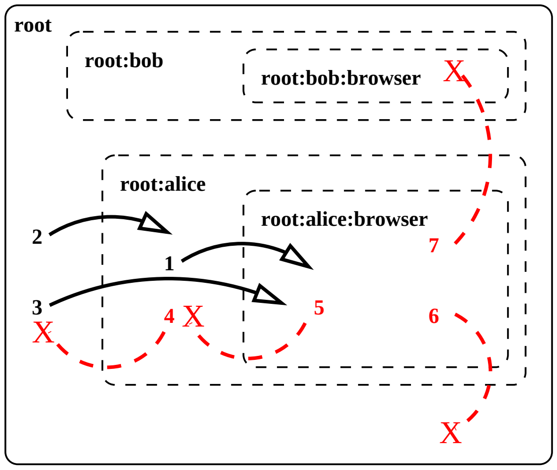


Figure 2.2: Permissions Granted by the Abstract Model of Sub-Identity. This figure shows the default permissions that the abstract model of Sub-Identity grants. Users have unrestricted access to files belonging to their sub-identities. In this figure, callouts **1**–**3** show the access that is allowed by the abstract model. **4**–**7** show access patterns that are not affected by the abstract model.

The quality of an identity being "effectively *root*" to its sub-identities can be defined as:

> An action consisting of a *subject*, *verb*, and an *object* shall succeed if the owner of *object* is a descendant of *subject* in the identity hierarchy.

The *verb* above corresponds to a system call in a traditional operating system. *subject* corresponds to a principal — usually simply called a "user". An *object* can refer to a variety of types, depending on the verb. For instance, the verb *chown* expects a file identifier as its *object*, whereas the verb *setuid* expects a principal or user identifier.

Note that the above condition is *sufficient* but not *necessary* for an action to succeed; it allows for actions to fail in cases where the object's owner is not a descendant of the subject. For instance, since *root:alice* is not an ancestor of *root:bob* and thus not effectively *root* to *root:bob*, the sub-identity hierarchy would not grant *root:alice* the ability to read a file owned by *root:bob*. Note, however, that some other access control mechanism could grant such access.

The default permissions for accessing objects are thus largely the same as in traditional Unix: users have access to all files that they own, as well as all the files on the system that are marked world-readable. In addition to the traditional Unix permissions, the abstract model gives users full access to files belonging to their sub-identities. These additional permissions are illustrated in figure 2.2.

2.3 Uses of sub-identity

A few examples following the Figure 2.2 should serve to illustrate uses of sub-identities.

Suppose that Alice wants to log into the console on a system that uses the standard Unix authentication mechanism for the top level of the identity hierarchy. After consulting */etc/passwd*, *root* creates a new identity *alice* (if, for example, this is her first time logging in) to run her programs. Alice then proceeds to work as normal. If she wishes to run any program that she does not fully trust, she may create a new sub-identity for that program. For example, if she has a visitor in the office, Betty, who wishes to use her computer, she may simply create a new user *betty*. This new identity protects Alice from any mishap by Betty, but it also gives Betty a clean workspace and the ability to store data under her own name and return to it later if needed.

As another example, if Alice browses the web in a traditional operating system, she runs the risk of being attacked by malicious software. To defend herself on a system which implements sub-identity, she may create a new user *browser* simply for the purpose of running a web browser. If the browser itself should be compromised, it will not be able to directly attack any programs owned by Alice or superior users. To go even further, the web browser *itself* could create a sub-identity *webapp* in order to protect itself from any helper applications it invokes. The ability to create sub-identities allows for a multi-layered defense.

A web server can also make good use of sub-identities. Many powerful web services are implemented by running sub-programs from the web server. These programs often contain security vulnerabilities [29]. A web server may defend itself by running each sub-process with a sub-identity. Each sub-identity may employ a meaningful name that allows it to access selected portions of the filesystem. For example, a database administrator might deploy data and make it accessible only by the *root:www:webdb* user, thus isolating files only pertinent to databases from other web services. Or, the web server might choose sub-identities based the name of the host issuing the HTTP request, such as *root:www:laptop.cse.nd.edu*. Content developers could then use standard filesystem tools in order to control access to remote users, or to present special content to previously-registered users, by having separate *htdocs* directories that are selected based on authentication.

Finally, consider how sub-identities simplify the administration of a network authentication service such as Kerberos [35]. A traditional Kerberos installation has a globally shared user database, but it also requires the creation of local users in */etc/passwd* on each machine, corresponding in name and attributes to users in the global database. This can be an enormous administration hassle for large sites. Sub-identities simplify the administration of network logins by divorcing the user database from the enforcement mechanism. Suppose that the *root* user on a workstation creates the necessary processes owned by *root:kerberos* to admit Kerberos logins. As users log in with Kerberos credentials, they can simply be assigned new sub-identities such as *root:kerberos:david*. No interaction or coordination with the local user database is thus needed.

CHAPTER 3

BACKGROUND

The abstract model of sub-identity relates to several bodies of work.

3.1 Access control models

There is a large body of work that is concerned with access control models, which are generally broken up into a handful of families:

3.1.1 Discretionary access control

A Discretionary Access Control (DAC) mechanism allows the owner (who is usually also the creator) of an object to dictate who may have various kinds of access to the object. A well-known example of DAC is Unix mode bits, which are 9 bits arranged into three sets of three bits, each set of bits determining whether the object is readable, writable, and executable by each of the owner, the object's group, and any user. For example, the */etc/passwd* file often has the (octal) mode bits 0644, signifying that its owner has the read and write permissions and its group and other users have read permission. Access Control Lists are another example of DAC.

The sub-identity model is not a form of Discretionary Access Control, but a DAC mechanism may co-exist with a mechanism that implements the abstract model of sub-identity. Like other blends of access control models that include DAC like Flask [34] and REMUS [5], where the sub-identity model and DAC disagree, the sub-identity model overrides any permissions specified by the owner of an object. For instance, if *alice:browser* sets the Unix mode bits of a file such that only the owner may read and write the file, the sub-identity model still grants *alice* complete access to the file.

The property of implicit access to files belonging to sub-identities could be encoded in DAC by explicitly giving access to each of the identity's ancestors, but it would be on an "at-will" basis, since each identity can remove access to a file from its ancestors by modifying the mode bits or ACL of the file. That is, the "effective *root*-ness" would depend on the goodwill of the sub-identities.

The converse, implementing DAC in terms of sub-identities, does not seem to make sense, because access control in the sub-identity model is based on the relative ancestry of the owner of the object, over which the owner has no control.

3.1.2 Lattice-based / mandatory access control

Denning formalized [9] and Sandhu refined [32] a model of Lattice-Based Access Control (LBAC), which is a mechanism enabling policies that dictate the direction in which various types of information in a system may flow. Mandatory Access Control (MAC), which was developed in order to assure that classified documents do not flow "down" into a lower classification level, is a special case of LBAC. The canonical example of MAC is that of a military multi-level security system involving the security labels "Unclassified", "Classified", "Secret", and "Top Secret". Such a system makes strong guarantees about the propagation of information by enforcing the constraint that if a subject has a given clearance, such as “Classified”, then it may not write to any file with a lower classification level, such as “Unclassified”. Thus, information can not leak “down” the lattice towards a lower classification label, a very desirable property for sensitive military applications. These types of access control are considered “mandatory” because they impose strict limits on the operations that an *object* can undergo, in terms of the *subjects* that act on them. Here, a *subject* refers to a program, such as a shell or editor, acting on behalf of a human operator, who is termed a *user*.

Sub-identity is not concerned specifically with the direction of information flow in a system, so it does not fall into the realm of LBAC or MAC. However, note that systems that implement LBAC have relatively static classification levels, which are typically administered by a single security officer. One of the basic ideas of the sub-identity model is to give users the power to create protection domains — this idea could be carried over into LBAC, allowing users to create classification levels in order to give more fine-grained information flow constraints. Allowing individuals to add security labels to an LBAC model would not enable any downward motion or “leakage” of information, but would allow individual users to further compartmentalize information according to organizational concerns, such as sensitive projects with finer-grained security labels than just “Unclassified”, “Classified”, and so on.

The sub-identity model could, in the absence of a DAC mechanism, be used to partially emulate LBAC and thus MAC. The basic idea would be to think of the sub-identity hierarchy as the lattice. Consider a sub-identity hierarchy consisting of the identities *unclassified* (the root of the hierarchy), *classified*, *secret*, and *topsecret* (the leaf of the hierarchy). In this case, a subject running as the *classified* identity would be unable to write to objects owned by *unclassified*. However, the *classified* identity would also be prevented from reading any objects owned by *unclassified*, which contradicts the semantics of LBAC. Further, the lattice formalism requires there to be a “top” security label as well as a “ground” one, which means that lattices are represented in general as directed acyclic graphs (DAGs), which are not supported by the abstract model of sub-identity. So, there is no robust mapping from sub-identity to LBAC.

Could LBAC be used to emulate the abstract model of sub-identity? First, most LBAC systems have a fixed or semi-fixed set of security labels that the administrator or security officer is responsible for maintaining, so unless it were an LBAC implementation that allowed users to define new security labels, one of the basic requirements of the sub-identity model would not be satisfied. Second, in LBAC, subjects are allowed to read objects with a security label “less than or equal to” their security clearance, and are allowed to write objects with a security label “greater than or equal to” their security clearance, where “less than”, “greater than”, and “equal to” are interpreted as comparisons on relative order in the lattice. Because of this kind of far-reaching permission that extends both up and down the lattice depending on the operation involved, and because of the inability to create new security labels, LBAC would be an inappropriate base on which to approximate the abstract model of sub-identity.

