NetAuth: Supporting User-Based Network Services

Manigandan Radhakrishnan
mani@rites.uic.edu
University of Illinois at Chicago

Jon A. Solworth
solworth@rites.uic.edu
University of Illinois at Chicago

Abstract

In User-Based Network Services (UBNS), the process servicing requests from user U runs under U's ID. This enables (operating system) access controls to tailor service authorization to U. Like privilege separation, UBNS partitions applications into processes in such a way that each process' permission is minimized. However, because UBNS fundamentally affects the structure of an application, it is best performed early in the design process.

UBNS depends on other security mechanisms, most notably authentication and cryptographic protections. These seemingly straightforward needs add considerable complexity to application programming. To avoid this complexity, programmers regularly ignore security issues at the start of program construction. However, after the application is constructed, UBNS is difficult to apply since it would require significant structural changes to the application code.

This paper describes easy-to-use security mechanisms supporting UBNS, and thus significantly reducing the complexity of building UBNS applications. This simplification enables much earlier (and hence more effective) use of UBNS. It focuses the application developer's attention on the key security task in application development, partitioning applications so that least privilege can be effectively applied. It removes vulnerabilities due to poor application implementation or selection of security mechanisms. Finally, it enables significant control to be externally exerted on the application, increasing the ability of system administrators to control, understand, and secure such services.

1 Introduction

Computer networking was designed in a different era, in which computers were kept in locked rooms and communication occurred over leased lines, isolating systems from external attackers. Then, physical security went a long way in ensuring adequate computer security. Today, however, attackers can remotely target a computer system from anywhere in the world over the Internet.

Given that physical separation is no longer an alternative, securing networked applications requires isolation of a different form, including in general:

1. authentication of both users and hosts;
2. protection of communication confidentiality and integrity; and
3. authorization (also known as access controls) using least privilege [32].

The first two tasks are typically provided for within the application, for example by using SSL [12] or Kerberos [38]. The last task is ideally enforced by the Operating System (OS), since then failures in the application (e.g., a buffer overflow) do not bypass authorization.

But (1) and (2) are complicated by Application Program Interfaces (APIs) which are both difficult and tedious to use; for example, in addition to the basic authentication mechanism, it is necessary to communicate information from client to server (perhaps using GSSAPI [24]), interface to PAM [33], and the OS. The application programmer must choose from a large variety of authentication techniques (e.g., password or public-key), and compensate for their weaknesses. Since complexity is the enemy of security, it is especially important to avoid complexity in security critical code. And authentication is often attacked, for example, password dictionary attacks against SSH [1], as well as the implementations of authentication[2].

Consider the dovecot IMAP server. Over 9,000 lines are devoted to (1) and (2), consuming 37% of the IMAP service code (see Section 6 for details). Clearly, this is a large burden on application developers, and as we shall show, unnecessary. In contrast, the partitioning of the application into processes, and their attendant privileges,

is a concern of application programmers since it is fundamental to program structure. The impact of this partitioning includes the number and purposes of processes, the privileges associated with processes, the communication between processes, the organization of data the processes access, the data and operations which must be performed within a process, the sequencing of operations, and the security vulnerabilities. For a general discussions of these issues, see [4].

One important way of partitioning network services is by the remote user U they serve. That is, a server process which receives requests from U runs under U's user ID, so that its "ownership" is visible to, and limited by, external authorization. Although this scheme is widely used, we don't know a term for it, so we shall call it *User-Based Network Services (UBNS)*. UBNS is used, for example, in dovecot, SSH, and qmail. It prevents a user's private data from being commingled with other user's data and provides the basis for OS authorization. The latter enables system administrators to be able to configure secure services easily. Given the many sources of service code—and frequent releases of the services—it is highly desirable to move the security configuration and enforcement outside the service. This minimizes the harm that errant services can do, reduces the need to understand (often poorly documented) application security, enables strong protections independent of service code, is more resilient in the presence of security holes, and vastly increases the effectiveness of validating service security.

Despite the advantages of UBNS, authentication is often performed in a service-specific way or not at all. A prime example is the Apache web server (and most other web servers). In Apache, the users are not visible to the OS. The crucial independent check provided by OS-based authorization is lost. And application developers often avoid service-specific authentication, due to the complexity it engenders. Hence, an application's initial design often forgoes security concerns which then must be retrofitted after the fact [13]. But retrofitting UBNS requires restructuring and re-implementing substantial portions of the application. And since it is difficult to restructure existing applications, the service may never be made into a UBNS.

If UBNS were easier to implement at the application level, it could be integrated from the beginning of system design. Application complexity would be decreased and security would be improved. In this paper, we describe how to radically reduce complexity in UBNS service using *netAuth*—our network authentication and authorization framework. In netAuth, a service requires only 4 lines of code to implement authentication and 0 for encryption and authorization. Hence, netAuth

1. allows authenticated services to be easily integrated and
2. enables requests for the same user to be directed to the same back-end server.

The first is essential to support UBNS. The second makes it easier to re-use per user processes, removing the need for concurrent programming while increasing system efficiency. In addition, these mechanisms enable more modular construction of applications.

We describe NetAuth APIs and the implementations it gives rise to. By making these mechanisms almost entirely transparent, an application developer adds only minimal code to use these mechanisms. We describe sufficient networking interfaces to support UBNS and describe their implementations. These mechanisms are quite simple and thus are easy to use. The protections provided are also considerably stronger than those in most applications. We then describe a port of a UBNS service, dovecot to netAuth, and the substantial savings of code, simplifications to process structure, and reduced attack surface of this port.

The remainder of the paper is organized as follows: Section 2 describes related work. Section 3 describes the overview of our system. We then describe our system in more depth: Section 4 describes how our authentication mechanism can be used to write application. Section 5 describes briefly our implementation and some performance numbers. In Section 6, we describe the experience of porting dovecot to netAuth. Section 7 discusses the security achieved and finally we conclude.

2 Related work

UBNS is not the only way to partition a service into multiple processes. Another complementary way is *privilege separation* [29]—in which an application is partitioned into two processes, one privileged and one unprivileged. For example, the *listening* part of the service which performs generic processing—initialization, waiting for new connections, etc. is often run as `root` (i.e., with administrative privileges) because some actions need these privileges (for example, to read the file containing hashed passwords or to bind to a port). Unfortunately, exploiting a security hole in a root level process fully compromises the computer. By splitting the server into two processes, the exposure of a root level process is minimized. In contrast to UBNS, retrofitting privilege separation is not difficult, and there exists both libraries [20] and compiler techniques [6] to do it. Both UBNS and privilege separation are design strategies to maximize the value of least privilege [32].

`SSH` is a widely used UBNS service [42, 29], but is ill-suited to implement UBNS services—such as mail, calendaring, source control systems, remote file systems—because of the way network services are built.

