Abstract

Traditionally, the kernel has provided extremely limited protection of network communications. This is especially nettlesome since network connections are the conduits through which the computer systems are most vulnerable. IPSec is now widely available to protect network connections, encrypting network communications in the kernel.

But cryptography alone is insufficient to implement end-to-end network security. End-to-end security needs the ability to flexibly configure encryption, to authenticate users between hosts, and to reduce the privileges at which network servers need to execute. This paper describes an architecture and kernel-based implementation which addresses these issues.

1 Introduction

Consider end-to-end network security—the security from a process initiating a network connection to the process responding to the connection request. Typically, the initiating process is a client while the responding process is a server: Alternatively, both may be peers. End-to-end security encompasses the entire path from initiating to responding processes. Since security is only as strong as its weakest link, a systematic approach along the complete path is needed.

One critical component of end-to-end network security is cryptography which is used to guard network communications. Cryptographic techniques can provide message validation—ensuring that IP packets are not tampered with or fabricated—preventing spoofing and man-in-the-middle attacks, and encryption—ensuring that the contents of IP packets cannot be read—preventing snooping [KPS02]. Message validation and encryption can be provided by IPSec which, of course, are the Internet Protocol SECurity protocols which provide host-to-host security over unsecured networks [KA98a].

Because cryptography is computationally expensive, it is desirable to be able to control its use: for example, cryptography is not needed on physically secured networks. The decision on when to encrypt traffic typically depends on many factors including: the requesting host, the IP number of the requesting host, the IP number of the responding host, the network service, and the user requesting service. These factors determine whether a connection should be allowed and if so, how the connection must be protected. While IPsec allows these to be factors, it does not specify a mechanism by which these factors can be communicated to the operating system.

Moreover, while encryption is a necessary part of network security, it is insufficient. It is also essential to correctly identify on whose behalf the process is executing, this is called authentication. One way of performing authentication is with passwords for the remote machines, but in practice this is not very secure. It is insecure since it is possible to obtain or just guess them: the better the password the harder it is to remember, hence the more likely it is to be recorded and then captured.

Secondly, we must determine what a remote process is allowed to do, called authorization. Normally servers (eg. web) run as system defined users and

---

1 An alternative term for message validation is message authentication: In this paper we avoid the latter term to avoid confusion with user authentication.

2 Requesting host is different than the IP of host when using DHCP or NAT.

3 There is a related issue of who does the network connection “speak for” [LABW92].
determine authentication and authorizations. This is redundant, as it bypasses the normal kernel-level authorizations for properly authenticated processes.

It is also desirable to reduce the damage done by errors in programming, such as the dreaded buffer overflow problem. The redundancy, in and of itself is harmful since it increases the size of the Trusted Computing Base (TCB)—that part of the computer system which if it fails can effect security [oD85]. This harm is not just theoretical; many server processes run as superuser and small programming flaws can result in malicious takeovers of the computer system. In fact, the vast majority of CERT advisories deal with exactly these types of problems.

In this paper we describe an end-to-end scheme in which the kernel determines the protection to be accorded to network connections, authenticates users over network connections to remote processes without passwords, and reduces the number of root level daemons. The network connections described in this paper are for TCP, although it is possible to extend this work to UDP.

Our scheme is very non-intrusive. It supports legacy network connections, uses IPSec, adds a single new system call and modifies a few network system calls. It may require (some) changes to the coding of processes that use its model. It is especially attractive for new applications since it limits the amount of security code which must be included in such an application by relegating most of the responsibility to the kernel.

The techniques espoused here are for known users on portable or stationary systems to securely communicate over the internet or within an organization. They enable the differentiation of authorization based on location (i.e. IP number) from where the user’s computer connects. They support road warrior and DHCP configurations, increase the effectiveness, and decrease the hassle of remote access. And they simplify the construction and improve the safety of internet servers intended for such users.

This paper is organized as follows: Section 2 describes the basic architecture, and Section 3 describes how network security is specified. Section 4 describes the implementation. Section 5 describes how an ftp daemon is ported to our model. In Section 6 we describe related work, in Section 7 we describe the security afforded, and in Section 8 we present our conclusions.