3.1.3 Role-based access control

Compared to DAC and MAC, Role-Based Access Control (RBAC) is a more recent development, which stemmed from the needs of businesses and commercial entities. In RBAC, the administrator or security officer defines *roles* to consist of sets of *permissions* on objects. These permissions can allow any sort of access, such as appending, modifying a record, sending a message, and so forth. In addition, the administrator assigns zero or more roles to each *user*.

A canonical example of the necessity of the separation of duties according to role is a company that is performing "clean room" reverse engineering. The main legal concern in such a situation is that the reverse engineering team may only communicate to the clean room reimplementation team through documentation produced during the course of reverse engineering. The "reverse engineer" role would be given full access to files relating to the product being reverse engineered, as well as read and write access to output documentation files. The "clean room" role would be given full access to the reimplementation code base, as well as read access to the reverse engineering documentation files, but no access to the files relating to the product being reverse engineered. In addition, a constraint would be created that prevented any user from being assigned to both the reverse engineer role and the clean room role. This demonstrates how a clean separation of concerns can be easily implemented in RBAC.

Thus, the roles form an extra layer of abstraction over permissions that ease administration, since promoting a user's access can mean merely adding another role to his set of allowed roles. In [33], Sandhu et al formalize role hierarchies and also describe a way to apply RBAC to the management of RBAC by formulating administration tasks as first-class RBAC permissions.

Of the existing access control models, the sub-identity model is closest to an instantiation of this rich definition of Role-Based Access Control. The abstract model of sub-identity can be approximated in RBAC as follows: Each identity in the sub-identity hierarchy can be seen as a role, with full permissions over all the objects "belonging" to the identity. In addition, each role carries a cardinality restriction such that it can only be assigned to a single user, which, for the lifetime of the role, is the user who created it. When an identity creates a sub-identity, it can be seen as exercising its administrative permission to create a new role. Similarly, when an identity destroys a sub-identity, it exercises its administrative permission to delete that role. The "effectively root" relation of the sub-identity model naturally derives from the role hierarchy, since roles are defined to have at least the set union of their sub-roles' permissions.

The sub-identity model could not be used to approximate RBAC, however, because of the lack of the role concept.

3.2 Operating systems

The MULTICS operating system [30] provides two security mechanisms that bear on this work. First, the GE-645 provided eight *protection rings*, which are a common feature on modern CPUs. Generally, code running in a given protection ring can perform any access to any higher-numbered ring, but may only interact with lower-numbered rings through special gates, such as the trap instruction, which signals that the process would like to enter a system call. The sub-identity model could be applied to the operating system and the system software, with higher-numbered rings representing sub-identities. Second, MULTICS user names had three parts, of the form *user.project.compartment*, and its ACLs supported matching against any or all parts thereof. Thus, if the user managed his ACLs in an ordered fashion, he could approximate a three-layered user hierarchy.

Recent versions of Microsoft Windows come with a security measure called User Account Control (UAC), which encourages the use of non-privileged users. This defense measure increases the security of Windows by reducing the incentive to use the Administrator account for daily computing. It also uses a file and registry virtualization feature that allows legacy applications that assume they have privilege to run as unprivileged users by providing a copy-on-write "view" of parts of the registry and filesystem. Although the mechanism is different, this feature of UAC can fool applications into thinking they have privilege, like a similar concept in the model of sub-identity. However, while UAC encourages the use of a two-level hierarchy of users similar to the standard Unix model of "*root* over all", it does not provide individual users with the tools they need to protect themselves. Specifically, it does not allow users to create protection domains, but focuses on reducing the inconvenience of using a non-Administrator account.

Finally, several security enhancement projects have targeted Unix, attempting to cleanly add new forms of access control to existing systems. These projects usually require modifications to the kernel, whether at the source level or at the module level. For example, TRON [4] adds capabilities to Unix, Flask [34], REMUS [5], and AppArmor [3] all add MAC, DTE-Unix [2] adds domain and type enforcement, and SELinux [25] and RSBAC [26] add both MAC and RBAC. All of these projects seek to enforce some additional access control semantics, without disturbing the Unix concept of identity. The end result of such efforts is considerable power in the hands of administrators or security officers, but no provision is made for users to manage their own security situation.

3.3 Privilege separation

Privilege separation [7, 28] is a technique that separates out the part of a process that requires privilege and runs only that part with privilege. A common paradigm in modern Unix systems is a process that needs to listen on a low-numbered port, requiring privilege to do so, but that immediately forks and drops privilege once it establishes a connection. Two particular examples of this are OpenSSH and Apache, which listen on ports 22 and 80, respectively, and which drop privilege once they've forked a process to handle the incoming connection. In general, a process is started with privilege and forks a copy of itself, whereupon the privileged process drops into a low-functionality mode that waits for and services requests from the unprivileged process, whereas the unprivileged process proceeds as normal except for sending requests for privileged operations to the privileged process.

However, this facility is usually only available to processes which are started as the superuser, since that is the only user authorized to perform the *setuid* system call, which is necessary in order to drop privilege. The sub-identity model provides a path for unprivileged processes to further drop privileges. Although the privileged process in practice typically involves the superuser so that low-numbered ports can be opened or sensitive configuration files can be read, there are nonetheless scenarios where it would be useful for a regular user to be able to drop privilege.

Since the toolkit implementation of the ideal model of sub-identity uses PAM and the *setuid* facility, it can additionally be said to employ privilege separation.

3.4 Sandboxing

Sandboxing is a well-researched technique for running untrusted code. A supervisor process is responsible for running an untrusted program while auditing its external operations via a reference monitor or performing them on the program's behalf via a delegate. The trapping technique may be the debugging interface [18, 27, 38], a kernel module [19], system-call reflection [21], or binary rewriting [22]. In addition to exploring trapping methods, various sandboxes have explored containment policies, such as associating rights with programs [1, 8], with data [20], or by deferring writes into a transaction which can be audited after execution [24].

While an ideal technique for prototyping new concepts in access control, there are significant disadvantages to sandboxes. As Garfinkel noted [16], the *ptrace* interface between the supervisor and trapped process is extraordinarily complex and subtle. Trapping system calls for sandboxing incurs a significant performance penalty, and the interface that a sandbox provides to applications is often incomplete or unreliable as compared to the native operating system interface. Few sandboxes support the full range of system calls, and many do not support multiple processes, multiple threads, or other complex interfaces. Thus, while a valuable research technique, sandboxing is not an appropriate substitute for an operating system access control facility.

3.5 Virtual machines

Virtual machines have been applied to isolation of faults for security [17, 23]. They are useful when one wishes to isolate a program or operating system completely from its surrounding environment. However, they are not an ideal mechanism for defining or enforcing access control, because they are relatively heavyweight.

A hierarchy of sub-identities could be constructed out of nested virtual machines, as suggested by Ford et al in [14]. While providing both containment of malicious code and some level of implicit control over sub-identities depending on the exact virtual machine technology used, such an approach is unwieldy at best. This is because virtual machines are complicated pieces of software that generally require non-trivial administration skills in order to manage. Consider just the act of setting up a single virtual machine such as VMWare: one must generate disk images and install an operating system and system software into the virtual machine. The creation and management of virtual machines is an activity only accessible to those already skilled in system administration, and is overkill for regular users who only wish to create protection domains.

CHAPTER 4

TOOLKIT IMPLEMENTATION

The abstract model of sub-identity can be approximated in a modern operating system in many ways. In this chapter I discuss the implementation of the abstract model as a user-space identity toolkit.

4.1 Motivation

Previous work includes implementation of the abstract model of sub-identity using user-space sandboxes [37]. Many observations from that work motivated the implementation of sub-identity as a toolkit. For instance, the correctness of sandboxes is difficult to verify, and their complexity rivals that of complete operating systems [16]. In addition, the type of sandbox used in [37] imposes a large performance penalty, since every system call is interposed upon.

4.2 Drop-in-ness

Since a goal of usable security is to be used, the ability to install a security measure into a wide variety of already-running systems is appealing. The sub-identity toolkit was designed with this "drop-in"-ness in mind; It consists of a set of tools that provide an interface to the operations on sub-identity described in section 2.1, an optional Pluggable Authentication Module called `pam_subid.so` to facilitate some of the operations, and an optional Filesystem in Userspace (FUSE) filesystem called **subidfs** that provides implicit filesystem access.

The toolkit can be dropped into a wide variety of systems: The tools themselves require only that the *setuid* bit is honored. `pam_subid.so` only requires that the subsystems that need to perform sub-identity-related authentication decisions support PAM. This is likely to be the case in any modern Unix-like operating system. Since `pam_subid.so` is not required, PAM support is not required in order to install the sub-identity toolkit. Likewise, since **subidfs** is not required, FUSE support is not required either.

4.3 Implementation details

The tools maintain */etc/subusers*, a secondary identity database that records the ancestry relationships between users. All of the tools access this database in order to determine whether requested operations should be permitted.

Each supported sub-identity operation is implemented by an aptly-named tool. In the following discussion, the user name that calls each tool is denoted *caller*.

4.3.1 Creating sub-identities

subuseradd [-q] [-h] ⟨name⟩

Creates a new user, directly inferior to the calling user. When the [-q] option is given, the ⟨name⟩ given is interpreted as already fully-qualified. In this way, it is possible to create sub-users that are at the "top level" of the identity namespace.

When **subuseradd** is invoked, it gains the effective user ID (*euid*) of *root* through the *setuid* facility. It needs this *euid* to add users to the system with **adduser** and to write to the file */etc/subusers*, which is world-readable but only writable by *root*, like */etc/passwd*. It makes sure that the requested user name does not already exist on the system before creating the new user. The new account is created in the disabled state so that it is impossible for an attacker to guess its password. Finally, it amends */etc/subusers*, adding the relationship "*caller* is the parent of ⟨name⟩". Further details on the format of */etc/subusers* are in section 4.3.5.