In the network case, the listening process exists before the connection is made and must at connect time know what user is associated with the service. SSH's port forwarding[3] performs user authentication at the service host—but not at the service—and hence, to the service the users of a host are undifferentiated[4]. As a result, traditional UBNS services use authentication mechanisms such as SSL or passwords and OS mechanisms such as setuid which are awkward to program and may not be secure. In contrast, netAuth both authenticates and authorizes the user on a per service basis, so that the service runs *only* with the permission of the user. Unlike SSH, netAuth provides end-to-end securing from client to service.

Distributed Firewalls [5] (based on Keynote [5]) in contrast to SSH, implements per user authorization for services by adding this semantics to the `connect` and `accept` APIs. While Distributed Firewalls sit in front of the service, and thus are not integrated with the service, Virtual Private Services are integrated and thus can provide UBNS services [16]. In DisCFS [27], an interesting scheme is used to extend the set of users on the fly by adding their public keys; although we have not yet implemented it, we intend to use this mechanism to allow anonymous access (assuming authorization allows it for a service) thus combining the best of authenticated and public services.

Shamon [25, 17] is a distributed access control system which runs on *Virtual Machines (VMs)*. It "knits" together the access control specifications for different systems, and ensures the integrity of the resulting system using TPM and attestation techniques. Its communication, like netAuth, is implemented in IPsec and uses a modified `xinetd` to perform the authorization. Shamon implements a very comprehensive mechanism for authorization (targeted for very tightly integrated systems), in contrast to netAuth's less complete but simpler service-by-service authorization.

We do not describe the authorization part of netAuth in this paper for two reasons. First, there is not sufficient space. Second, the authentication mechanism can be used with *any* authorization model. For example, even POSIX authorization, privilege separation, and VMs could be combined to provide a reasonable base for UBNS. The most value for authorization is gained when privileges are based both on the executable and the user of the process, increasing the value of privilege separation. Such separation is essential to allow multiple privilege separated services to run on the same OS. Examples of such mechanisms include SELinux [34], AppArmor [9], and KernelSec [30]. Janus[15], MAPBox[2], Ostia[14] and systrace[28, 22] are examples of sandboxing mechanisms which attenuate privileges.

SANE/Ethane [8, 7] has a novel method of authorizing traffic in the network. An authorizing controller intercepts traffic and—based on user authentication for that host—determines whether to allow or deny the network flow. This enables errant hosts or routers to be isolated. However, the authentication information available to Ethane using traditional OS mechanisms is coarse grain (it cannot distinguish individual users or applications). Ethane and netAuth are complementary approaches, which could be combined to provide network-based authentication with fine-grained authentication.

Distributed authentication consist of two components: a mechanism to authenticate the remote user and a means to change the ownership of a process. Traditionally, UNIX performs user authentication in a (user space) process and then sets the User ID by calling `setuid`. The process doing `setuid` needs to run as the superuser (administrative mode in Windows) [39]. To reduce the dangers of exploits using such highly privileged processes, Compartmented Mode Workstations divided root privileges into about 30 separate capabilities [3], including a SETUID capability. These capabilities were also adopted by the POSIX 1e draft standard [1], which was widely implemented, including in Linux.

To limit the setuid privileges further, Plan9 uses an even finer grain one-time-use capability [10], which allows a process owned by U_1 to change its owner to U_2. NetAuth takes a further step in narrowing this privilege since it is limited to a particular connection and is non-transferable; but a more important effect is that it is statically declared and thus enhances information assurance whether manually or automatically performed.

The traditional mechanism to provide user authentication in distributed systems is passwords. Such passwords are subject to dictionary and other types of attacks, and are regularly compromised. Even mechanisms like SSL typically use password based authentication for users [12] even though they can support public key encryption.

Kerberos [38] performs encryption using private key cryptography. Kerberos has a single point of failure if the KDC is compromised; private key also means that there is not non-repudiation to prove that the user did authenticate against a server; and requires that the KDC be trusted by both parties. Microsoft Window's primary authentication mechanism is Kerberos.

Plan9 uses a separate (privilege-separated) process called `factotum`, to hold authentication information and verify authentication. The factotum process associated with the server is required to create the change-of-owner capability. But factotum is invoked by the service, and hence can be bypassed allowing unauthenticated users to access the service. Of course, it is in principle possible to examine the source code for the service to determine whether authentication is bypassed, but this

is an error prone process and must be done anew each time an application is modified. NetAuth, enforces authentication and authorization which cannot be bypassed and is easier to analyze.

The OKWS web server [21], built on top of the Asbestos OS [11] does a per user demultiplex, so that each web server process is owned by a single user. This in turn is based on HTTP-based connections, in which there can be multiple connections per user, tied together via cookies. It uses the web-specific mechanism for sharing authentication across multiple connections. OKWS was an inspiration for netAuth, which allows multiple connections from a user to go to the same server. NetAuth works by unambiguously naming the connection so that it works with any TCP/IP connection; and hence is much broader than web-based techniques.

3 System overview

NetAuth is modular, so that the different implementations and algorithms can be used for each of the following three components:

1. **User authentication** is triggered by new network APIs which (a) transparently perform cryptographic (public key) authentication over the network and (b) provide OS-based ownership of processes. Part (b) inherently requires an authorization mechanism which controls the conditions under which the user of a process can be changed.

2. **Encrypted communication between authenticated hosts** ensures that confidentiality and integrity of communications are maintained, and also performs host authentication. This encryption is provided by the system and requires *no* application code.

3. **Authorization** is used to determine if a process can (a) change ownership, (b) authenticate as a client, (c) perform network operations to a given address, and (d) access files (and other OS objects). In UBNS this ideally depends on both the service and the user. Thus, the authentication mechanism essentially labels server processes with the user on whose behalf the service is being performed so that external authorization can be done effectively. It is highly desirable that the authorization system prevent attacks on one service spilling over to other services.

Due to space limitations, this paper focuses on user authentication. Authentication may seem trivial, but it requires significant amount of code in applications, so much so, that this mechanism is justified solely to improve authentication (without also improving authorization). Our server implementation is in the Linux kernel, but our client is user-space code which can be ported to any OS, including proprietary ones.

For encryption we require that hosts be authenticated and that cryptographic protections be set up transparently between hosts. Host authentication is important since if the end computer is owned by an attacker then security is lost. Such end devices can be highly portable devices such as cellphones. (For less important application one can use untrusted hosts.) Encryption can be triggered either in the network stack or by a standalone process. Currently, we are using IPsec [19, 18] for this purpose as recent standards for IPsec have made it significantly more attractive as it allows for one of the hosts to be NATed [40]. But we expect to replace it with a new suite being developed which will be far less complex and faster.

The netAuth API can be used with any authorization model, which would need to control both change of ownership and client authentication, perhaps using simple configuration files [20] as well as networking and file systems to some extent. NetAuth's authorization model controls who may `bind`, `accept`, and `connect` to remote services on a per user basis as well as fine-grain support for the user and services which can access a file. NetAuth's authorization model is fully implemented, and we will describe it in a forthcoming paper.