2 Architecture

In this section, our overall network security architecture is described. Our goal is to integrate IPSec into a kernel-based protection mechanism. This integration enables better control and more exact specification than with IPSec alone. It is similar in spirit to IPSec which integrated cryptographic support, virtual private network (VPN) features, and simple Firewalls together into a single kernel-level entity. Our architecture, like IPSec, supports both enhanced security schemes as well as traditional IP semantics.

We augment the information associated with network connections in three ways:

- The user on whose behalf the connection is made,
- The label of the data in the connection,
- Optionally, permission via delegation certificates for the remote process to assume the identity of the user on whose behalf the connection was established.

The description of our delegation certificates is beyond the scope of this paper. While we shall describe where delegation certificates are used, we will defer further discussion of their structure to a future paper [Sol03].

An alternative to providing kernel-based services is to provide them at the application level (in the network) and the process level (in the operating system). Examples of services at this level include [CGP02, DA99]. These have the advantage that they can be provided without kernel support and are more portable. Unfortunately, when security is implemented at the application/process level, the protections can be bypassed (if not coded in the process’s program), requires code to activate it (in the process), must be individually configured for each program, coding errors can negate the security code, and it’s difficult to understand an application’s security configuration without examining the application (or the application’s documentation).

In contrast, when implemented in the kernel they cannot be bypassed. This enables the system administrators to succinctly and centrally control the security configuration. Thus, we implement our end-to-end network security using kernel support. Several advantages accrue to such an approach:

- Uniform protection of network connections, determining:

4We shall use the term delegation in this paper to refer only to the ability of a process to act on behalf of a user (that is, to change the user ID).
users who are allowed to connect to specific services,

- the hosts from which they are allowed to connect,

- the level of encryption or validation required, and

- the type of information that can flow along them.

• Automatic labeling of network connections by the kernel, in a manner similar to the way files are labeled.

- Sensitive information are properly encrypted,

- Information is only read by authorized users (confidentiality), and

- Information is only created by authorized users (integrity).

With applications/process level protections the control of protection rests with the application. The application must be configured for it, if this configuration is insufficient the application and the computer which hosts it may be subject to attack. Moreover, since each application has its own security configuration, applications must be individually configured and protected. Since errors can exist at the application level from failures to correctly implement security, a long process of hardening is needed to gain confidence in the security of these applications.

This impedes network deployment of resources in two ways: new applications may not be deployed because of the danger they pose to existing services and the inability of system administrators to easily understand the security of their systems.

In contrast, kernel level security works without cooperation from the applications in the following sense: If the application does not interact properly with the kernel security then the application fails to function—it does not open up other services to attack.

This provides system administrators with the ability to strongly control how their systems are protected and reduces the knowledge needed to secure machines (or higher security for the knowledge they have).

3 Network security specification

Our protection mechanism is declarative since this enhances the ability to analyze and understand the security configuration. The base level is intended to be sufficiently flexible to cover most protection needs. Because this level is so simply specified, it is possible either by inspection or with simple tools to analyze the configuration and understand how it is protected enhancing its effectiveness and reducing the likelihood of erroneous specifications.

When the base level is not sufficiently flexible, it is possible to build higher levels on top of it, for example by using trust-based systems such as KeyNote [BIK02]. In that case, the base level limits the cases that the higher level needs to deal with, thus simplifying its analysis. In any event, the analysis of base kernel protections remains intact independently of what is decided at higher levels.

The specification is by a pseudo device tree (/dev/networking) in the filesystem, which describes who can use network connections and how they must be used.

The security features are of four flavors:

- Controlling the user, host, and services which can be used for incoming or outgoing connections,

- Transparently providing for network security to protect the information in transit against reading or modification,

- Authentication and safely changing the ownership of a process, and

- Authorizing access to information which flows on network connections.