When the [-h] option is given, the home directory for the new user is created as a subdirectory of the calling user's home directory. The [-h] option is thus called the "home-in-home" switch. This approach of placing home directories inside the parent's home directory is more aesthetically pleasing than having a single, flat */home* directory containing every home directory, regardless of identity hierarchy depth. In addition, if [-h] is forced with a compile-time definition, its presence eases accounting of disk space and maintenance of quotas.

4.3.2 Deleting sub-identities

subuserdel [-q] ⟨name⟩

Deletes the user given by ⟨name⟩, which must be inferior to the calling user. The [-q] option behaves as before, causing ⟨name⟩ to be interpreted as a fully-qualified user name.

subuserdel also attains the *euid* of *root* through the *setuid* facility. It needs this privilege to remove users from the system with **userdel**, and also to write to */etc/subusers*. When invoked, **subuserdel** first determines whether ⟨name⟩ has any sub-identities. If so, it prints an error message and exits. If not, it invokes **userdel** upon ⟨name⟩, recursively deletes ⟨name⟩'s home directory, and removes ⟨name⟩ from *caller*'s list of sub-identities in */etc/subusers*.

4.3.3 Changing file ownership

subuserchown [-q] ⟨name⟩ (⟨file⟩)+

Makes ⟨name⟩ the owner of each ⟨file⟩ that belongs to *caller* or any of *caller*'s descendants. The [-q] option behaves as above.

subuserchown requires *root*'s *euid* to successfully invoke the *chown* system call on filesystem objects that don't belong to *caller*. The first thing it does is ensure that ⟨name⟩ either is *caller* or is a sub-identity thereof. It then simply loops over all the ⟨file⟩s, making sure that each one belongs to *caller* or one of its sub-identities before invoking the *chown* system call. Ordinarily, the superuser is the only user capable of changing file ownership. **subuserchown** is thus necessary on a Unix system to give users control over their sub-identities' files.

To illustrate the necessity of **subuserchown**, consider a situation where *alice* causes her sub-identity *alice:browser* to download a sensitive file — an export of her banking information, for instance — into *alice:browser*'s home directory. The operating system prevents *alice* from accessing the file, unless *alice:browser* makes it world-readable. However, doing so for even a short amount of time effectively "lets the cat out of the bag," — that is, allows other users access to the file — so some other method for providing access to a file must exist. Even on a system with more expressive access controls (such as POSIX ACLs), the user would be required to manipulate the ACL, which is beyond many users.

Therefore the **subuserchown** command is necessary to enable superior users to lay claim to files owned by inferior users. It is able to perform this action because the *setuid* bit allows it to straddle the domain between two users. See also section 4.6.2 for a way this restriction can be lifted.

4.3.4 Assumption of identity

subusersu [-q] ⟨name⟩

Invokes ⟨name⟩'s default shell as that user. The [-q] option behaves as above.

Inheriting *root*'s *euid* allows **subusersu** to successfully invoke **/bin/su** without providing the password for ⟨name⟩. This is important because by default the accounts of sub-identities are disabled — that is, their password hash in */etc/passwd* is set to a string like "`*`" or "`!`". No password can possibly hash to one of these strings. **subusersu** is thus required for users to be able to assume the identity of their sub-identities. Another method for providing access to sub-identities is through `pam_subid.so`.

On systems that support Pluggable Authentication Modules, `pam_subid.so` offers a more transparent authentication method since it exposes hooks into the sub-identity model to the system's existing authentication scheme. For example, if `pam_subid.so` is installed and the line

```
auth sufficient pam_subid.so
```

is added to */etc/pam.d/su*, then users can invoke **su** instead of **subusersu**.

The way this works is as follows: a user runs a PAM-enabled program such as **su**, which is linked against the PAM library *libpam*. *libpam* processes each line in */etc/pam.d/su*, following the directives contained therein. Encountering the line concerning `pam_subid.so`, *libpam* calls *dlopen()* on the module and calls its authentication function. `pam_subid.so`'s authentication function verifies that the calling user (identifiable through the *getuid* system call) is an ancestor of the requested user (given by the *libpam* function *pam_get_user*). If so, it returns success. Since `pam_subid.so`'s approval is *sufficient* for the *auth* authentication task type, **su** proceeds just the same as if the user had successfully authenticated with a password.

4.3.5 Data structures

While the tools do not alter the format of */etc/passwd*, they do add and remove users and maintain a secondary database in the file */etc/subusers*.

This file describes a tree structure as an adjacency list. It consists of lines of the form

```
parent:child1,child2,...,childN
```

This line indicates that *child1* through *childN* are direct descendants of *parent*. There can be many such lines for a given parent user.

As in the abstract model, the sub-identity hierarchy is a tree. However, this file format could describe other topologies, such as singly- or multiply-rooted directed acyclic graphs or even arbitrary graphs with cycles.

4.4 Special cases

On systems which support sufficiently long user names, the issue of whether to specify the superuser as the ancestor of every user on the system is an aesthetic one. In a sense, it could be implied that every identity on the system exists by the grace of the superuser, so the *root:* prefix can be dropped from each identity. This minor asymmetry also lends itself conveniently to traditional filesystem placement of the superuser's home directory on a different partition than the one containing */home*.

Another issue is the implementation of the identity deletion operation, **subuserdel**, which carries some additional subtlety. Because files' ownership is recorded in the filesystem with the *uid* of the owner, once the user is deleted from */etc/passwd*, that *uid* becomes disassociated from any user name. Another user could be subsequently created and assigned the now-vacant *uid*. The new user would then have access to the files left by the previous holder of the *uid*. There are at least two possibilities for dealing with this situation.

The distinction here is one of garbage collection: Should files persist past the lifetime of the identity that created them? This approach corresponds roughly to allocating memory on the heap, since such memory remains valid when leaving the scope where it was allocated.

One solution is to "bubble up" the ownership of the orphaned files; that is, the **subuserdel** command could change the owner of each file owned by the moribund sub-identity to its direct parent. This way, once the sub-identity has outlived its usefulness and is deleted, the parent user can sift through the remains at his leisure, without worry that another user will gain access to the files. This approach can be seen as allocating "on the heap", since the files persist beyond the life of their owner.

The other approach involves the files being allocated "on the stack", such that the files are deleted when their owner "goes out of scope" — that is, when their owner is deleted. What this means is that the **subuserdel** command would be responsible for deleting all the files belonging to the users it deletes. In the interest of simplicity, **subuserdel** favors this approach, simply using the [-r] option of the system's **userdel** command, which deletes the user's home directory and mail spool.

However, fully realizing either of these approaches would require locating every file owned by the user to be deleted. There is ordinarily a fairly small candidate set of directories to look in for such files, namely, those that the user has write access to. However, since the **subuserchown** command allows the user's ancestors to gives files away to the user, the candidate set becomes the union of all the directories that the user's ancestors have write access to. This seemingly complicated problem is easily solvable by deeming that the responsibility to delete files lies with the identity that has write access to the directory containing them. Even if a user gives away a file in his home directory to one of his sub-identities, he must consciously do so, so the burden to either delete the file or reclaim ownership lies with him.

4.5 Potential avenues of attack

On Unix systems, when filesystems become full, various types of serious widespread breakage can occur. To mitigate this risk, Unix systems often set aside a certain amount of space on each filesystem for the superuser, so that he has some breathing room to log in and "un-wedge" the system. **subuseradd**, which any user can call, lengthens the files that store ancestry information (*/etc/subusers*) and the user database (*/etc/passwd*). Since these files are owned by *root*, they are accounted under *root*'s reserved disk space. It is thus possible for a user to generate sub-identities until disk space is exhausted, creating a denial of service.

To solve this, it would be trivial to add to the toolkit a mechanism for enforcing policies provided by the administrator. These policies would allow the administrator to place limits the depth of subtrees of the sub-identity hierarchy and also on the fan-out of each node in the tree.

4.6 Failures of the toolkit to the abstract model

While the toolkit implements the abstract model of sub-identity, it cannot do so completely. It deviates from the abstract model in the following ways:

4.6.1 UNIX username namespace paucity

First, the hierarchical nature of the namespace is imperfect. In theory, an implementation could store fully-qualified names such as *root:alice:browser*. However, many pre-existing system tools and programs can limit the practical length of user names, so storing and using fully-qualified names is not always possible. For instance, in our testing, the **adduser** command in Debian "sarge" only supports user names of length 32 characters or less, and group names of length 27 characters or less.

In order to get around this restriction, it is convenient to allow users to create sub-identities that are not qualified by their own user name. The toolkit's **subuseradd** command supports an option that allows the caller to create these "top-level" usernames. **subuseradd** prevents users from attempting to create the same sub-identity twice. Thus, although *alice* is free to create unqualified sub-identities, she cannot create top-level identities that have already been created. For instance, if the top-level identity *browser* exists on the system, no other user can create it.

Second, the toolkit does not provide implicit privilege over their inferior users. That is, superior identities are not "effectively *root*" to their sub-identities. While the sub-identity toolkit does implement the model of sub-identity, restrictions of the Unix environment and the requirement that the toolkit be a "drop-in" set of utilities result in the user needing to explicitly perform certain actions instead of having them be implicitly allowed by the operating system. For instance, Unix semantics prevent signals from reaching processes owned by other identities. So, if *alice* wants to send a signal to a process owned by *alice:browser*, she must take the extra step of assuming the identity of *alice:browser* with the **subusersu** command.

4.6.2 Adding implicit filesystem access

Since users don't have implicit access to files belonging to their sub-users, they must explicitly use the **subuserchown** command if they want to retrieve such a file from a sub-user. This extra step distances the toolkit implementation even further from the abstract model and increases the cognitive load on the user; if at all possible, users should not have to jump through any extra hoops to gain the access they should have implicitly.

Implicit control of files by superior users is limited by the access control scheme supported by the filesystem. While it is possible to encode the "effectively root" file semantics of the sub-identity hierarchy into filesystem permissions, such an implementation would be problematic.