A central tenet of our design is a clear separation between administration and use of our system. Even when the same person is performing both roles, this separation enables allowed actions to be determined in advance, instead of being interrupted in mid-task with authorization questions (e.g., "do you accept this certificate?"). It also supports a model of dedicated system administrators; further partitioning of the system administration task is possible, for example to allow outsourcing of parts of the policy.

In netAuth, user processes never have access to cryptographic keys and cryptographic keys can only be used in authorized ways. Hence, from the authorization configuration the system administrator can easily determine which users are allowed to use a service and how services can interact with each other.

NetAuth enables successive connections by the same user to be directed to a single process dedicated to that user. We shall see that this has both programming and efficiency advantages. In addition to its uses in traditional network services, it can be used to easily set up back ends on the same system, and thus allow for further opportunities for UBNS.

We next give an overview of network authentication and UBNS mechanisms in netAuth.

Network authentication netAuth enables the owner of a process to be changed upon successful network authentication. Authentication is implemented as follows:

- the server system administrator must enable UBNS change-of-ownership by specifying the *netAuthenticate* privilege for the service.
- the client process requests the OS to create a connection and a time-limited connection-specific digitally signed authenticator[5] [31].
- the server process explicitly requests the OS to perform network authentication. The user authentication is only usable by the designated server process (it is non-transferable).

This mechanism requires that the client-side system administrator enable the client to use netAuth authentication, and the server-side administrator provide the netAuthenticate privilege. As we shall see, application code changes to support authentication are trivial on both client and server sides.

Because public key signatures are used for authentication, the log containing these signed exchanges proves that the client requested user authentication. This property both helps to debug the mechanism and to ensure that even the server administrator cannot fake a user authentication. Lastly, since no passwords are used over the network, this scheme is impervious to password guessing attacks.

UBNS netAuth has a built-in mechanism to support UBNS. All connections to a specified service from user U_i can be served by a single server process p_i unique to that user. For users U_j, for which there does not exist a corresponding process p_j, a listening process p preaccepts (see Section 4) the connection and creates a new process p_j[6]. Figure 1(a) shows two types of queues of unaccepted connections maintained by netAuth (one for new users and the other for users for which there exists a user process).

Per user server processes are created on demand for efficiency and flexibility. Successive connections for U_i will reuse server process p_i. NetAuth can also support other commonly used methods such as pre-forking processes or forking a process per connection.

This mechanism provides a very clean programming model as it is trivial to create back-end services for each user on demand. For example, Figure 1(b) shows a calendar proxy which caches a user's local and remote calendars (and no one else's) and provide feeds to a desk planner, email to calendar appointment program, a reminder system, etc. The reminder mechanism might know where the user is currently located and where the appointment is, so that reminders can be given with suitable lead times. As the user's connections are always to the same process, requests are serialized for that user preventing race condition (and the need to synchronize) and enable easy adding of calendar applications without configuring for security (since the configuration is in the proxy). Such a model also allows different parts of the application to execute on different systems. For example, a user interface component could run on a notebook, and a backend store could run on an always available server.

We next look at the uses of NetAuth in more detail.

4 NetAuth Application Programming Interface

There are several ways to set the owner of a network service: (1) the service can be configured to run as a pseudo user (e.g., apache) with enough privileges to satisfy any request. (2) the service may need user authentication to ensure that it is a valid user (e.g., for mail relay), but all users are treated identically. This service too can be owned by a pseudo user. (3) the service provided depends on the user, who therefore must be authenticated—it is usually appropriate that the service process be owned by its user (i.e., UBNS).

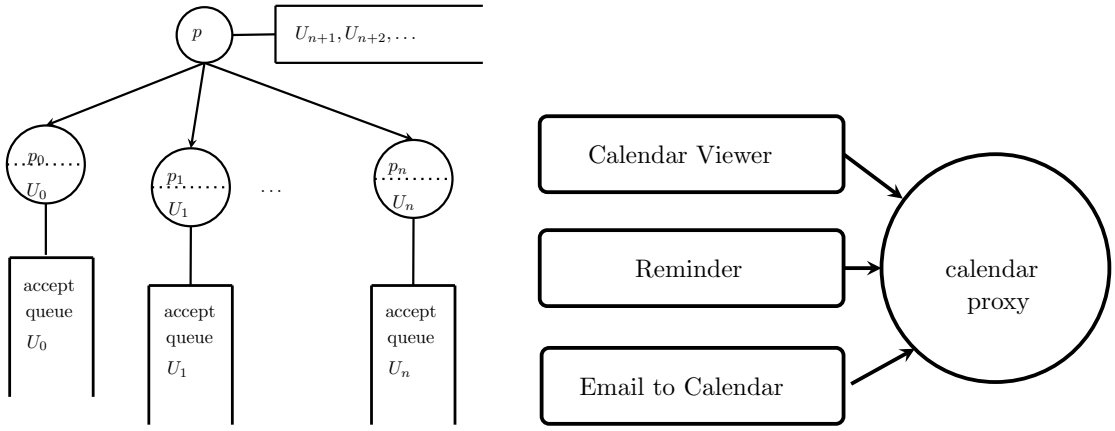
A UBNS service (a process run under the user's ID) performs the following steps: (a) it accepts a connection, (b) performs user authentication to identify the user requesting the service, (c) creates a new process, and (d) changes the ownership of the process to the authenticated user. Once the ownership of the process is changed to the user, it cannot be used by anyone else.

We next examine how this general paradigm is performed in Unix and then in netAuth.

Figure 2(a) shows the call sequence for implementing a user authenticated service using UNIX socket APIs. The client creates a socket (`socket`), connects to the server (`connect`), and then does a series of sends and receives (`send/recv`), and when its done closes the socket.

The server creates a socket (`socket`), associates it with a network address on the server (`bind`), allocates a pending queue of connection requests (`listen`), waits for a new connection request to arrive (`accept`). To perform UBNS, it spawns a process (`fork`), and after determining the user via network messages (not shown) it then changes the owner of the process (`setuid`). At this point the newly created service process is operating as the user. It communicates back and forth with the client and then closes the connection. Since there is typically no way to reuse the process after it closes the socket, it exits.