3.1 Network Protection and Hierarchical Specification

From the perspective of the host involved in a network connection, a connection has two components:

Incoming connection request is the connection request received by the host. It describes the ability of a local daemon to bind a network address to a socket, listen on the socket for an incoming connection, accept the connection (based on authentication), generate a socket exclusively for the connection that belongs to the incoming connection user and finally read/write on the socket.

Outgoing connection request is the connection request sent to a host. It describes the ability of

\[ \text{It is also possible to use trust based systems to implement the base level.} \]
a process (acting on behalf of an user) to bind a network address to a socket, connect to the other end as an outgoing connection (based on authentication) and finally read/write on the socket.

The security mechanism of incoming requests is independent of that of outgoing requests.

3.2 The /dev/networking directory

The permissions to create or accept a network connection are described in terms of a pseudo-device hierarchy and in terms of a number of group descriptors. These components determine which users can request outgoing connections or accept incoming requests on a service-by-service basis. Moreover, the level of security used for transmission between hosts is also determined.

The contents of the /dev/networking directory are described below:

**networkInterface:** Each network interface has a different security level since multi-homed hosts may be attached to physically secure and insecure networks.

**in/out** subdirectory contains the set of allowed incoming and outgoing network connections respectively.

**security level:** The security level, specifying both encryption and authentication is one of the below choices:

- **legacy:** Traditional IP.
- **authenticate:** Unencrypted but authenticated users. Used only in environments where the network path is physically secure.
- **validate:** Each transmission is cryptographically signed so that the receiver knows that no messages have been inserted, removed, or modified during the transmission. That is, this ensures integrity of the transmission.
- **encrypt:** In addition to providing validation, encryption ensures that no third party can read the message. That is, this ensures confidentiality of the transmission.

**connection group** specifies the name of a set of internet protocol (IP) addresses. The connection group can be described with wildcards, set union, and set difference. Each address specifies the address at the opposite end of the connection.

**aspect group** specifies the name of a set of aspects. Each aspect is a pair [userId, hostId]. This enables differentiation of access privileges based on the computer from which the principal is logged in. Note that the hostId defines the computer, while the connection defines where it is connected to the internet.

**service group** specifies the name of a set of ports (services are associated with ports).

As a result, the network protection mechanism assumes a hierarchical structure under the directory /dev/networking, as shown in Figure 1. The label on the service group component specifies the authorization associated with using that service group as specified below.

Consider a connection from IP address c by aspect a using service s. This is allowed only if there is a path /dev/networking/in/cg/ag/sg in which c ∈ cg, a ∈ ag, and s ∈ sg and where the label on the sg component can be read by the user. (Outgoing is similar.)

The designation of the host is not by connection group (its IP address), since DHCP and road warrior configurations make that infeasible—a single machine may be connected at different times through different IP addresses. Rather the machine is implicitly identified by the aspect, which is then mapped to the operating system public key to set up IPSec tunnels between the machines. Hence the IP number is used to determine security, not to identify host (the exception is legacy).

The network security level (one of legacy/authenticate/validate/encrypt) specifies the minimum security required by the connection. We note that in this model, the hosts on both ends of the connection specify a network security level, and the level used is the maximum of the two specified labels. To see why differences in security levels do not signify a configuration error, consider the case of a notebook computer which will be connected via various networks to its home server—for example, it may be on the same subnet as the server or it may be on a public WiFi connection. The notebook specifies the lowest network security level allowed to communicate with the server. The server specifies different network security levels based on the IP number of the notebook (which is a function of its network connection) and hence raises the security level.
level based on the insecurity level of the network to which the notebook attaches.

**Examples**  In the following, Let $a$ be an aspect, $c$ a connection, and $s$ a service.

- For example, the set of unencrypted TCP connections destined for the trustedIPs are available to all aspects in localUsers for the services listed in userServices. This is written as follows (we don’t show the /dev/networking/eth0/out/authenticate prefix here):

  \[ /\text{trustedIPs}/\text{localUsers}/\text{userServices}. \]

  For $a$ to request $s$ from $c$, then $a' \in \text{localUsers} \land a \leq a', s \in \text{userServices}$ and $c \in \text{trustedIPs}$ and the label on userServices must allow the user to read the information.

