Leveraging IPsec for Distributed Authorization

Trent Jaeger* David King* Kevin Butler* Jonathan McCune° Ramón Cáceres†
Serge Hallyn‡ Joy Latten‡ Reiner Sailer† Xiolan Zhang†

* *Department of Computer Science and Engineering, The Pennsylvania State University, University Park PA 16802 USA*

° *Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh PA 15213 USA*

† *IBM T.J. Watson Research Center, Hawthorne NY 10532 USA*

‡ *IBM Linux Technology Center, Hawthorne NY 10532 USA*

Abstract

Mandatory access control (MAC) enforcement is becoming available for commercial environments. For example, Linux 2.6 includes the Linux Security Modules (LSM) framework that enables the enforcement of MAC policies (e.g., Type Enforcement or Multi-Level Security) for individual systems. While this is a start, we envision that MAC enforcement should span multiple machines. The goal is to be able to control interaction between applications on different machines based on MAC policy. In this paper, we describe a recent extension of the LSM framework that enables labeled network communication via IPsec that is now available in mainline Linux as of version 2.6.16. This functionality enables machines to control communication with processes on other machines based on the security label assigned to an IPsec security association. We outline a security architecture based on labeled IPsec to enable distributed MAC authorization. In particular, we examine the construction of a `xinetd` service that uses labeled IPsec to limit client access on Linux 2.6.16 systems. We also discuss the application of labeled IPsec to distributed storage and virtual machine access control.

1 Introduction

Mandatory access control (MAC) enforcement has arrived for commercial operating systems, such as Linux 2.6. With the Linux Security Modules (LSM) framework [22], Linux has a comprehensive reference monitor implementation, and MAC enforcement systems have been built to leverage this infrastructure. For example, RedHat includes the SELinux [14] LSM enabled in its Fedora Core 5 distribution.

However, application of LSM has focused on a single machine at a time. For example, SELinux MAC policy covers 29 different types of objects, but control of network communication has been primitive to date. The SELinux system controls which ports, IP addresses, and network interfaces a process may use, but such controls are notoriously flawed (e.g., IP spoofing) and coarse-grained.

What we want is to be able to control communication, such that we can determine whether two processes on two different machines are permitted to communicate with one another. Two machines X and Y have n and m processes, respectively. A reference monitor (e.g., LSM) on X must be able to determine whether a process X_i can communicate each particular process Y_j on Y in an independent manner based on MAC policy. Further, different application paradigms, such as grid computing, client-server computing, and migrating processes, imply different types of enforcement requirements. For example, grid computing implies that only processes with common labels can communicate, whereas client-server systems often permit the server be more privileged than the clients and be able to run limited processing on behalf of clients.

In order to enable MAC control of network communication, we have implemented a network access control mechanism for LSM [25] based on IPsec [10, 11, 12, 17]. The mechanism's design is motivated by prior work that restricted socket access to IPsec security associations for the Flask system [21]. In that work, a process's access to network communication requires two permissions: (1) a process's permission to use a socket and (2) a socket's permission to use a security association. We have applied this same approach to the Linux kernel using the Linux implementation of IPsec, called the `XFRM` subsystem (pronounced "transform"). We call the mechanism *labeled IPsec*, and our implementation has become available in the mainline kernel as of version 2.6.16. This mechanism goes far beyond the original prototype in that: (1) it is a complete implementation, including IPsec policy management and negotiation; (2) we have experience using it on real applications; and (3) it is part of a widely used system.

In this paper, we examine the application of Linux labeled IPsec to build secure systems. Our primary example the `xinetd` program, a (more) secure version of the old `inetd` program that launches server programs upon

requests at their designated port. The new Linux labeled IPsec mechanism enables control of who can use a machine's `xinetd`, and it enables `xinetd` to start services with limited permissions based on the source of the communication. Also, Linux labeled IPsec can be used to convey labels to third party machines, such that processing can be limited to client access even when the request is forwarded from an intermediate party trusted to make such decisions, not typical of the current `xinetd`.

This paper has the following contributions:

- We detail a mechanism for authorization of network communications based on the labeling of IPsec objects, called *labeled IPsec*. This mechanism is now available in mainline Linux, as of version 2.6.16.
- We develop an architecture for distributed authorization based on labeled IPsec.
- We detail the implementation of this architecture and its application to the `xinetd` service. We also sketch two other applications of this architecture.
- We examine issues related to building secure services using labeled IPsec, such as the granularity of IPsec flows, the impact of client-server asymmetry, and the building of Internet-scale distributed systems.

In Section 2, we describe the problem of building a distributed authorization system for server applications like `xinetd`. In Section 3, we outline the architecture that we envision and describe how it will be achieved. In Section 4, we detail the implementation of the labeled IPsec mechanism that is the basis for the distributed authorization architecture, the additional services necessary to complete the architecture, and describe its application to `xinetd`. We also outline the application of this architecture in other more extensive examples. Section 5 examines some significant issues identified in the course of this work. We summarize our findings and describe future work in Section 6.