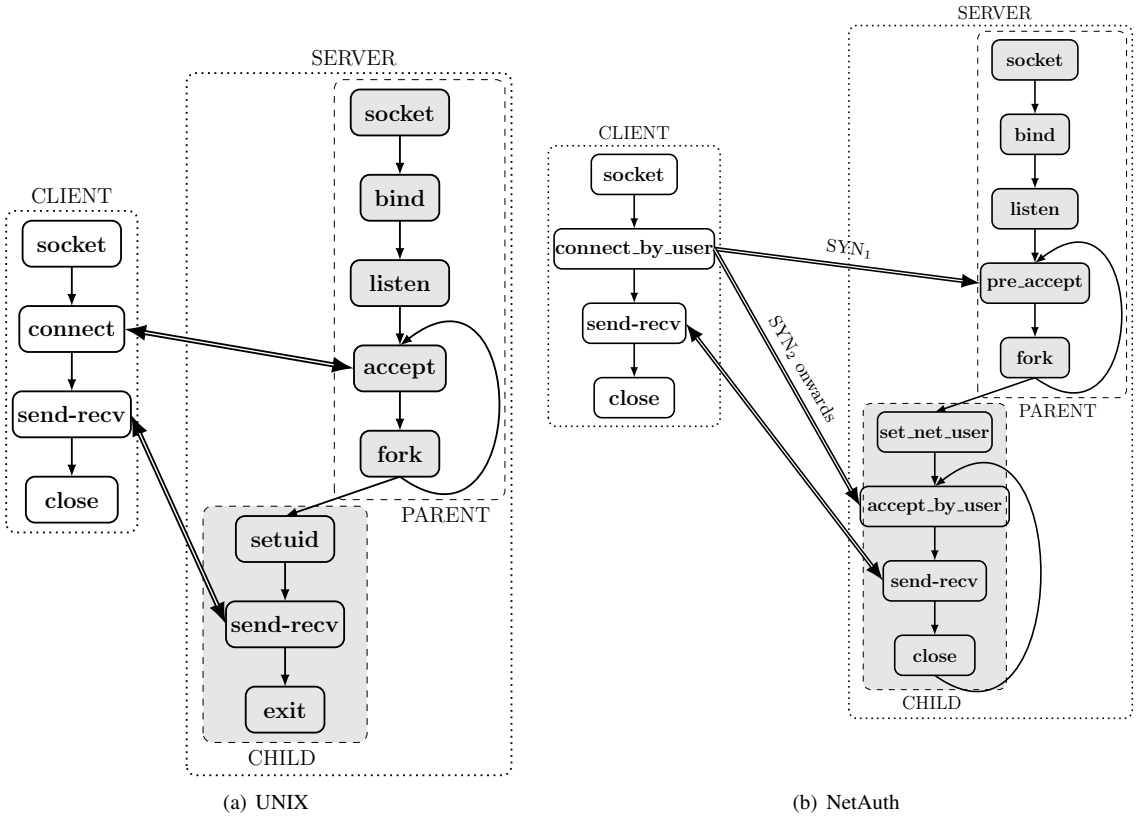
Figure 2(b) shows the equivalent sequencing for ne-



(a) mapping all the connections of a user to the same process

(b) calendar privilege separation example

Figure 1: Privilege separation in netAuth



(a) UNIX

(b) NetAuth

Figure 2: Sequence of system calls executed by a client and a server. The server forks a process to service a request; the forked process is owned by the authenticated user.

tAuth. On the client side, the only programming change needed to adapt to netAuth is to replace `connect` with `connect_by_user` (of course, the application-level authentication must be removed).

On the server side, netAuth basically splits the accept for a new connection into two phases:

- The first phase is called the `pre_accept`, which determines when a new user (one that does not have a service process) arrives. Hence, the `pre_accept` blocks until there is a waiting connection for some user U without a corresponding service process owned by U. (To prevent race conditions, a process which has a temporary reservation for U by virtue of having done a `pre_accept` but not yet having changed the owner is reserved by U.)

- The second phase is the `accept_by_user` to actually accept the connection, after having created a process owned by the new user.

The accept is split into two APIs because there are now two actions (1) determining that there is an unaccepted connection for a new user (so that a new process can be created) and (2) completing the accept by a (child) process owned by the new user. (2) ensures that the accepted socket can be read or written (since the process is owned by the user). Hence, the split accept ensures that the `accept_by_user` only succeeds if the owner of the process is the authenticated user on the connection.

The change of ownership of the process is performed by `set_net_user`. The `set_net_user` changes the owner of the process to the authenticated user and consumes the `netAuthenticate` privilege for that process. Thus, `set_net_user` serves as a highly restricted version of `setuid`, and is far safer to use.

5 Implementation

In this section, we describe the netAuth architecture, the protocol for user authentication, and the implementation. We then describe some performance numbers.

5.1 Architecture

The design of netAuth emphasizes the separation of authentication, authorization, and cryptographic mechanisms away from the application.

The overall architecture is shown in Figure 3. Applications communicate with each other using APIs which emphasize process authentication—the one component of netAuth which must be visible to networked application code. There are two types of communications, both of which flow over an IPsec tunnel between the hosts:

- the application's protocol (or data, for performing its function) and

- the netAuth authentication information.

The authentication information is managed by two netAuth daemons—netAuthClient and netAuthServer—which perform both the public key operations for user authentication and enable the process' change of ownership.

5.2 Authentication protocol

Because IPsec is used for communication, IPsec performs host authentication. This means that the remote service is authenticated, because the service type is determined by port and the IP is verified using IPsec's public key host authentication.

Before application communication is established, user authentication is performed:

netAuthClient signs an authenticator which describes the connection.

netAuthServer receives the authenticator and verifies its signature.

Public-key cryptographic operations can be considerably more expensive than symmetric key algorithms. Fortunately, signing (which is done on the relatively idle client) takes significantly longer than verifying (on a busy server). For example, RSA public key signing times (client) and verification times (server) for 1024 and 2048 bit keys are shown in Table 1[7].

Once the `netAuthClient` has proved that it can sign the authenticator, successive signings prove little (since from the first signing we know that the netAuthClient has the requisite private key). Hence, successive connects for that user employ a quick authentication based on hash chains [23].

We use a separate connection to send our authenticator, rather than the more traditional mechanism of piggybacking authentication on the application connection. This is done both to increase the flexibility of communications and to allow connections to be re-authenticated periodically. Re-authentication determines whether the user's account is still active, and hence a re-authentication failure disables the user's account and stops their processes, something that is difficult to do with other protocols. We re-authenticate using the same hash chain scheme as for successive connects for the same user.

5.3 kernel-based implementation

The first NetAuth implementation has been integrated into the Linux kernel. Our implementation has three key

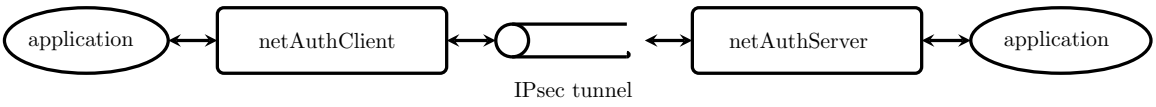


Figure 3: Architecture of netAuth

key size	signing	verifying
1024	680 μs	40 μs
2048	2,780 μs	80 μs

Table 1: RSA signing/verification times in μseconds

components:

- **kernel extensions** (code integrated into the mainline kernel) implementing networking support for processes with per-user privileges and providing the new system calls `pre_accept`, `set_net_user` and `accept_by_user`.
- a **loadable kernel module** implementing netAuth authorizations, uses the Linux Security Module (LSM) framework [41].

  The LSM framework segregates the placement of hooks (scattered through the Linux kernel) from the enforcement of access controls (centralized in an LSM module). Thus changes in the mainline kernel (mostly) do not affect LSM modules.
- Three **user-space daemons** which (1) download the networking policy into the kernel using the `netlink` facility (2) sign authenticators and (3) verify authenticators.