- Here is an example of an incoming connection which is allowed on emailServices (including imap, pop, sendmail) from nonLocalIPs which requires encryption. (we don’t show the /dev/networking/eth0/in/encrypt prefix here):

  \[ /\text{nonLocalIPs}/\text{emailUsers}/\text{emailServices} \]

### 3.3 Eliminating root network daemons

Conventionally, a network daemon runs as root at least until it authenticates the user or as some system defined user name. In the latter case, application based authorization of user accesses is required, duplicating functionality at the kernel level thus increasing the complexity of protecting the system.

In the former case, the root process needs to perform the following privileged operations:

- bind to privileged ports.

- authenticate and set the uid for the server process.

Such root daemons pose a serious security threat, since by taking over the root process, as in the Code-Red Worm, the entire system could be compromised. To avoid most root user services, network daemons are run in our model as nullUser. The nullUser has very limited privileges: The ability to bind to a port, perhaps the ability to write to a log, and the ability to switch under controlled circumstances to a non-nullUser\(^7\). Since the nullUser is a synthetic user, the security administrator controls the use of nullUser. The nullUser impact on security is minimal, binding to a privileged port allows at most a

\(^7\)The nullUser is very similar to nobody in Unix with a few capabilities including the rather nonstandard one to do a setnetuid.
denial-of-service to the port and after a `setnetuid` can only do as much damage as the user could do.

![Diagram of a Concurrent Network Daemon](image)

Figure 2: A Concurrent Network Daemon

By moving network authentication into the kernel and providing an indirect way of setting the `user`, network daemons need no longer run as `root` or have powerful `setuid` capability. We initially considered a design in which `accept` set the process’s `user` to that of the network connection, but this means that the server process can no longer function as a network daemon (since it no longer has permissions to do an `accept` since that socket is owned by `nullUser`).

Instead, we introduce a new system call, `setnetuid(int socketFd)`, read as “set network uid”, sets the `uid` of the process to that of the socket `socketFd`. For the `setnetuid` call to succeed, the current user must be `nullUser` and there must be a delegation certificate which allows it. The listening socket is owned by `nullUser` while the socket created with `accept` is labeled with that of the connection. Hence, only the `nullUser` can do `accepts`, while only user’s who can read the label can communicate on the new socket.

Note that there is no password required here. Since both the source’s and destination’s kernels see the password, an untrusted kernel could replay the password. Hence, the kernel must be trusted and the passwords do not provide additional security.

This allows the network daemon to `fork` a new server process and then `setnetuid` in the child process. A failure to invoke the system call `setnetuid` would only prevent the process from accessing the socket and would still remain a `nullUser`. The process flow of a concurrent network daemon is shown in Figure 2. Note that this is significantly more restricted than Unix’s `setuid` since the `uid` it can assume is restricted by our mechanism.

We note that with this scheme a separate process is spawned for each user, rather than share a process with multiple users. This might impose significant overheads for very busy web servers: however, these web servers are likely to be public web servers and hence have little need for the mechanisms espoused here. Web servers that use our mechanisms are likely to be considerably less loaded, and it is also possible to deploy them more widely (spreading out the load) because of their kernel-based protections.

### 3.4 Labeled connections

This authorization mechanism is useful even when the remote process does not change `uid`. It can be used to ensure that:

- sensitive information does not flow unencrypted,
- the server is allowed to read the data, and
- the client is allowed to send the data.

### 4 Implementation

In this section the primary components of the implementation are described. They consist of:

- The primary components needed for our scheme,
- The negotiation process which implements the mechanism for determining network protection and on whose behalf the connection “speaks for”,
- The kernel changes necessary to support this scheme,
- The detailed sequence of system calls and messages necessary to implement this scheme for TCP/IP, and
- Interoperability issues associated with legacy code.

#### 4.1 Primary components

The network structure to support our semantics on TCP consists of four primary components, as shown in Figure 3.