2 Problem

In this section, we define the problem that we aim to solve in this paper: process-level access control across machines. We examine related work in solving this problem at the end of the section.

2.1 Example

`inetd` provides a centralized service for starting network daemons on demand. Such daemons use well-known ports, so `inetd` simply needs to maintain passive sockets open on those ports and start the appropriate service based on the port where the packet is received. `inetd` uses a tcp wrapper `tcpd` to actually start the services, so that `hosts.allow` and `hosts.deny` rules can be enforced on the request. Such rules are based on the IP address or hostname of the source of the request.

`xinetd` is a replacement for `inetd` that provides more security function, including full logging, denial of service prevention measures, and access control for other types of services.

The access control in `xinetd` does not distinguish which individual users can access services. This is fundamentally important in multi-level security [2] (MLS) because we do not want services with access to higher-secrecy data to respond to lower-secrecy users (or vice versa). For example, we do not want an `ftp` server to communicate with a lower-secrecy process.

In non-MLS cases, we can also envision cases where limiting access by process rather than by client may be relevant. Consider a corporate service that should only communicate with the certain corporate software running on behalf of employees. In this case, arbitrary malware on the same machine should not be able to communicate with the corporate service.

At present, the distinction between arbitrary malware and client software is hard to make on commercial operating systems (e.g., Windows and Linux). In most cases, the malware and user processes run with the same privilege. However, the availability of mandatory access control (MAC) enforcement in commercial systems, such as Linux, enables separation of important processes from low integrity ones.

2.2 Network Controls

We identify three types of network controls that we want to enable for `xinetd` and other server applications:

- **Authorize Remote Access By Process**: Control which individual remote processes can access `xinetd` and other server processes.
- **Limit Worker Process Access**: Limit the rights that worker processes can obtain based on the remote process for whom it was generated.
- **Convey Limits Remotely**: Enable the server to construct a worker process on another machine whose rights are still limited by those that would be made available to the original remote process.

First, the service (e.g., `xinetd`) must be authorized to receive packets from the remote client and vice versa. Authorization is based on the *label* of the process and the *MAC authorization policy*. The label identifies the permissions in the MAC authorization policy available to the process with that label (e.g., a SELinux subject

type which is analogous to a UNIX UID). For example, an `xinetd` server may run with the label *top-secret* for an MLS authorization policy to prevent it from sending packets to client processes labeled *unclassified*. This also should enable control for non-MLS cases, such as an `xinetd` that runs with a label *corporate_xinetd* such that it can only receive packets from the corporation's users via approved client programs.

Second, `xinetd` creates a worker process for handling the client request. `xinetd` must be able to identify the label of the client peer and create processes with an appropriate label to control the remote client's access. For example, if `xinetd` starts an `ftpd` process, the client should only be able to access files that its label would be authorized for.

Third, it may be desirable to ship the worker process to a third machine (e.g., for load balancing). In addition to the IP address of the client, the service needs to convey the client's security label to the subcontracting service.

2.3 Related Work

An early solution for MAC control of network communications between applications is the IP Security Options [18] (IPSO) which is still used today (e.g., by Trusted Solaris). In IPSO, the MLS sensitivity levels and categories are encoded in the packet header. The receiving system can determine the label of the packet from the IPSO values, and can authorize whether this packet can be delivered to the destination process. We are not aware of a method by which applications may extract this labeling information from the operating system using IPSO.

The original Linux Security Modules (LSM) proposal used IPSO to encode labeling information in packets. The overhead of extracting this information and maintaining labels as packets are fragmented and defragmented added significant overhead to packet processing, even when no security information was specified [22]. This part of the LSM proposal was rejected by the Linux networking subsystem maintainers, so LSMs currently control socket access by restricting the network interface, IP addresses, and ports that sockets may use. Also, `iptables` can control access, but this does not account for the label of the process using a specified port. As a result, `iptables` (and firewalls in general) cannot enforce MAC policies currently.

The Distributed Computing Environment (DCE) defined a mechanism by which remote clients could be authorized. Within an administrative domain, subjects could be identified (e.g., using Kerberos v5), but for other users they are identified as *foreign*. Since Kerberos has largely been limited to a single administrative domain, many users would be foreign. Also, the translation of tickets that maintain client labels between the service and the subcontracting system is not supported.

Trust management enables distributed authorization by including authorization information in credentials [3, 8, 7]. In fact, the KeyNote trust management mechanism has been integrated with IPsec [4]. KeyNote is used in order to manage the creation of security associations between hosts, rather than to enable control of application-to-application communications. Since KeyNote enables packet filtering some control of application-to-application communication is enabled via the packet filtering rules that reference ports, but these rules do not limit the communication based on the application's security label. Any application that can use a port is able to do so in the KeyNote policy.