Traditional Unix filesystem semantics allow the owner to specify coarse-grained *permissions* on their files and directories as a vector of 9 "mode bits", each bit determining whether the file or directory is readable, writable, and/or executable by the owner, the group, and by "other", meaning anybody. The coarseness of this discretionary access control mechanism is problematic for some situations, as it does not allow certain access policies to be expressed without some tricky manipulations of groups.

Many modern filesystems support Access Control Lists (ACLs), which allow the user to enumerate precisely which rights are to be allowed to which users for each object they own. However, as Zurko and Simon noted:

ACLs are the assembly language of security policy. [43]

This is because each filesystem that supports ACLs implements them in slightly incompatible, idiosyncratic ways, forcing users to internalize the ACL interpretation algorithm for each filesystem that they use.

Because the interpretation of ACLs and Unix mode bits is obscure or cumbersome, even if sub-identity semantics were formulated in terms of ACLs or mode bits, users would most likely be reluctant to disturb this house of cards. Therefore, rather than encoding implicit privilege into an existing permissions model, it is attractive to move the implementation down a layer, into the filesystem itself. The Filesystem in User Space project (FUSE) empowers Linux users to write their own filesystems without requiring changes to the kernel. FUSE consists of a kernel module, a userspace-to-kernelspace communication device, and a dynamically linked library that allow filesystems to be implemented as a userspace process. Since FUSE can be installed into a running system, it fits the "drop-in" criterion.

The toolkit includes a FUSE filesystem called **subidfs** that mirrors the root directory, providing a view of the filesystem with the additional semantics of implicit filesystem privilege over sub-identities. It provides this view on a filesystem mount point, */mnt/subidfs* by default. To grant access to identities on behalf of their sub-identities, **subidfs** must be executed as the *root* user.

Filesystems receive system calls such as *chown*, *chmod*, *unlink*, and so forth. FUSE exposes to the filesystem process the *userid* associated with the process that invoked the system call. **subidfs** can *stat* the file that is referred to by the system call to determine the *userid* of the object's owner. Armed with these two *userid*s, **subidfs** can refer to */etc/subusers* to determine whether to "short-circuit" the underlying filesystem's access control logic and allow the request to proceed. It can disregard this access control logic because it executes as *root*.

Thus, through the mirrored mount point, users can freely perform any filesystem operations such as *chown*, *chmod*, and *unlink* on files belonging to their subidentities. If the superuser provides a *setuid-root* utility that *chroot*s to the (well-defined) mount point of **subidfs**, then users can use this to *chroot* into the mount point so that they no longer have to explicitly operate in */mnt/subidfs*. However, because of the overhead associated with FUSE, the user may elect to not *chroot* into the **subidfs** mount point, and simply access files in */mnt/subidfs* when explicitly necessary. See section 4.6.2.1 for further discussion of this possibility.

This approach to granting implicit privilege in the filesystem is not without its caveats. Because **subidfs** does not know about any site-specific access control mechanisms that may be in place, it may not be suitable for deployment at every site. Rather than erring on the side of permissibility, **subidfs** only checks Unix mode bits and */etc/subusers*. The concern is that because **subidfs** does not honor any additional access control mechanisms, users of such system, expecting homogeneity of access on the filesystem, may have an adverse user experience. However, the same essential functionality is present in the toolkit through the **subuserchown** tool.

The toolkit, including the tools, `pam_subid.so`, and **subidfs** can be downloaded from `http://cse.nd.edu/~ccl/software/subid`. It runs on (at least) Debian "sarge" and Red Hat Fedora Core 5.

4.6.2.1 Subidfs performance

The addition of implicit filesystem control over sub-identities is appealing, however, lifting the cognitive load from the user must not come at the price of intolerable performance. If every operation carried out through **subidfs** were to incur a noticeable latency, the cumulative annoyance could offset the convenience of implicit privilege.

Table 4.1 lists several "microbenchmarks" showing the amount of overhead that is incurred by **subidfs** through FUSE. These figures are the result of averaging the time required to do each system call repeatedly 10000 times on a 2.8 GHz Pentium 4, averaged over 10 trials. Even though some operations do not seem to be overtly influenced, *write* and *open* system calls tend to incur a hit of around an order of magnitude in latency.

Table 4.2 shows the results of the Bonnie benchmark on **subidfs** and on the native hardware. These results show that **subidfs** incurs a performance penalty of only a factor of two for long periods of sustained operations.

However, such microbenchmarks do not demonstrate whether it is feasible to do all of one's work inside of **subidfs**. To that end, I started a shell as *root* and used the *chroot* system call to set its filesystem root to the **subidfs** mount point, and then used that shell for some everyday operations such as running Firefox. Firefox is responsive and performant while running inside **subidfs**.

I also measured the time required to build *cctools*, a distribution of the Cooperative Computing Laboratory's software, inside about outside of **subidfs**. These numbers were gathered as follows: a script times the operations of extracting the *cctools* archive, running the **configure** script, running **make**, and deleting the directory with **rm -rf**, both inside and outside of **subidfs**. In between test runs, the script both calls **sync** and takes advantage of a facility added to the 2.6.16 Linux kernel that flushes the page cache, *dentry* cache, and *inode* cache when the ASCII character `3` is written to */proc/sys/vm/drop-caches*. Without this extra measure, some steps of the test were completing unrealistically quickly because, for instance, **configure** always checks the same 20 or so system libraries, and they were fully cached after the first iteration.

The results of this experiment are shown in Table 4.3. Each step of the build process takes incrementally longer inside **subidfs** than on the native filesystem. However, this overhead is scarcely noticeable.

These results favor the use of **subidfs** as the preferred method for allowing implicit sub-identity control in the filesystem.

TABLE 4.1

SYSTEM CALL OVERHEAD INCURRED BY SUBIDFS

system call	native	**subidfs**
getpid	0.1861 ± 0.0012 μs	0.1857 ± 0.0003 μs
write 1 B	1.2687 ± 0.0066 μs	29.4926 ± 0.0956 μs
write 4 KB	2.8549 ± 0.0285 μs	34.9164 ± 0.0938 μs
write 16 KB	19.2197 ± 0.0857 μs	141.7822 ± 0.3382 μs
write 64 KB	19.1968 ± 0.0997 μs	142.0165 ± 0.1683 μs
write 256 KB	19.2795 ± 0.0734 μs	142.0126 ± 0.1363 μs
read 1 B	0.6667 ± 0.0138 μs	0.6195 ± 0.0054 μs
read 4 KB	1.3137 ± 0.0167 μs	1.2549 ± 0.0311 μs
read 16 KB	1.1790 ± 0.4136 μs	1.1317 ± 0.3981 μs
read 64 KB	1.1789 ± 0.4138 μs	1.1316 ± 0.3982 μs
read 256 KB	1.1789 ± 0.4138 μs	1.1316 ± 0.3982 μs
stat	1.5448 ± 0.0103 μs	2.5082 ± 0.0392 μs
open	2.7872 ± 0.1309 μs	39.4873 ± 0.4774 μs

TABLE 4.2

SUBIDFS BONNIE BENCHMARK RESULTS

	Sequential Output			Sequential Input		Random
	Per Char	Block	Rewrite	Per Char	Block	Seeks
	KB/sec	KB/sec	KB/sec	KB/sec	KB/sec	KB/sec
native	40820	42760	21274	35589	46493	112.4
subidfs	25916	19506	14210	21158	48895	100.7

Results of the Bonnie benchmark both inside **subidfs** and on the native filesystem on the same hardware. For sustained, intensive filesystem operations, **subidfs** is slower than the native filesystem by only a factor of two.

TABLE 4.3

TIME REQUIRED TO BUILD CCTOOLS

	unpack	configure	make	remove
native	0.63 ± 0.017	1.33 ± 0.036	18.51 ± 0.102	0.01 ± 0.002
subidfs	0.79 ± 0.054	1.38 ± 0.105	20.84 ± 0.151	0.06 ± 0.002

Time in seconds required to build the Cooperative Computing Laboratory's *cctools* distribution of tools inside and outside of **subidfs**. The build is broken up into four steps: unpacking the distribution, running **configure**, running **make**, and removing the source distribution and the built files with **rm -rf**.

CHAPTER 5

APPLICATIONS OF SUB-IDENTITY

Application authors can employ sub-identity to isolate the user or the application itself from potentially dangerous operations. Thus, the application author provides for a better user experience by reducing the potential for compromise of the user's private information. In this chapter, I provide some examples of applications of sub-identity.

5.1 Descriptions of the applications

To demonstrate the power and simplicity of the sub-identity toolkit, I applied it to four scenarios: a "safe window" nested X server, a web browser, a web server, and securing remote execution over the network. The nested X server provides a way to run untrusted X applications, to keep them from accessing the X Windows protocol stream. The web browser uses sub-identity to safely execute programs downloaded from untrusted sources. The web server uses sub-identity to safely execute uploaded scripts. A user may use per-task sub-identities to facilitate remote automation without risking compromise of his own files. The necessary code changes to employ sub-identity were minimal in these applications.

5.1.1 Safe X window

The X Windows protocol stream is essentially a bus that applications can listen on, acting on events that concern them. Keystroke events are visible to every application listening to the stream. A malicious application with access to the X display could sniff keystrokes and send them off to third parties. Here I provide a mechanism to isolate potentially malicious applications in their own X server with a separate X protocol stream, nullifying the keystroke sniffing attack.