---

8One can argue that they may decrease the window of vulnerability, and so may be beneficial. But the deleterious effects earlier discussed are also present with passwords. So we consider this a wash.
**Tunnels**  Tunnels are created dynamically, as necessary, in the IPSec layer. Cryptographic protection is provided using tunnels between hosts within which individual connections are made. Tunnels allow either encrypted or validated communications. We shall also use, for ease of exposition, the term tunnel to apply to traditional (non-IPSeced) communications. Hence, between each pair of hosts there logically exists 3 tunnels; unencrypted, verified, and encrypted. We note that this scheme is easily extensible to more tunnels—to include for example multiple encryption strengths or encryption and message authentication\(^9\).\(^{10}\)

**Negotiation Process**  The Negotiation Process runs on the initiator and the responding host of a connection and performs the following functions:

- Determines an appropriate security level for an incoming connection based on the proposed minimal security level, the sensitivity of the service requested, the trustworthiness of the source of incoming connection and its userId.

- Produce and process delegation certificates.

- Decide whether to enable a remote process to change the userID.

- Updates the kernel-based connection descriptions, based on its source, userId and connection security level, when performing the requested connection.

---

\(^9\)Message authenticated IPSec tunnels authenticate headers as well as contents, while encrypted IPSec tunnels authenticate only message contents.

\(^{10}\)Message authentication cannot be used on NAT (Network Address Translation) [EF94] systems since they rewrite IP headers.

**Connection table**  The connection table contains an entry for each TCP connection. Each entry in the connection table contains a `connectionSpecifier`. The connection table entries are created by the `negotiationProcess` and is checked at `sendto/recvfrom`.

### 4.2 Determining the security level

In the process of negotiating the security level, the connection initiator provides the negotiation server with the proposed `connectionSpecifier`, which contains the fields shown below:

- **userId**: the user ID on the initiating system.

- **label**: the label of the information flowing over the connection.

- **sourceId**: the public-key that identifies the initiating system uniquely.

---

\(^{11}\)ISAKMP specifies a large amount of different information which can be transmitted, but does not describe the effect of transmitting the information nor does it describe how it is used by upper levels.
- **serviceId**: the port number that provides the service.
- **sourcePort**: port of the initiator.
- **protocol**: this can be either TCP or UDP, but we describe only TCP in this paper
- **sourceInterface**: enables differentiation of interface, for example a wireless from DHCP.
- **sourceIp**: the IP address of the source computer.
- **security**: a lower bound on the security level of the connection.

The negotiation process responds to the security proposal, with an secure *connectionSpecifier* which contains the security level which is the maximum of the initiator proposed security and the responder proposed security (as determined from the /dev/networking tree). Note that the pair, [userId, sourceId] defines the aspect.

For the *uid* to be meaningful across machines, the same *uid* must be assigned on both initiator and responder. A more general scheme to use globally unique user identifiers, as described in SPKI RFC [EFL+99], in which duplicated uids are not possible even on separately administered systems. We have not implemented this because it requires more infrastructure than we currently have, but we intend to integrate this feature in the future.

### 4.3 Kernel changes

The Linux kernel has been changed in the following ways:

- Addition of *setnetuid* system call, to change the owner of a process in a controlled way.
- Adding logic before and after the following system calls:
  - **connect**: to determine the security level and to communicate the authentication information to the responding host.
  - **accept**: to use the connection table to determine connection type and authentication information.
  - **sendto/recvfrom**: These are called directly or indirectly through *send/recv*. Permission to access is checked not on *open* but on *sendto/recvfrom*.

- **read/write**: Permission to access is checked not on *open* but on *read/write*.
- **bind**: allows the binding by nullUser to privileged ports (ports less than 1024).

These changes have no effect on legacy network connections, although legacy connections must be verified with the *negotiationProcess* to ensure that legacy is the correct policy. This logic was provided by a kernel module which intercepted traditional kernel calls and provided the additional functionality.