The kernel implementation currently consists of about 3,700 lines of C code ($\sim$3,000 in the kernel module and $\sim$700 in the kernel extensions).

5.4 Performance

We now report on NetAuth's performance. All the experiments were run using a server—an AMD 4200+ (2.2 GHz) machine with 2GB RAM—and a client—an AMD 4600+ (2.2 GHz) machine with 1GB RAM. Both computers ran Linux kernel v2.6.17, used gigabit networking, and were connected by a crossover cable[8]. We measured elapse times (from the applications) in all cases.

We performed two types of performance tests to measure latency. First, we measured the overhead of netAuth authorization and compared it to unmodified Linux, for the cases of the `bind`, `connect` and `connect-send-recv` operations. Second, we measured latency for netAuth's per-user services. For the second part, there is no comparable Linux scenario and hence we report absolute times there.

	UNIX (μs)	NetAuth (μs)	Overhead
`bind`	6.00	6.75	12.5%
`connect`	28.00	32.00	14.28%
`connect-send-recv` (Unix style)	145.00	157.00	8.27%

Table 2: Elapse times for the micro-benchmarks and the Unix-style concurrent server (see Section 3). No authentication is performed in any of these cases. The time specified are all in micro seconds.

5.4.1 System call overhead (no authentication)

Our first measurements determine the authorization overheads for netAuth, by using a light weight authentication with minimal overhead. The authorization mechanism limits which users can use the service, it is implemented outside of application.

The measurements are given in Table 2, are of netAuth vs. unmodified Linux:

- the time to perform a `bind` by a server increased by 12.5% due to the overhead of doing the authorization checks.
- the time to complete a `connect` (as measured) on the client-side increased by 14.28%, due to client-side and server-side authorization checks. The elapsed time includes a round trip packet time.
- the time to do a `connect-send-recv` (as measured) on the client is considered next. (The server must do a `accept-fork-setuid-recv-send`). The send and recv are 128 bytes of data. For the UNIX case, the total time was about 145 μ seconds while for the NetAuth case the time was about 157 μ seconds, an overhead of 8.27%. The most costly operation is the `fork` performed at the server to create a new per-user process.

We note that these overheads are best case [26], normally latency issues are higher. Moreover, no performance tuning has yet been done on the netAuth implementation.

5.4.2 Using netAuth authentication

This section describes the case of a server in which the process for user U_i satisfies *all* of the requests from U_i, for a particular service (as described in Section 5). There does not exist a comparable scenario in UNIX. Hence, we report the latencies observed on the client side in Table 3.

Connection	netAuth (with auth.)	Linux (w/o auth.)
first	4200 μs	147 μs
successive	67 μs	147 μs

Table 3: Elapse times observed on the client side to perform a `connect-send-recv`. The netAuth connections are established with user authentication. Successive netAuth connections are to the same per-user server process created by the first connection on the server. The UNIX connections are established without user authentication.

For the first connection, using netAuth authentication mechanisms, a new connection results in the following set of actions: (1) on the client, the kernel requests an authenticator from the user-space daemon; (2) the client generates the authenticator and sends it to the server where it is verified; (3) there is a RTT for sending the authenticator to the server and receiving response from the server; (4) there may be context-switch times (between client process and authentication daemon); and (5) there may scheduling delays. The costliest operation by far is the cryptographic signing of the authenticator.

All subsequent connections on behalf of the same user run much faster because they re-use the same server process and fast authentications. In comparison, the elapse time for the UNIX case is the same for all cases because there are no schemes for a client to re-use a previously created per-user process. The values for UNIX shown in the Table are without authentication overhead.

5.4.3 Server throughput

We next consider server throughput in terms of new connections. In netAuth, although the first authentication must be signed, successive authentications require only a very fast cryptographic hash. From table 1, the service-based verification of signatures takes only 80μseconds. Hence, a single core can perform authentication for 45,000,000 users per hour assuming authentications are cached for one hour. We believe that such performance levels eliminate the need to consider weaker authentication mechanisms, even for very high volume services.

5.5 Alternative implementations

Our first implementation, which is described here, is a kernel-level implementation. Of course, we would like the APIs described here to be available on other systems without kernel modifications, particularly for those OSes for which the source code is not available.

We consider here only the client side issues as we would like netAuth services to be usable from any operating system. (Server OS, on the other hand, is under the control of the service provider.) In section 6.2 we describe a proxy implementation which uses netAuth, but which could be easily extended into one which implemented netAuth at the protocol level from user space rather than using netAuth APIs.

6 Porting applications to netAuth

To show the effectiveness of netAuth we ported a UBNS service. We have not yet attempted to port a service which is not UBNS organized (such as Apache), as that is a far more difficult problem. We chose an application, dovecot, which supports both privilege separation and UBNS.

Process name	executable name	user ID
`master`	dovecot	root
`auth`	dovecot-auth	root
`login`	imap-login	dovecot
	pop3-login	dovecot
`imap`	imap	U

Table 4: Dovecot processes and their respective user ID's. Here U refers to the user ID of the (remote) user whose is accessing her mail.

Dovecot is an open source IMAP and POP mail server (and is included in Linux distributions such as Debian and Ubuntu). Users can access dovecot-based services remotely using a *Mail Viewer Agent* (`MVA`) such as Thunderbird or Outlook. The `MVA` on the client communicates with dovecot using the IMAP or POP protocols over SSL or unencrypted connections.

Dovecot was built with security as a primary goal. Since January 2006, its developer has offered an as-yet-uncollected reward of 1000€ for the first provable security hole[9]. To support both privilege separation and UBNS, dovecot has four process types, running under root, dovecot pseudo user, and the user U retrieving her mail, as shown in Table 4.

Table 5, shows the code organization of the dovecot distribution supporting IMAP (v1.0.9)[10]. Dovecot also uses pam, crypto, and ssl libraries which are not included in these line counts. The source distribution to support

IMAP is 24,628 lines of code, of which 9,307[11] (37.8%) are associated with authentication and encryption. The port consisted of removing this code, and copying over less than 1,000 lines from `master` (configuration and the concurrent server loop) and `login` (the initial handshake code) to `imap`.

The port reduces the number of process types from four to one. With a traditional Unix authorization model, the port still requires root to bind to port 143 and to do setuid; but unlike the pre-port version, our `imap` process never reads user input while running as root and thus is not subject as root to buffer overflow attack. (The privileges can be still reduced further using netAuth's authorization model).

When implementing a `imap` service from scratch, only 4 netAuth specific lines would be needed to provide authentication and encryption over that required for an unauthenticated service.

6.1 Dovecot before and after

The standard version of dovecot is more complex because of the privilege separation mechanism and especially the complexity of using standard authentication and cryptographic mechanisms. We describe first the processes and then later the operations needed to retrieve IMAP mail in standard dovecot.

6.1.1 Standard dovecot

The dovecot distribution is composed of the following processes:

`master` process starts the `auth` process and n (by default, 3) `login` processes. The `master` process is also responsible for the creation of an `imap` process after a successful authentication.