I adapted a script to take one argument, the name of a sub-user of the caller, and present the user with a *Xnest*, or nested X Windows server. The script generates a "magic cookie" that will allow access to the nested X server to the bearer and merges it with the sub-user's X authority file, allowing the sub-user to connect to the nested X server but not the enclosing one. The window manager that manages this nested X server runs as the named sub-user, so any applications that the user starts will run as the named sub-user too. The script gives the nested X server a distinctive background to provide a visual cue, and sets the title of the window housing the X server to the name of the sub-user. An example of this setup is illustrated in figure 5.2. X servers can be recursively nested inside others, so if a user's situation demands it, sub-users can run nested servers, until resources are exhausted.

This visual metaphor fits well with the idea of the user hierarchy, since everything that runs in the window is running as the sub-user.

5.1.2 Safe web browsing

It is a common paradigm to design a computationally expensive application to be downloaded over the Internet and executed on users' systems [11, 36] in order to distribute load to many clients. However, because the user might not fully trust the provider of the application, or because the user cannot be assured that the application has not been tampered with [41], users may want to run such applications inside a protection domain, so that their systems are not at risk of compromise. Running the untrusted application as a sub-user provides such a protection domain.

FlashGot is a Firefox extension that allows a URL to be sent to an arbitrary command on the local system. I wrote a URL handler wrapper script for this extension that automatically invokes a sub-user whenever the user clicks on an executable file. The handler downloads the application from the web page and runs it as a sub-user corresponding to the name of the remote server. This *FlashGot* URL handler wrapper script enables users of Firefox to select "Run as a sub-user..." from the context menu of any link.

The extension uses the fully qualified domain name of the server the application was downloaded from as the name of the sub-user. However, if the application were hosted on a secure web page, the name of the sub-user could be taken from the X.509 certificate presented during the secure connection negotiation. While this would be a better solution, because it would make identity spoofing attacks more difficult, it is not yet implemented. Each time the application is run, if downloaded from the same server, it runs as the same sub-user. Since files owned by that user can persist between invocations, the application has access to any files that it created in the previous invocation. Multiple applications can also run concurrently as the same sub-user. In total, this application of sub-identity required writing two dozen lines of code as a wrapper script.

The question of garbage collection arises when considering this application; some applications' performance could benefit from caching downloaded data and program state locally. This is particularly useful for programs that checkpoint their own status periodically. In addition, allowing state to be stored locally can reduce network traffic, so it makes sense to support persistent protection domains. The question, then, is "when should hard drive space given to a sub-user be reclaimed?" This is simply a matter of local user policy, much as web browsers allow the user to control when cached or downloaded files are deleted.

5.1.3 Untrusted hosted execution

Consider the scenario of a web site hosting tutorials on some programming language. Rather than requiring the users of the site to install an interpreter or compiler for the programming language, the site could offer a service that allows visitors to create an account and upload programs that they've written. Such an approach would make it easier for beginners to learn languages that require environments that are difficult to acquire or compile, such as Haskell or Erlang, would lower the "barrier to entry" for new language learners, and could increase adoption of the language.

I wrote a CGI script that provides this kind of service. Using the sub-identity toolkit, this script accepts uploaded programs, changes their ownership to the identity associated with the connecting user, and then executes them as that identity. Since any user can use the toolkit once it is installed, any user can then run this type of service in their own web server on a high-numbered port. The toolkit insulates the user running the web server from potential attacks included in the payload of an uploaded program.

As an example, the user *webserver* runs a minimal web server called *tiny-httpd* [6], which serves the aforementioned script to a previously-established list of users. The connecting user enters his authentication information, and also a listing of a program. Upon submission of the form and successful authentication, the script saves the contents of the text area into a file, changes its ownership to that of the connecting user with *subuserchown*, moves it into the user's home directory with */bin/su -c* (and thus also `pam_subid.so`), then finally executes it, also with */bin/su -c*. The script then prints the output of the program to the user's web browser.

Currently, the script requires that the uploaded content be a script that doesn't require compilation, since it merely executes the uploaded file. However, it wouldn't be difficult to allow the user to choose from a list of compilers.

One of the traditional reasons for running the web server as the superuser is so that privilege can be dropped to a minimum-privilege user such as *nobody* when serving HTTP requests. This new approach to containment eliminates one reason to run as the superuser; with the toolkit, users and application developers can easily create new protection domains.

5.1.4 Securing remote execution

Often, when setting up any sort of automation task among multiple hosts in a cluster, users set up SSH keys, or store some other secret such as a password in plaintext, so that they don't need to interactively type their password. However, one issue with this method is that it is possible for an attacker to discover or crack the password protecting the SSH key, or to gain access to the file containing the password. If that were to happen, the attacker would have unfettered access

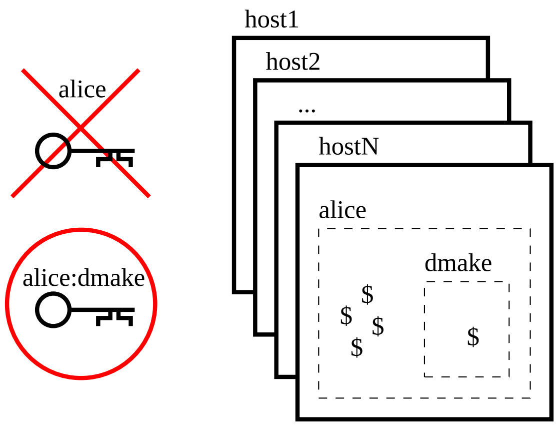


Figure 5.1: Risk of Information Breach with and without Sub-Identity. Here is shown the decreased risk of information breach in the event of a successful attack gaining a stored secret or brute-forcing an SSH key. The dollar signs represent valuable files. They are enclosed with dashed boxes that indicate the limits of each user's access. Clearly, if *alice:dmake*'s account is compromised, the potential for damage is reduced.

to the user's account on the various hosts of the cluster until the user discovered the intrusion and removed the offending key from his SSH authorized keys file or changed his password.

The problem with this scenario is twofold: first, there's no way to keep users from storing their passwords in the clear or allowing access to their accounts through SSH keys, since doing so fills a very real need. Second, the hapless user can only give away access to his whole account, not to any subset of his account. By storing a secret to protect access to his account, the user places the "keys to the kingdom," so to speak, at risk.

If the user stores a secret such as a password or uses SSH keys with a key agent to provide automated network access across a cluster, a successful attack gaining the secret or brute-forcing the SSH key yields total access to the user's account. What is needed is a way to reduce the risk in the event of a successful attack.

If the sub-identity toolkit were installed on all the hosts on the cluster, the user could just create a sub-user on each host, and store a secret for access to that sub-user. Figure 5.1 illustrates this setup. For instance, here *alice* has created a sub-identity *alice:dmake*, which she will use for a distributed build system. Should an attacker gain access to *alice:dmake*, he will not be able to access *alice*'s valuable files, but will only have access to those files which are world-readable and those that belong to *alice:dmake*.

By doing so, the user places the keys to only a controlled subset of the "kingdom" at risk of being stolen.

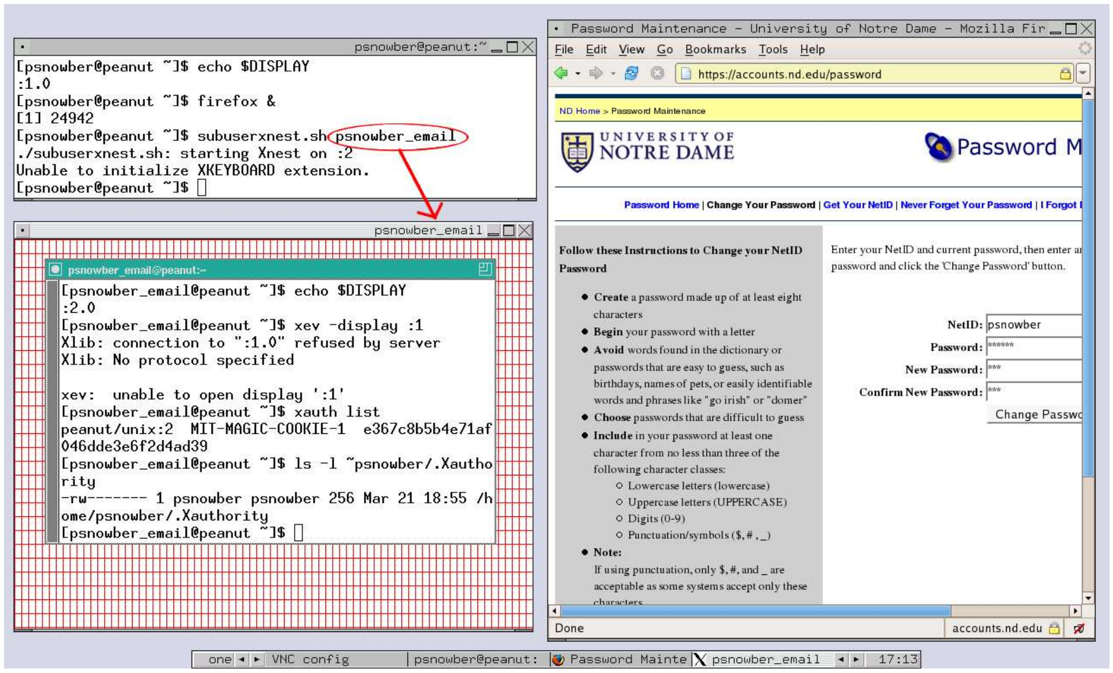
5.2 Evaluation

What follows is an assessment of the resiliency of each of the above applications of sub-identity to attack. Where appropriate, an attack is demonstrated to fail because of the safeguards employed.

5.2.1 Safe X window exploit test

This application relies on the X Windows security model and the MIT-MAGIC-COOKIE-1 authentication scheme. In short, an X Windows server using this authentication scheme either generates or is given a "magic cookie" when it starts, and connecting to such a server requires presenting this cookie. Thus, controlling access to an X Windows server consists of controlling access to the cookie.