- Use of a Linux netlink, which enables the *negotiationProcess* to make down calls to the kernel and the kernel to make up calls to the *negotiationProcess*.
- Addition of a *connectionTable* which holds the state of all network connections.
- Addition of ownership (*uid*) to sockets.

### 4.4 TCP

We cover TCP connections in this document. Our scheme also supports UDP, but that part is not yet implemented and hence is the subject for another paper.

We describe the sequence of steps involved in a TCP connection below, which is also shown in Figure 4, the enumerated number is keyed to the numbers in the figure. The concurrent server spawns processes based on incoming connection requests. We do not show the sequence to create the *negotiationConnection*. The connection semantics for TCP has been described below along with Figure 4, as a sequence of system calls along with their respective modifications.

1. **socket** (server): This system call is unmodified.
2. **bind** (server): Allows binding to privileged ports by the nullUser.
3. **listen** (server): This system call is unmodified.
4. **socket** (client): This system call is unmodified.
5. **connect** (client): This system call has been modified to include secure connection semantics. A summary of the events that happens when the client invokes the **connect** system call is shown below:
   
   (a) It binds the socket to an ephemeral port.
Figure 4: Connection Sequence for TCP

(b) constructs and sends the proposed connectionSpecifier to the negotiationProcess. The system call then suspends until after the connectionTable is updated from the negotiationProcess as shown in step 11.

6. proposed connectionSpecifier (client connect): Message sent to the negotiationProcess on the client side, describing every field except the security level.

7. proposed connectionSpecifier (client negotiationProcess): Fills in the security level using the /dev/networking description and forwards proposed connectionSpecifier to the server negotiationProcess. When the server negotiationProcess receives the proposed connectionSpecifier, it determines the appropriate security level needed for the connection and constructs the secure connectionSpecifier.

8. updateConnectionTable (server negotiationProcess): Sends a down link call to update the connectionTable with contents of secure connectionSpecifier.

9. secure connectionSpecifier (server negotiationProcess): Send the secure connectionSpecifier to the client negotiationProcess. This gives the client negotiationProcess the address of the secure tunnel to update the connectionTable on the client side.

10. updateConnectionTable (client negotiationProcess): after the receipt of secure connectionSpecifier, sends a down link call to insert the received secure connectionSpecifier into the connectionTable. This wakes up the client which is blocked in connect.

11. connect wakes up (client connect): The client suspended in the connect
system call is woken up as a consequence of the negotiationProcess’s updateConnectionTable. It looks up its connection in the connectionTable and then connects to the destination over the appropriate tunnel.

12. TCP SYN (client connect): SYN packets sent from the client to server.

13. TCP ACK (server accept): ACK packets sent as a response to SYN packets.


15. accept (server): This system call has been modified to include secure connection semantics. When the server invokes the accept system call, the following sequence of events happens.
   (a) with the receipt of connection request from the client (marked by the exchange of SYN and ACK packets), it determines the IP address and port of the incoming connection.
   (b) checks this information with the sourceIP and sourcePort information in the connectionTable to find an appropriate entry describing the connection.
   (c) if it finds an appropriate connection entry, accepts the connection, else rejects it.
   (d) changes the label of the newly generated socket to that of the label of the incoming connection.

16. fork (server): This system call is unmodified. The network daemon being a concurrent one, invokes this call to spawn a server exclusively for the client.

17. setnetuid(int sockFd) (server): New system call, invoked to set the uid of the process based on the ownership of the socket file descriptor, sockFd passed as a parameter. For the setnetuid to succeed, the negotiationProcess must hold the appropriate delegation certificate (this information is recorded in the connection table).

18. send/recv: These calls can be invoked either by the client or the server depending on the data exchange sequence. These system calls that normally do not have any permission checks on the invoking process have been modified to include them. They check the label of the socket with the uid of the invoking process and to ensure that the uid has permission to read/write the socketFd.

4.5 Interoperability

There are three compatibility/interoperability issues discussed in this section, associated with legacy code. The first is legacy connections, which can be mixed, even in the same server with authenticated and encrypted connections.