`auth` process authenticates new users for the `login` process (over a UNIX socket). The `auth` process also verifies successful authentications to the `master` before it creates a `mail` process.

`login` process listens on the appropriate port (e.g., 143 for IMAP) for new connections. Once a connection is established it negotiates with the `MVA` process to initialize the connection (sending server capabilities, setting up SSL, etc.) and requests authentication of the user. Upon successful authentication, the `login` process requests the `master` process to create a new `imap` process and then exits.

`imap` process receives the socket descriptor over a UNIX socket from the `login` process. The `imap` process then communicates with the remote `MVA` to access the user's mailbox on the server.

Figure 4 shows the sequence of events that are necessary to create a new `imap` process to service requests from the `MVA`.

1. Messages 1a and 1b establish the initial connection between the `MVA` and dovecot. During this step, the `MVA` requests and receives the server capabilities (not shown in the figure).

2. authentication step (shown as messages 2a-2e and action 2f). (a) The `MVA` sends the user's authentication information as part of a LOGIN message. (b) In response, the `login` process requests the `auth` process to authenticate the user (c) The `login` process request the `auth` process to authenticate, (d) on successful authentication the `login` process sends a response back to the `MVA` and (e) requests the `master` process to create a new `imap` process. (f) the `master`, after verifying a successful authentication with the `auth` process, creates the new `mail` process running on behalf of user U_k.

3. The `imap` process then services the `MVA`'s future requests.

6.1.2 Porting dovecot to netAuth

The porting of dovecot to netAuth consists of (a) removing code, (b) moving some code into the `imap` process and (c) removing three of the four processes. A dovecot process ported to netAuth is not expected to perform the following functions: message encryption using OpenSSL, GNU-TLS or the like; user authentication; performing the complex `setuid()` operation and related code to ensure that the process does not have any privileged left-overs (in the form of file descriptors) in the unprivileged process. Hence, code for these security sensitive operations need not be implemented by the dovecot executable and can be removed. Thus, summarizing the dovecot port to netAuth:

- the `auth` process (and its code) is eliminated completely as the user authentication is performed by the OS as part of connection establishment.

- from the `master` process, only the code to `bind` to the privileged port and to configure a new mail process with the appropriate set of environment variables is retained. (Dovecot passes configuration information to the `imap` process as environment variables)

- from the `login` process, only the initialization of a new connection (1a and 1b in Figure 4) is retained.

- the core functionality of accessing and maintaining mailboxes in the `imap` process is retained.

directory	total lines of code
master	2,460
auth	5,469
imap-login	484
imap	3,456
lib-auth	490
lib	6,268
login-common	1,138
lib-imap	1,069
lib-settings	101
lib-ntlm	304
lib-sql	882
lib-dict	470
lib-storage	574
lib-mail	1,463
total	24,628

process	dovecot's libraries used	total lines of code	dynamic libraries
master	lib	8,728	
auth	lib, lib-settings, lib-ntlm, lib-sql	13,024	pam crypto
login	lib, login-common, lib-auth, lib-imap	9,449	ssl crypt
imap	lib, lib-dict, lib-mail, lib-imap, lib-storage	13,300	ssl

Table 5: Table with lines of code in the various directories in dovecot. The command '`cat *.c *.h | grep ";" | wc -l`' was used to determine this count.

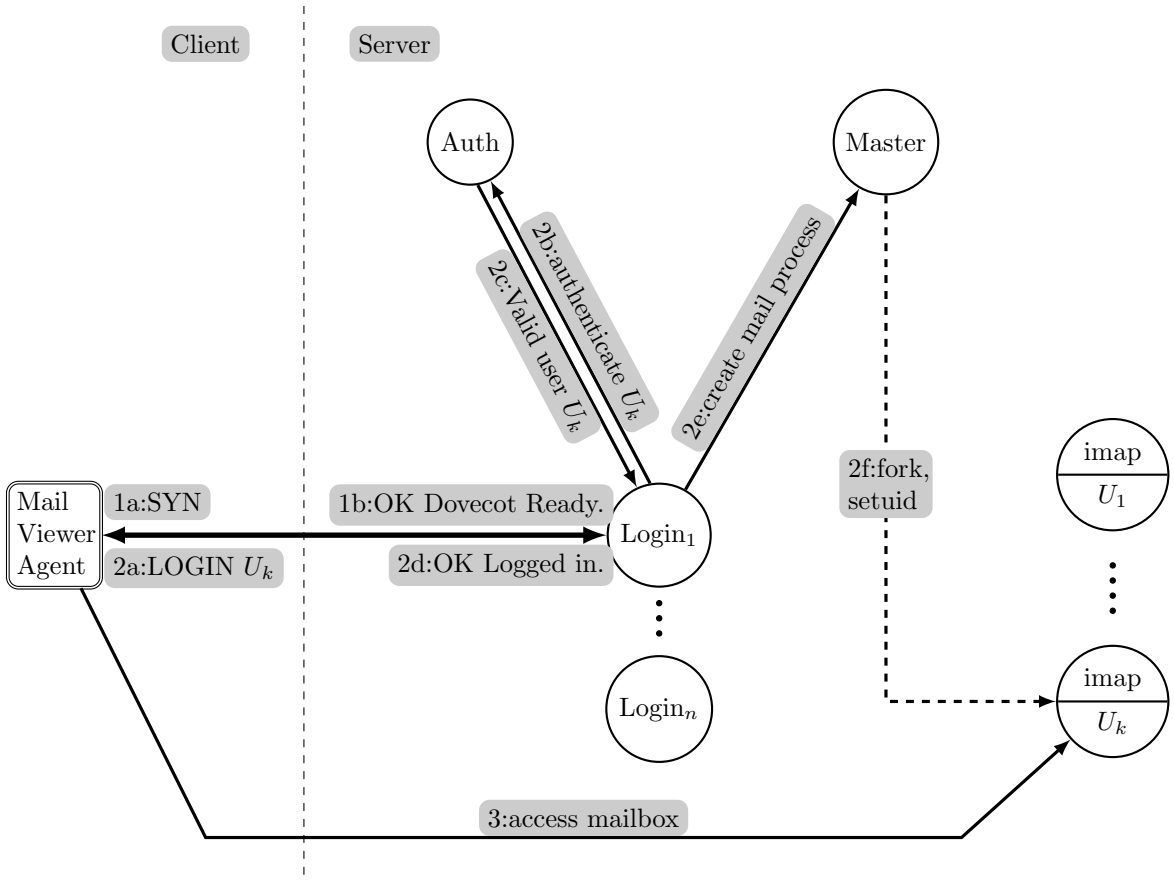


Figure 4: The processes that comprise standard dovecot and their interaction to authenticate a user. Solid arrows indicate message exchange while dashed ones represent process actions. Message exchange across system boundaries use a network socket while those within the same system use UNIX sockets.