The script takes care of giving the sub-user the cookie associated with the nested X Windows server. It does so by generating a new X Authority file with the **xauth** command, populating it with a single randomly-generated cookie. The script runs **Xnest**, instructing it to use the cookie in this file as its magic cookie.


psnowber@peanut:~
psnowber_email
psnowber_email@peanut:~
Password Maintenance - University of Notre Dame - Mozilla Fir
File Edit View Go Bookmarks Tools Help
https://accounts.nd.edu/password
Done
accounts.nd.edu
one VNC config psnowber@peanut: Password Mainte psnowber_email 17:13

```
[psnowber@peanut ~]$ echo $DISPLAY
:1.0
[psnowber@peanut ~]$ firefox &
[1] 24942
[psnowber@peanut ~]$ subuserxnest.sh psnowber_email
./subuserxnest.sh: starting Xnest on :2
Unable to initialize XKEYBOARD extension.
[psnowber@peanut ~]$ 
```

```
[psnowber_email@peanut ~]$ echo $DISPLAY
:2.0
[psnowber_email@peanut ~]$ xev -display :1
Xlib: connection to ":1.0" refused by server
Xlib: No protocol specified

xev:  unable to open display ':1'
[psnowber_email@peanut ~]$ xauth list
peanut/unix:2  MIT-MAGIC-COOKIE-1  e367c8b5b4e71af
046dde3e6f2d4ad39
[psnowber_email@peanut ~]$ ls -l ~psnowber/.Xautho
rity
-rw------- 1 psnowber psnowber 256 Mar 21 18:55 /h
ome/psnowber/.Xauthority
[psnowber_email@peanut ~]$ 
```

ND Home > Password Maintenance

UNIVERSITY OF NOTRE DAME

Password M

Password Home | Change Your Password | Get Your NetID | Never Forget Your Password | I Forgot

Follow these Instructions to Change your NetID Password

- **Create** a password made up of at least eight characters
- **Begin** your password with a letter
- **Avoid** words found in the dictionary or passwords that are easy to guess, such as birthdays, names of pets, or easily identifiable words and phrases like "go irish" or "domer"
- **Choose** passwords that are difficult to guess
- **Include** in your password at least one character from no less than three of the following character classes:
  - Lowercase letters (lowercase)
  - Uppercase letters (UPPERCASE)
  - Digits (0-9)
  - Punctuation/symbols ($, # , _)
- **Note:**
  If using punctuation, only $, #, and _ are acceptable as some systems accept only these characters

Enter your NetID and current password, then enter a password and click the 'Change Password' button.

NetID:	psnowber
Password:	******
New Password:	***
Confirm New Password:	***

Change Passw

Figure 5.2: Visually Intuitive Usage of Sub-Identity with a Safe Window. A nested X server is shown containing a sub-user's X session. The callout shows that the name of the sub-identity is displayed in the title of the nested X server. Since the window manager is running as the sub-user, every application inside the nested X server will also run as the sub-user, providing a visual metaphor to the model of sub-identity. The nested server is shown attempting to read the keystrokes in the enclosing X server with the **xev** program, which hooks into an X event stream.

It uses `pam_subid.so` to add the entry from this authority file to the sub-user's ~/*.Xauthority* file and start a window manager in the nested server. This last action allows the sub-user access to the nested X server.

The key to securing access to the enclosing server's protocol stream is the location of the cookie that grant access to the enclosing server. The only place this cookie exists on disk is in the superior user's ~/*.Xauthority* file, which is inaccessible by the sub-user because of Unix permissions.

This application is illustrated in Figure 5.2, which shows a prevented attempt to hook into the enclosing X Windows server's protocol event stream.

Note that there is a possibility of loss of containment in this application of sub-identity, if the superior user turns off access control to the enclosing X server. In this event, his X server becomes open to a wide variety of attacks, including keystroke sniffing. However, doing so requires an explicit action on the part of the user: he must change the arguments given to **Xnest** in the script. In addition, the superior user could circumvent the X authority model by explicitly giving the cookie to the sub-user, whether by typing it in from memory, or granting the sub-user access to a file containing it.

It should be noted that *Xnest* was written for testing purposes and it is not yet widely believed to have airtight security. While this is an open issue with the implementation of this application, the mechanics of the situation do not detract from the elegance of the visual metaphor.

5.2.2 Safe web browsing exploit test

Figure 5.3 shows an attempted attack that is prevented because the program is run inside the protection domain of a sub-identity.

This application of sub-identity gives protection from attacks by creating a containment area in which to run the downloaded program. When a malicious application attempts to vandalize the system, the attack fails, because of the ordinary Unix protection mechanism that keeps users from attacking each other. Note that normally, these protection mechanisms do little to prevent certain kinds of denial of service attacks, such as attempting to exhaust the space of process identifiers, using up as much processing time and I/O bandwidth as possible, and filling up filesystems. For this reason, the system should also have sensible quotas and, if possible, sensible defaults for maximum numbers of processes and so forth per user in place.

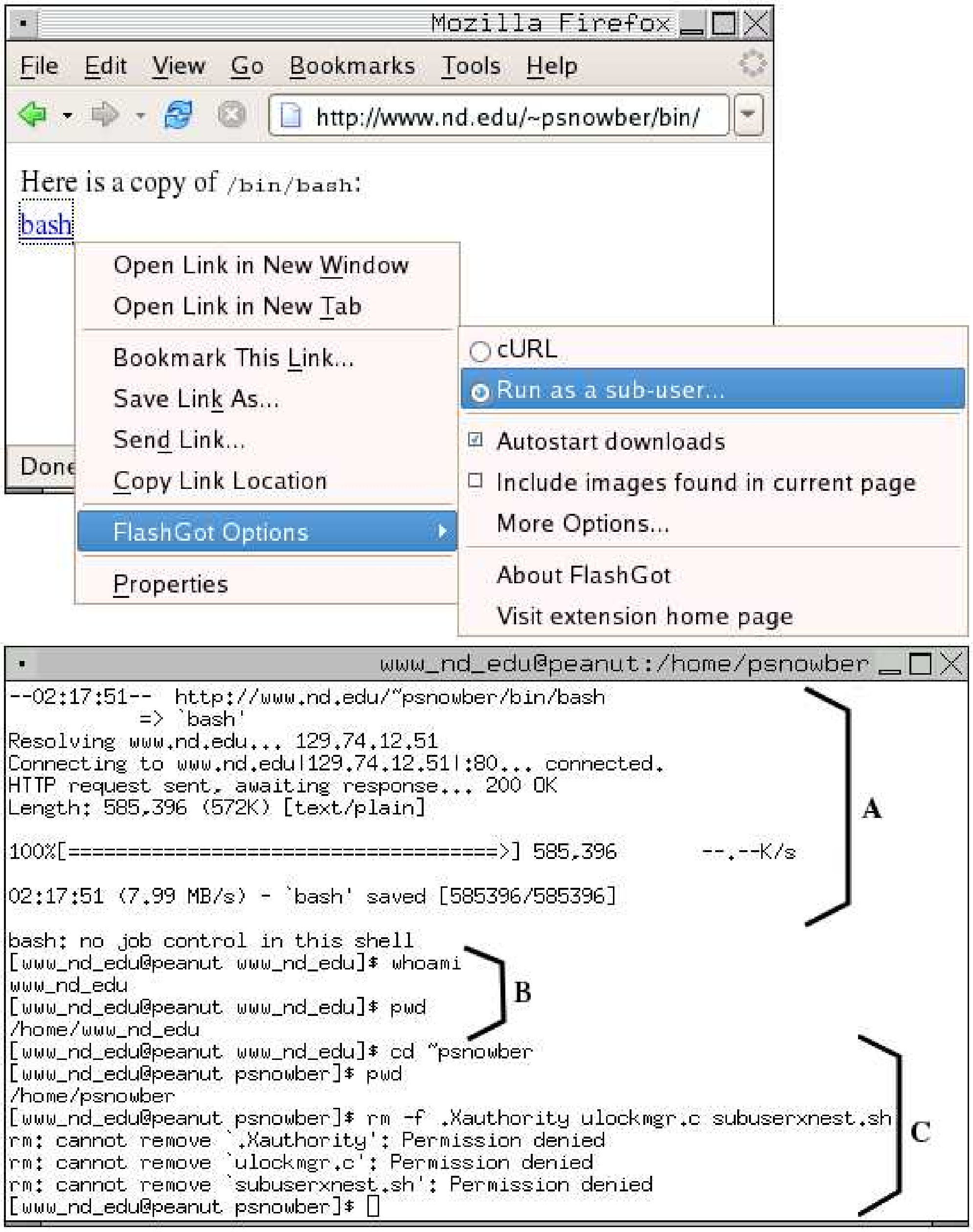
5.2.3 Safe web serving exploit test

This application of sub-identity behaves similarly to the previous one, except that the potentially malicious content is being “pushed” instead of “pulled”. The mechanics of the situation are exactly the same, so the above caveats about denial of service attacks also apply to this application.

5.2.4 Securing remote execution exploit test

Note that this extra measure of security does not make it any more difficult for a would-be attacker to brute-force an SSH key or password; it is concerned merely with mitigating the damage caused by such a break-in.

With that said, the situation is again a close parallel to the above situations. If an attacker should succeed in breaking in, he will find he has access only to those files that are world-readable on the system as well as those owned by the cracked sub-identity. It falls on the user controlling the sub-identity to distribute his “eggs” in his “baskets” appropriately.