Secondly, iterative servers can be run as root if they are to act on behalf of multiple users, since setnetuid is by definition one-way: once called, the process can no longer accept new connection for different users. An alternative is if the server can read the labeled information as an ordinary user, perhaps because it is a member of a group that has read access on that network connection.

Finally, daemons that are started and run with the uid of the client are compatible with authenticated connections, which provides extra protections by checking that network connections can be legally read by the user.

5 Ftp server

To test the new, modified design in a legacy application, we chose to use FTP. We downloaded the FTP daemon source code from WU-FTPD Development Group\(^\text{12}\) and replaced the setuid system call\(^\text{13}\) with the setnetuid system call. The user-password-based authentication procedures have been bypassed in the authenticated case as the new design’s authentication is network-based. To retain compatibility with the legacy model, the user-password-based authentication procedures have been left unmodified and are only bypassed when using authenticated connections.

As a result, the child process invokes the setnetuid system call, immediately after the daemon issues a fork. This is in contrast with the legacy code that invokes the setuid call only after authenticating the user, based on the user-password-based authentication technique. The calls to the DNS server have been left unmodified, using legacy DNS server code. We are still polishing this code, but by publication time we will be able to report on the exact number of lines of code changed to support our model.

The FTP client source code needed no modifications.

\(^\text{12}\)http://www.wu-ftpd.org

\(^\text{13}\)this was setreuid and seteuid in the exact code
6 Related Work

The work was highly influenced by Lampson et al’s work on the “speaks for” relation, in which network connections are said to be making utterances on behalf of users [LABW92]. This logic was used in the Firefly system, and described by Wobber et al [WABL94]. In that system, processes explicitly labeled which network connections “speak for” a given user and processes do not assume the identity of the user.

The use of authentication also draws on SSH [Ylo96]. SSH uses public key encryption to enable users to seamlessly authenticate as remote users. Authorization of SSH is held in system defined files specifying what type of connections are allowed. SSH allows port forwarding, but it is neither as flexible as our technique, does not label network connections, nor does it provide for processes to change ownership.

Another related mechanism to that which we present here is Kerberos [SNS88]. Kerberos is an authentication system with little trust in client hosts: it therefore strictly limits the time that authorizations are outstanding, and requires server support to generate ticket granting tickets from which the client can locally generate authorizations. This mechanism is orthogonal to our purposes here, and could be integrated with our mechanism to place less reliance on the information supplied by client hosts.

A very interesting project, Plan 9, provides extensive password-based security at the application/process level (with kernel support) [CGP+02]. It enables a type of setnetuid, and has an interesting method (a password cache) for containing a kind of capability to use a network service. We are considering implementing a form of the latter, since it allows users—not just system administrators—to control how the system is used. We differ from Plan 9 in the level that our mechanism is provided (network/kernel) vs. (application/process) and hence the degree of control it gives system administrators.

We implemented encryption and message authentication using the IP Security (also called IPSec) Protocols: IPSec [KA98a] includes both message authentication [KA98b] and encryption [KA98c]. We used the FreeS/WAN (www.freeswan.org) IPSec implementation for Linux.

It may seem that some of the mechanisms we propose, in particular the /dev/networking tree can be supplanted with IPSec’s Security Policy Database (SPD) [KA98a]. In fact, the SPD description does say that user (i.e. aspect) and label can be used in determining the Security Association for a socket. Unfortunately, there is no API specified for including this information into the SPD (in fact, the only IPSec API is for PF_KEY [MMP98], a manual keying mechanism)—and FreeS/WAN does not implement this feature. One way of looking at the /dev/networking tree is as an implementation of this SPD functionality. In addition there is no standard way to communicate the “User” of a connection across hosts. In general, the IETF deals with the network layers, so there is always a problem of passing information up to higher levels.

Another related work is the implementation of IPSec policies using KeyNote, a language for describing security configurations [BIK02]. The implementation they reported corresponds roughly to our /dev/networking structure. Their implementation, being language based, is more general in that it can specify rules that we cannot. However, our implementation is declarative meaning that it can be more easily analyzed. A multilevel implementation could combine the strengths of both techniques, and we are considering that for future development.