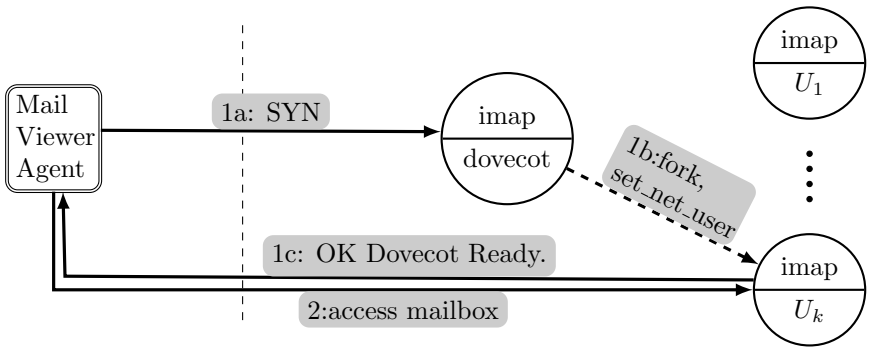


Figure 5: The message exchanges between the ported netAuth dovecot and the MVA.

Thus, the dovecot port to netAuth runs as a single process type (following the design for a concurrent server implementation shown in Figure 2). The master, auth and login processes are eliminated after taking a small amount code from them.

The resulting imap code performs the following steps:

- initializes a socket to listen for new connections. It performs a bind on the privileged port, a listen, sets the accept mode to acceptByUser and blocks on pre_accept waiting for a connection from a user for which there is no imap process.
- when a connection from a new user arrives, the process returns from pre_accept with the new user's information. The process forks a child process to handle the user and returns to waiting for a new user.
- the child process changes the user by executing set_net_user with the user information from the pre_accept call. The child process runs as the new user. This process can now accept the connections (for that user) and process the MVA's requests.

The user is authenticated as part of the processing in the network stack to accept a connection Hence, pre_accept returns only for authenticated users. Connection requests of users that fail to successfully authentication are dropped (with a RST sent back).

6.2 Client side modifications

To test out the server-side modifications it was necessary to produce a netAuth-enabled MVA. Rather than port an existing MVA, such as Thunderbird, we instead built a netAuth proxy. This has several advantages, including portability to systems which do not allow kernel modifications and ability to support a wide variety of MVAs without doing multiple ports. The proxy presented the least invasive approach.

The proxy binds to the IMAP (or POP) port on the localhost. The events to setup a new connection:

- proxy binds to the privileged IMAP port and waits blocking for connection request.
- when a new connection request comes in from the MVA, the proxy authenticates the MVA. Once authenticated, the proxy initiates communication with the dovecot server using the connect_by_user system call.
- Once connected, the proxy just forwards messages to and from the MVA.

Multiple dovecot servers It is not unusual for a user to have multiple mailboxes maintained at more than one server. In this case, the proxy maintains a system-wide mapping (common to all users) from non-routable local IP addresses in the range 127.0.0.0/8 to the well-known routable IP address of the remote host running the dovecot server. All the MVA's on the client are then configured to use IP addresses in this range (published by the proxy) to refer their respective hosts.

The proxy binds and listens for connection requests on all the published local interfaces (i.e., all the 127.0.0.0/8 IP addresses configured for the proxy). A request on a given IP address corresponds to a particular remote host (known to the proxy). The proxy can then follow the scheme outlined above to authenticate the user and establish the connection.

7 Security achieved

The user never has access to his private keys, and in fact needs permission to authenticate using the private keys. This mechanism can be expanded to allow different private keys for different uses, although we do not yet support that. One use of such a facility is to allow the user to

perform personal chores, such as banking with one key and to perform business functions with another key.

Only the specified users can connect to the service, since they must be authorized. This authorization is independent of a service; if the service is designated as an authenticated and authorized service, there is nothing the service can do (either deliberately or accidentally) to evade this mechanisms. The process may avoid actually setting the user ID, but the mechanism pairs user authentication with the connection, so that it can only be used by the process which accepts that connection *and* it is necessary to authenticate before reading or writing to the connection.

Because the authorization and authentication of user services are totally declarative, it is possible to automatically analyze them. (In contrast, this is not possible in general, due to decidability problems, when these functions are performed by application code.) We are planning on extending our previous work in DAC and MAC access controls to automatically analyze authorization properties *across* computer systems [36, 35, 37].

8 Conclusion

UBNS requires a mechanism for (1) authentication of users over the network and (2) allowing server processes to change the user on whose behalf they execute. Implementing the cryptographic mechanisms for user authentication as part of the application is complex and error prone, and as we showed, requires a substantial amount of code. Moving the authentication and cryptographic mechanisms outside the application makes the application independent of these mechanism, and application programmers are usually not skilled in this area. Moreover, the OS mechanisms for change of process ownership are also dangerous as such privileges are among the strongest in a computer system, since changing a user typically allows the privileges of *any* user to be appropriated.

And hence, programmers typically defer such considerations, ignoring them during initial design. But UBNS affects the very structure of programs and when its consideration is delayed, it becomes increasingly expensive to retrofit. Thus many applications will not be structured as UBNS and the design will not satisfy the property of least privilege.

NetAuth is a simple mechanism to invoke network authentication and process change-of-ownership, thus encouraging the design of UBNS. It builds on the work of Kerberos, SSH, and Plan9 but seeks to do so with the style of mandatory access controls and to provide better information assurance. It

- Requires only four lines of code for authenticated and cryptographically protected communications vs. a (concurrent) service which neither authenticates nor encrypts traffic.
- Enables the application developer to focus on the key task of partitioning the application into processes early in the design process.
- Remove the need for privileged processes to receive external input, and thus guards against a range of attacks including buffer overflow.
- Makes application code independent of the authentication method, thus enabling changes in the authentication methods without affecting either source or binary code.
- Externalizes authorization, making it independent of application failures.

While the authentication mechanism and APIs described here can be used with *any* authorization model, we have also built an authorization model (to be described elsewhere) which has a highly analyzable configuration in which strong properties can be understood independently of the application code.

NetAuth integrates public key and a fast reauthentication mechanism to achieve high performance authentications with the strongest possible properties. Further increases in performance are enabled by the reuse of processes for the same user, saving system overhead. This simplifies the structure of such applications, and makes it much easier to build UBNS. Such an easy-to-use mechanism will encourage programmers to integrate security from the start, and thus construct more secure applications.

Not only do these mechanisms enable the construction of more secure services but also provide significant advantages for system administration. These mechanisms enable strong controls to be imposed on services without resorting to application specific configuration and without analyzing application code.

Acknowledgements

The authors would like to thank Jorge Hernandez-Herrero and the anonymous referees for their valuable feedback.

This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

Notes

[1] http://www.securityfocus.com/infocus/1876

[2] http://www.dovecot.org/security.html

[3] Alternatively, SSH allows a remote executable to be invoked, but that remote executable is not connected to as a network service.

[4] hg-login http://www.selenic.com/mercurial/wiki/index.cgi/SharedSSH, as used in Mercurial, performs remote authentication using SSH, but execs a new program rather than connect to a running network service.

[5] We are using a simplified, and easily customized, certificate rather than the complex X.509 certificates.