Here is a copy of `/bin/bash`:
bash

```
--02:17:51--  http://www.nd.edu/~psnowber/bin/bash
           => `bash'
Resolving www.nd.edu... 129.74.12.51
Connecting to www.nd.edu|129.74.12.51|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 585,396 (572K) [text/plain]

100%[====================================>] 585,396       --.--K/s

02:17:51 (7.99 MB/s) - `bash' saved [585396/585396]

bash: no job control in this shell
[www_nd_edu@peanut www_nd_edu]$ whoami
www_nd_edu
[www_nd_edu@peanut www_nd_edu]$ pwd
/home/www_nd_edu
[www_nd_edu@peanut www_nd_edu]$ cd ~psnowber
[www_nd_edu@peanut psnowber]$ pwd
/home/psnowber
[www_nd_edu@peanut psnowber]$ rm -f .Xauthority ulockmgr.c subuserxnest.sh
rm: cannot remove `.Xauthority': Permission denied
rm: cannot remove `ulockmgr.c': Permission denied
rm: cannot remove `subuserxnest.sh': Permission denied
[www_nd_edu@peanut psnowber]$ []
```

Figure 5.3: Applying Sub-Identities to Untrusted Web Browsing. A modified Firefox is shown that allows the user to run untrusted programs as a sub-identity. On the top is Firefox with the *FlashGot* extension enabled, and the new menu item highlighted. Clicking the menu item creates the terminal on the bottom, running the program which was clicked on. In callout **A**, the program is shown being downloaded. **B** shows that the process is running as a sub-user named *www_nd_edu*. **C** shows a failed attack against the superior user *psnowber*.

CHAPTER 6

DISCIPLINES FOR USING SUB-IDENTITY

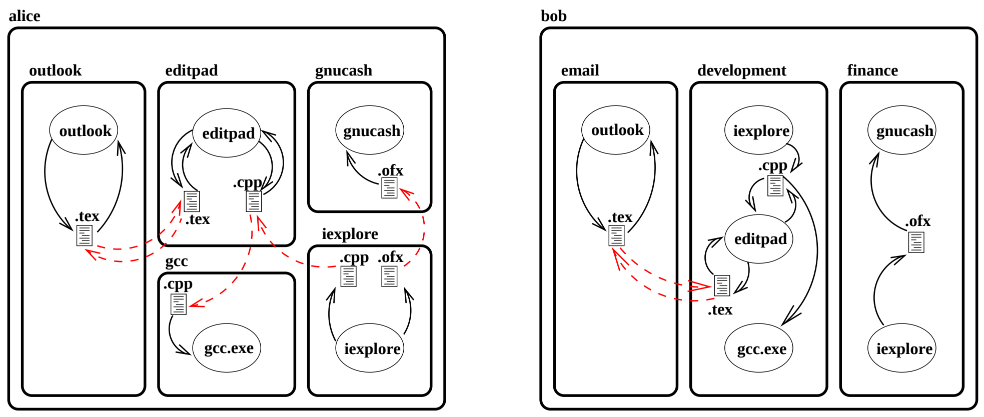
To further illustrate the practical applicability of sub-identity to the working habits of users, we now present a continuum of *disciplines* for using sub-identity.

6.1 A continuum of disciplines

A discipline for using sub-identity is a decision process that a user carries out to determine what identity a given process will run as. These disciplines are meant to provide the reader with an intuitive feel for how one might employ sub-identity in day-to-day computing.

Each point on the continuum represents a trade-off between usability and process isolation. This trade-off stems from requiring the user to intentionally push files between identities — a lax discipline will require fewer such extra steps and will therefore be more convenient, whereas a strict discipline requires more of these steps but will be more resistant to attack. It is up to the user to determine a discipline for using sub-identity that best fits his own comfort level.

The continuum of disciplines has various points that are useful to attach labels to and consider in more detail. Figure 6.1 presents a working example, framed as two specific points along the continuum of sub-identity use: both *alice* and *bob* receive some documentation as an e-mail attachment, which they edit and then



(a) One identity per application

(b) One identity per role

Figure 6.1: Sub-Identity Usage Disciplines. Solid arrows represent operations on files; dashed arrows that cross identity boundaries represent an explicit action that the user must perform. The discipline in *(a)* is more secure, while the discipline in *(b)* allows more convenience.

reply to with their changes. Also, they both download a *.cpp* source file from the Internet, make a few changes to it, compile it with **gcc.exe**, and then run it. Last, they both download their latest banking information from their bank's website and import it into GnuCash. We term the discipline observed by *alice* as the "per-application" discipline, and that observed by *bob* as the "per-task" discipline.

The disciplines given here are just two of the many that are possible, given the flexible control over security that sub-identity gives.

Let us then consider the two disciplines, and how their practitioners *alice* and *bob* fare, if any of the following hypothetical events were to occur: a flaw is discovered in how GnuCash imports banking data, leading to an exploit that allows an attacker to execute arbitrary code; the *.cpp* file the user downloads also contains a stealthily hidden code payload that acts as a Trojan horse; the documentation file that arrives is actually a malicious executable disguised as a *.tex* file.

Since this discussion is focused on how a user may use sub-identity to protect himself, we assume that the host operating system is not itself vulnerable to privilege-escalation attacks.

6.1.1 Per-application

In this discipline, illustrated in figure 6.1(a), user runs every application as its own identity.

Note that if any of the malicious content involves “zombifying” the computer — that is, allowing the machine’s network bandwidth other resources to be used for unauthorized purposes — then sub-identity will not prevent the attack. Because of this, it is necessary for users to continue to use anti-virus and anti-spyware software.

The additional security granted by sub-identity comes at the expense of some usability; every step in *alice*’s workflow that involves a file changing hands between programs becomes a speed bump, since the file must explicitly be transferred across the identity boundary with **subuserchown** or **subidfs**.

6.1.2 Per-task

Many users have more involved workflows involving multiple applications such as an entire office suite, such that the previous discipline would be cumbersome. In contrast to application-specific identities, having task-based identities can reduce the number of times the user needs to explicitly bring a file across an identity boundary.

The second discipline gives every “task” or “role” an identity of its own, potentially with some applications being able to run in multiple identities. However, even though the same application may run in multiple identities, they each store their configuration and so forth in their respective identities’ file areas or registries. This discipline is illustrated in figure 6.1(b).

This discipline is more susceptible to the spread of malware and spyware: When *bob* causes GnuCash to execute the arbitrary code from the payload of the compromised banking information, it also has access to the *finance* identity’s web browser and could thereafter sniff all keystrokes that are typed into the browser. In addition, *bob*’s browsing history could be broadcast to the whole Internet. When *bob* runs the compiled program, his compiler, editor, and *development*’s browser files are all equally compromised. Finally, when *bob* opens the executable disguised as a documentation file, the same breach occurs, and the whole *development* identity is compromised.

6.1.3 Per-user

It should be noted that the opposite end of the continuum from the “per-application” discipline could be called the “Unix” discipline: it is the degenerate discipline in which every application belonging to a given user runs as the same identity.

6.1.4 Hybrid

Over time, a user who is using one of the above disciplines may grow tired of the restrictions imposed and create an identity explicitly for bending the discipline in a specific case:

If a user using discipline 6.1(a) constantly exchanges a file between a given set of applications, it may make sense to allow them to run as the same identity. Conversely, a user who is using discipline 6.1(b) may be alerted to a vulnerability in an application that is part of his main workflow, causing him to isolate the offending program in its own identity.

These users are now using a hybrid discipline that involves some application-specific identities and some role-based or task-based identities; the exact mix of disciplines will vary depending on the situation, the desired level of security, and the level of inconvenience that is tolerable in the name of security.

6.2 Evaluation

The following sections detail the ease of use and the resistance to attack of each of the disciplines for using sub-identity.

6.2.1 Per-application

This discipline is resistant to many forms of malware: when *alice* makes GnuCash execute the malicious payload, the *gnucash* identity is compromised, but the penetration stops there because the identity has access only to a very restricted of files. Similarly, when *alice* runs the compiled program as her sub-identity, it is unable even to corrupt the compiler. If *alice* attempts to open the documentation file, executing the malicious content, her editor and all the files accessible to it be compromised, but the payload cannot access her email address because it exists in another identity.

6.2.2 Per-task

This discipline is more susceptible to the spread of malware and spyware: When *bob* causes GnuCash to execute the arbitrary code from the payload of the compromised banking information file, it also has access to the *finance* identity's web browser and could thereafter sniff all keystrokes that are typed into the browser. In addition, *bob*'s browsing history could be broadcast to the whole Internet. When *bob* runs the compiled program, his compiler, editor, and *development*'s browser files are all equally compromised. Finally, when *bob* opens the executable disguised as a documentation file, the same breach occurs, and the whole *development* identity is compromised.

6.2.3 Per-user (UNIX)

This discipline provides the least resistance to malware, since there is no mechanism protecting users' private files. A user of this discipline can expect their files to be completely compromised, if they are not encrypted or otherwise obfuscated. However, user-applied cryptography is unintuitive for the average user [42]. Users should therefore be encouraged to make use of sub-identities on systems that support them.

6.2.4 Hybrid disciplines

Whether the user is conscious of the continuum of sub-identity disciplines or not, if he chooses to work at some point in between the extremes of the continuum, the consequences of that decision on his security can be subtle.

When a user chooses to run two applications in the same identity, they may be conceptually very different programs, such as a text editor and a C compiler, or an email program and a spreadsheet application. There is a potential for the user to forget the way in which he has drawn the lines surrounding each of his sub-identities. Should this occur, the user could inadvertently suffer a wider compromise than he expects when his email program is infected by malware contained in a spreadsheet.

6.3 Enforcement methods

The use of sub-identity discipline can be either a "best-effort" assignment of applications to identities, or it can be more strictly enforced or made "compulsory", perhaps by the operating system itself or by the user interface of the operating system.

6.3.1 Voluntary enforcement

With the voluntary enforcement method of sub-identity discipline, the user is tasked with starting each application as the correct identity.

Visual cues given by the safe X window detailed in section 5.1.1 make it easier for the user to start applications as the correct identity. For command-line applications, the user could rely on his shell to remind him what identity it is running as, by changing the default shell prompt to include the user name.