From an implementation point of view, IPSec/KeyNote is implemented purely at the process level, by modifying the IPSec IKE in openBSD. Since we implement authentication, our technique requires kernel modifications, although we have exported all the policy level up to a process. We decided not to implement this in terms of FreeS/WAN’s IKE process, since FreeS/WAN is undergoing rapid development and our work is too preliminary to include in a FreeS/WAN distribution.

We note that capabilities-based systems, such as POSIX 1e (since withdrawn but parts of which are widely implemented) allow processes to execute with a subset of superuser capabilities [10097]. Hence, it is possible to give processes the right to bind a socket or to execute a setuid. However, the setuid is not restricted as to which user it can change to and hence still is quite dangerous.

Another project whose goal was to isolate authentication from server daemons—and thereby allow them to run at lower privileges is [HM01]. They built a separate process, asp which communicated via ioctl with a user process to enable it to indirectly execute a controlled setuid. Unlike the technique we describe, asp is an authentication server, based on passwords.

We have presented here a mechanism which enables remote processes to assume the ownership of the network connection. In another paper, we describe our delegation and trust model which adds permission to do so, further limiting damage performed by inade-
quately protected (or untrusted) remote hosts [Sol03].

7 Security afforded

In this section we briefly examine the security afforded by our architecture and its weaknesses. All of our security is based on public key cryptography, and hence its security depends on keeping its private key secret. Techniques such as keeping keys in a form which requires some authentication to use may prevent low-threat attacks based on attempting to read these keys by, for example, booting the computer with a debugger. But this is a difficult problem. Alternatively, smart card implementation of public keys may afford greater protection; this is the form used by the US military.

Similarly, we have not considered extensive use of timestamped information, although this too could limit the amount of time that an authorization could be misused.

In what follows, we consider the private keys safe. If they are not, then there is no security and no accountability.

Network Security The network security is very high, assuming a correct implementation, since each side determines the minimum security. Hence, the security determination enables one side to arbitrarily increase security. The reliance on widely used IPSec implementations reduces the risk of implementation bugs. However, if the communication’s security fails it is possible for an attacker to actively and passively attack the network communications.

Network Authentication Network authentication requires cryptographically signed delegation certificates. Moreover, either system administrator (from the client or server) may filter out delegation certificates, further controlling the ability to remotely authenticate. Additionally, system administrators can control from which computers users access their systems. Since using a non-trusted computer inherently eliminates the possibility of security, we believe this is an important and unusual feature. Although the reliance on uids might seem a weakness, they will not match up with the public keys if the uids are not aligned on client and server hosts.

Authorization The authorization information is based on labels. Vanilla Posix is relatively weak at providing mandatory access controls, but these can be provided by projects such as SELinux [LS01] or DTE [BSSW96].

8 Conclusion

Traditionally, the kernel has provided extremely limited protection of network communications. This is an enormous source of problems since network connections are the conduits through which the computer systems are most vulnerable.

In this paper, we consider the problem of end-to-end network security. Building on top of the mechanisms of IPSec, we added in the ability to flexibly configure level of network encryption, to authenticate user, and to reduce the privilege at which server processes run.

The techniques described here are for known users on portable or stationary systems. They enable such users to securely communicate over the internet or within an organization. Furthermore, authorization can be differentiation based on location (i.e. IP number) from where the user’s computer is connecting. Both road warrior and DHCP configurations are supported and the hassle of remote access is decreased. Lastly the construction and the safety of internet servers, intended for such users, is improved.

User authentication is performed without typing in passwords to remote processes. Because the kernels of both the source system (where the user is logged on) and the destination system are already part of the TCB, eliminating passwords does not add in any new vulnerability: the systems cannot be secure if their kernels are insecure.

Finally, the ability to restrict process ownership to ordinary users, rather than as superuser, protects the computer system if these processes are compromised. Since a large number of attacks on system are through superuser owned processes, the techniques presented here close some important operating system vulnerabilities.

References