[6] The application code forks the new process p_j. This explicit structure allows also non-privilege-separated iterative and concurrent service, although these exist largely for legacy applications.

[7] Source http://www.cryptopp.com/benchmarks-amd64.html, for an AMD Opeteron 2.4 GHz processor

[8] The server has an nVidia 570 chipset and the client an nVidia 430 chipset. They both run the open source forcedeth driver.

[9] The webpage at http://www.dovecot.org/security.html displays a list of security holes found in dovecot since the announcement of the award. The dovecot developer (maintainer of the webpage) claims that these holes cannot be exploited under reasonable circumstance stated as a set of rules on the same page.

[10] Dovecot also supports POP, which we ignore for this comparison.

[11] Code from the directories: auth, imap-login, login-common, libauth and master (except the configuration code).

References

[1] 1003.1E, I. D. S. Draft Standard for Information Technology–POSIX Part 1: System API: Protection, Audit and Control Interface, 1997.

[2] ACHARYA, A., AND RAJE, M. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium* (Denver, Colorado, Aug. 2000), USENIX.

[3] BERGER, J. L., PICCIOTTO, J., WOODWARD, J. P. L., AND CUMMINGS, P. T. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering 16*, 6 (1990), 608–618. Special Section on Security and Privacy.

[4] BERNSTEIN, D. J. Some thoughts on security after ten years of qmail 1.0. In *First Computer Security Architecture Workshop* (2007), ACM, p. 1. Invited paper.

[5] BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., AND KEROMYTIS, A. RFC 2704: The KeyNote Trust-Management System Version 2, Sept. 1999.

[6] BRUMLEY, D., AND SONG, D. X. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004), pp. 57–72.

[7] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: taking control of the enterprise. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 2007), J. Murai and K. Cho, Eds., ACM, pp. 1–12.

[8] CASADO, M., GARFINKEL, T., AKELLA, A., FREEDMAN, M., BONEH, D., MCKEOWN, N., AND SHENKER, S. Sane: A protection architecture for enterprise networks. In *Usenix Security* (Oct. 2006).

[9] COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. Subdomain: Parsimonious security server. In *14th Systems Administration Conference (LISA 2000)* (New Orleans, LA, 2000), pp. 355–367.

[10] COX, R., GROSSE, E., PIKE, R., PRESOTTO, D., AND QUINLAN, S. Security in Plan 9. In *Proc. of the USENIX Security Symposium* (2002), pp. 3–16.

[11] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005), 17–30.

[12] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The SSL protocol — version 3.0. Internet Draft, Transport Layer Security Working Group, Nov. 1996.

[13] GANAPATHY, V., JAEGER, T., AND JHA, S. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symposium on Security and Privacy* (2006), pp. 214–229.

[14] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (February 2004).

[15] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. of the USENIX Security Symposium* (San Jose, Ca., 1996).

[16] IOANNIDIS, S., BELLOVIN, S. M., IOANNIDIS, J., KEROMYTIS, A. D., AND SMITH, J. M. Virtual private services: Coordinated policy enforcement for distributed applications. *IJNS 4*, 1 (Jan. 2007). http://www1.cs.columbia.edu/~angelos/Papers/2006/ijns.pdf.

[17] JAEGER, T., BUTLER, K., KING, D. H., HALLYN, S., LATTEN, J., AND ZHANG, X. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks* (Aug. 2006).

[18] KAUFMAN, C. RFC 4306: Internet key exchange (ikev2) protocol, Dec. 2005.

[19] KENT, S., AND SEO, K. RFC 4301: Security architecture for the internet protocol, Dec. 2005.

[20] KILPATRICK, D. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track* (2003), USENIX, pp. 273–284.

[21] KROHN, M. N. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track* (2004), pp. 185–198.

[22] KURCHUK, A., AND KEROMYTIS, A. D. Recursive sandboxes: Extending systrace to empower applications. In *SEC* (2004), pp. 473–488.

[23] LAMPORT, L. Password authentification with insecure communication. *Commun. ACM 24*, 11 (1981), 770–772.

[24] LINN, J. Generic interface to security services. *Computer Communications 17*, 7 (July 1994), 476–482.

[25] MCCUNE, J. M., JAEGER, T., BERGER, S., CÁCERES, R., AND SAILER, R. Shamon: A system for distributed mandatory access control. In *ACSAC* (2006), IEEE Computer Society, pp. 23–32.

[26] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA* (pub-USENIX:adr, 1996), USENIX, Ed., USENIX Conference Proceedings 1996, USENIX, pp. 279–294.

[27] MILTCHEV, S., PREVELAKIS, V., IOANNIDIS, S., IOANNIDIS, J., KEROMYTIS, A. D., AND SMITH, J. M. Secure and flexible global file sharing. In *USENIX Annual Technical Conference, FREENIX Track* (2003), USENIX, pp. 165–178.

[28] PROVOS, N. Improving host security with system call policies. Tech. rep., CITI, University of Michigan, 2002.

[29] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 231–242.

[30] RADHAKRISHNAN, M., AND SOLWORTH, J. A. Application security support in the operating system kernel. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)* (Taipei, Taiwan, Mar. 2006), pp. 201–211.

[31] RIVEST, R., SHAMIR, A., AND ADLEMAN, L. On digital signatures and public key cryptosystems. *Communications of the ACM (CACM) 21* (1978), 120–126.

[32] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer system. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[33] SAMAR, V. Unified login with Pluggable Authentication Modules (PAM). In *Proc. ACM Conference on Computer and Communications Security (CCS)* (1996), C. Neuman, Ed., ACM Press, pp. 1–10.

[34] SMALLEY, S., VANCE, C., AND SALAMON, W. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised April 2002.

[35] SOLWORTH, J. A., AND SLOAN, R. H. Decidable administrative controls based on security properties, 2004. Available at `http://www.rites.uic.edu/~solworth/kernelSec.html`.

[36] SOLWORTH, J. A., AND SLOAN, R. H. A layered design of discretionary access controls with decidable properties. In *Proc. IEEE Symp. Security and Privacy* (2004), pp. 56–67.

[37] SOLWORTH, J. A., AND SLOAN, R. H. Security property-based administrative controls. In *Proc. European Symp. Research in Computer Security (ESORICS)* (2004), vol. 3139 of *Lecture Notes in Computer Science*, Springer, pp. 244–259.

[38] STEINER, J. G., NEUMAN, B. C., AND SCHILLER, J. I. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference* (Dallas, TX, 1988), pp. 191–201.

[39] STEVENS, W. R. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[40] SWANDER, B., HUTTUNEN, A., VOLPE, V., AND DIBURRO, L. RFC 3948: UDP encapsulation of IPsec ESP packets, Jan. 2005.

[41] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General security support for the Linux Kernel. In *Proc. of the USENIX Security Symposium* (San Francisco, Ca., 2002).

[42] YLONEN, T. SSH—secure login connections over the internet. In *Proc. of the USENIX Security Symposium* (San Jose, California, 1996), pp. 37–42.