This enforcement method is the least onerous of the ones given here, since it allows the user to start applications as a given identity without interference. However, because it doesn't make any efforts to guide the user, a user who is still learning to use sub-identities may make a misstep and compromise the integrity of a trusted identity by running untrusted code as that identity.

6.3.2 Automatic enforcement

With the "automatic" enforcement method, the user is removed to a certain degree from deciding which identity each application should run as. This method of enforcement allows the user to specify the identity each application should run as, while also taking care of the extra step of explicitly changing identity before invoking the application.

To implement this method, the operating system's graphical shell could be altered such that the first time the user runs an application, the shell asks the user what identity the application should run as. The shell would then automatically cause each application to run as the assigned identity.

However, this approach has several usability issues: First, because the identity change is not caused by an explicit action of the user, the graphical shell would likely need to provide some consistent visual clue that applications are being run as a different identity, such as dynamic security skins [10]. Such measures require some operating system support to prevent the visual clues from being faked by programs that want to trick the user into thinking that an application is running with reduced privilege when in fact it is not.

Second, without invasive changes to the graphical shell such as enabling it to perform binary rewriting or system call interposition to catch *exec* system calls performed by the applications it runs, it would not know when applications were starting other subsidiary applications, so it would not be able to impose the user-assigned identity upon the subsidiaries. This could result in an application being run outside the identity that the user has chosen for it. Unless the user is aware of this, he is likely to assume that his files are safe from compromise by the application, which would not necessarily be the case.

This approach would not remove from the user the responsibility for determining which applications should run as which identities. However, if the above issues with this enforcement method were addressed, it would ameliorate some of the danger of inadvertently forgetting to invoke a sub-identity before running an application. This danger would be reduced because the user would have to decide which identity to run each application as only once, because thereafter, the shell could take care of actually invoking the application with the given identity.

6.3.3 Compulsory enforcement

The "compulsory" enforcement method addresses the limitation of the automatic enforcement method that necessitates binary rewriting or system call interposition. While still giving the user the ability to provide a mapping between applications and identities, this method relies on an operating system change to enforce the mapping.

The operating system kernel itself could be modified to enable compulsory enforcement, by adding code to the *exec* family of system calls that ensures that the program is running as the correct identity. Since *exec* and its relatives represent a single point of entry for the invocation of applications, modifying them would suffice to correctly assign each application to its assigned sub-identity.

However, such a mechanism would be intrusive in that the supporting code would need to span both user space and kernel space, as well as the intersection between the two. User space code would be required to interact with the user and to allow the specification of a sub-identity for each application. Kernel space code would need to be added to add the extra functionality to the *exec* family of system calls. Interfacing code between user space and kernel space would be necessary to inform the kernel of the user's application $\mapsto$ identity mappings.

This additional code would not be appropriate for the implementation presented in Chapter 4, because it would violate the "drop-in" requirement. However, if one were designing a new operating system or an implementation of the sub-identity without a focus on installability on a running system, this enforcement method would be more viable.

CHAPTER 7

CONCLUSION

Sub-identity empowers the ordinary user to take charge of his security environment. This work presents an abstract model of sub-identity and described a working, drop-in implementation of that model in the form of a user-space utility toolkit, Pluggable Authentication Module, and a FUSE filesystem. In addition, it also demonstrates four applications of sub-identity that were implemented somewhat trivially with the identity toolkit. Further, it describes a continuum of disciplines for using sub-identity that can be used as guidelines for personal use, allowing the individual to strike his own balance between convenience and compromise isolation.

The experience of implementing sub-identity demonstrates several things. First, sub-identities are easy to use! A simple user interface added to a web browser is accessible to nearly any user, unlike many other complex security technologies. Second, very few code changes were necessary to implement applications that use sub-identities for protection. Experimentation shows that the implementation of sub-identities protects superior users from undesired actions by inferior users, such as unauthorized deleting of files or sniffing of the X Windows protocol stream.

> *Only by permitting the individual user some control of his own administrative environment can one insist that he take responsibility for his work.* - J. Saltzer [30]

Ultimately, sub-identity gives users the tools necessary to protect themselves in an undeniably hostile computing environment. In systems that do not give this power to individual users, each user is a defenseless participant that relies on the administrator for protection. In a modern, highly-networked computing environment, this is no longer acceptable; Users must have tools that allow them to take responsibility for their own safety. Sub-identity is a step towards robustly providing this capability in an accessible way within the operating system.

BIBLIOGRAPHY

1. A. Acharya and M. Raje, MAPbox: Using parameterized behavior classes to confine applications. Technical Report UCSB TRCS99-15, University of California at Santa Barbara, Computer Science Department (1999).

2. L. Badger, D. F. Sterne, D. L. Sherman and K. M. Walker, A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1): 47–83 (1996).

3. M. Bauer, Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148): 13 (2006).

4. A. Berman, V. Bourassa and E. Selberg, TRON: Process-specific file protection for the unix operating system. In *USENIX Technical Conference* (1995).

5. M. Bernaschi, E. Grabrielli and L. Mancini, REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1): 36–61 (February 2002).

6. J. D. Blackstone, Tiny HTTPd. http://tinyhttpd.sourceforge.net.

7. D. Brumley and D. Song, Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (August 2004).

8. C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle and V. Gligor, Subdomain: Parsimonious server security. In *USENIX Systems Administration Conference* (2000).

9. D. E. Denning, A lattice model of secure information flow. *Commun. ACM*, 19(5): 236–243 (1976).

10. R. Dhamija and J. D. Tygar, The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, ACM Press, New York, NY, USA (2005).

11. Distributed.net, Distributed.net home page. http://www.distributed.net.

12. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham and R. Neugebauer, Xen and the art of virtualization. In *Symposium on Operating Systems Principles* (2003).

13. P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek and R. Morris, Labels and event processes in the asbestos operating system. In *Symposium on Operating Systems Principles* (2005).

14. B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back and S. Clawson, Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation* (1996).

15. E. Fred B. Schneider, *Trust in Cyberspace*. National Academy Press (1999).

16. T. Garfinkel, Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Network and Distributed Systems Security Symposium* (February 2003).

17. T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum and D. Boneh, Terra: A virtual machine-based platform for trusted computing. In *Symposium on Operating Systems Principles* (2003).

18. T. Garfinkel, B. Pfaff and M. Rosenblum, Ostia: A delegating architecture for secure system call interposition. In *Symposium on Network and Distributed System Security* (2004).

19. I. Goldberg, D. Wagner, R. Thomas and E. A. Brewer, A secure environment for untrusted helper applications. In *USENIX Security Symposium*, San Jose, CA (1996).

20. S. Ioannidis and S. M. Bellovin., Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop* (February 2000).

21. M. Jones, Interposition agents: Transparently interposing user code at the system interface. In *14th ACM Symposium on Operating Systems Principles*, pages 80–93 (1993).

22. V. L. Kiriansky, Secure execution environment via program shepherding. In *USENIX Security Symposium* (August 2002).

23. M. Laureano, C. Maziero and E. Jamhour, Intrusion detection in virtual machine environments. In *EUROMICRO Conference* (September 2004).

24. Z. Liang, V. Venkatakrishnan and R. Sekar, One-way isolation: An effective approach for realizing safe execution environments. In *ISOC Network and Distributed System Security* (2005).

25. National Security Agency, Security enhanced linux. http://www.nsa.gov/selinux (2005).

26. A. Ott, *Rule Set Based Access Control as proposed in the Generalized Framework for Access Control approach in Linux*. Master's thesis, University of Hamburg (November 1997).

27. N. Provos, Improving host security with system call policies. In *USENIX Security Symposium* (August 2004).

28. N. Provos and M. Friedl, Preventing privilege escalation. In *USENIX Security Symposium* (August 2003).

29. A. Rubin, R. Geer and M. Ranum, *Web Security Sourcebook*. John Wiley and Sons (1997).

30. J. H. Saltzer, Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7): 388–402 (July 1974).

31. J. H. Saltzer and M. D. Schroeder, The protection of information in computer systems. *Proc of IEEE*, 69(9): 1278–1308 (September 1975).

32. R. Sandhu, Lattice-Based Access Control Models. *IEEE Computer*, 26(11): 9–19 (1993).

33. R. Sandhu, E. Coyne, H. Feinstein and C. Youman, Role-based access control models. *IEEE Computer*, 29(2): 38–47 (1996).

34. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen and J. Lepreau, The Flask security architecture: System support for diverse security policies. In *USENIX Security Symposium* (August 1999).

35. J. Steiner, C. Neuman and J. I. Schiller, Kerberos: An authentication service for open network systems. In *USENIX Winter Technical Conference*, pages 191–200 (1988).

36. W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye and D. Anderson, A new major SETI project based on project serendip data and 100,000 personal computers. In *5th International Conference on Bioastronomy* (1997).

37. D. Thain, Identity boxing: A new technique for consistent global identity. In *International Conference for High Performance Computing, Networking, and Storage (Supercomputing)* (November 2005).

38. D. Thain and M. Livny, Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans (September 2003).

39. K. Thompson, Reflections on trusting trust. *Commun. ACM*, 27(8): 761–763 (1984).

40. VMware, Vmware. http://www.vmware.com (2006).

41. X. Wang, D. Feng, X. Lai and H. Yu, Collisions for hash functions md4, md5, haval-128 and ripemd. Cryptology ePrint Archive, Report 2004/199 (2004), `http://eprint.iacr.org/`.

42. A. Whitten and J.D.Tygar, Why johnny can't encrypt. In *USENIX Security Symposium* (August 1999).

43. M. E. Zurko and R. T. Simon, User-centered security. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, pages 27–33, ACM Press, New York, NY, USA (1996).

This document was prepared & typeset with LaTeX 2_ε, and formatted with ND diss2_ε *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*