A previous prototype on the Flask security architecture [6] demonstrated that socket access to IPsec security associations could be used to control packet sends and receives. This work serves as a motivation for our basic approach. This prototype only demonstrated that the approach is feasible. This work did not integrate IPsec labeling into the systems infrastructure necessary to manage such policies, build IPsec security associations (e.g., negotiation), or demonstrate how to use such an approach for distributed authorization. We address those issues in this paper.

3 Distributed Authorization

Figure 1 shows the system architecture that uses labeled IPsec in Linux 2.6.16 to enable distributed authorization, addressing the problems described in Section 2.2. The fundamental concepts are: (1) a verifiable trusted computing base; (2) a consistent authorization policy and enforcement across machines; (3) secure, authorized communication between each pair of machines; and (4) extending services to extract and use these labels to create less privileged worker processes (5).

A trusted computing base on each machine that maintains a consistent mandatory access control (MAC) policy provides the foundation of the system. The trusted computing base of each machine must be verified for dependable enforcement of policy. For example, if the client's trusted computing base is untrusted, then the server's `xinetd` system cannot be certain that the client's processes are labeled correctly. Thus, per process access control is not possible, although the server can limit access at the granularity of the client machine. Having the machines within a common administrative control or verification via remote attestation [16, 20] provides such a guarantee.

Consistent labeling of subjects is necessary, so that a process labeled in a certain way on the client will have a consistent meaning at the `xinetd` server. A process labeled *foo* on the client may be limited to *foo* communication channels, so that `xinetd` can limit the client

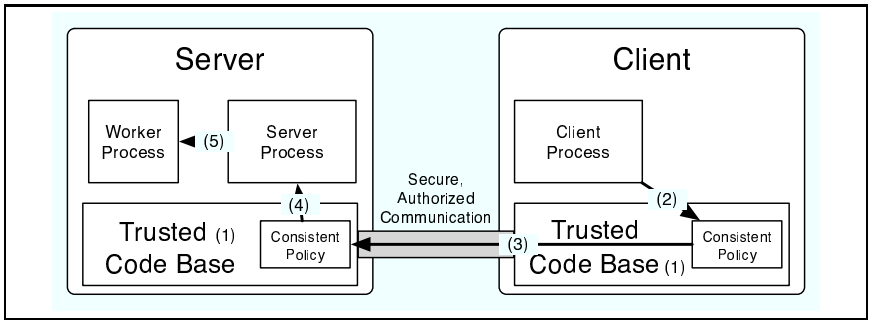

Figure 1: **Distributed Authorization Architecture.** Based on: (1) an attested trusted computing base; (2) consistent labeling of subjects across the two machines; (3) a secure, authorization communication mechanism; (4) a mechanism for obtaining peer security labels; and (5) modifications of servers to use of these labels in creating limited-access worker processes.

process to *foo* accesses on the server and subcontractor. However, the meaning of *foo* must be consistent across the machines. It cannot mean *unclassified* on the client and *top secret* on the server. The same goes for objects. There are two problems: (1) conveying the labeling policy among machines and (2) checking consistency of the policy under dynamic conditions. A variety of policy distribution methods are feasible, but we believe that a remote attestation mechanism is necessary to verify the labeling policy used, particularly given policy changes. We have heard of cases where a centrally administered policy diverges among machines as the system runs.

Communication between machines must be secure and authorized on both ends using the MAC policy. The trusted computing base on each machine must be able to authorize communication based on the security label of the individual processes. For example, we can restrict the access of particular `xinetd` servers to specific clients, such as those that belong to a particular corporation. We use Linux labeled IPsec security associations described in Section 4.1 as the basis for authorizing secure communications. The labels of the processes and their sockets determine whether they can send or receive particular packets.

`xinetd` needs to be able to extract the labels for its clients, so that it can create the appropriately labeled worker processes. As enforcement of the MAC policy for communication is done by the operating system (Linux), it is necessary for the operating system to provide a means for extracting such labels. We extend the kernel-based, labeled IPsec implementation to enable extraction of these labels (see Section 4.2).

Lastly, `xinetd` needs a mechanism by which it can create worker processes that run with the appropriate security label, even if it ships this processing to subcontractor systems. Again, this requires a mechanism between the `xinetd` application and the operating system that enforces access control. For the subcontractor case, this also requires that the server be able to convey the label securely to the subcontractor system. We describe how an application uses a IPsec labels to create a worker process of the appropriate label in Section 4.3.

4 Implementation

We describe the implementation of labeled IPsec (Section 4.1), label extraction (Section 4.2), and the modification of the `xinetd` server program to use such labels (Section 4.3).

This implementation assumes the presence of a remote attestation mechanism to verify the integrity of the trusted computing base. We also envision that this remote attestation would be used to verify the consistency of their MAC policy labeling. For example, we would expect that attestation of the programs that perform labeling and the inputs that they use for generating labels would be sufficient. We discuss the use of remote attestation in the virtual machine system example in Section 4.4.2.

4.1 Labeled IPsec

The implementation of the LSM/SELinux, IPsec, and ipsec-tools extensions that enable packet-level access control based on labeled security associations is shown in Figure 2. For background on these systems, see Appendix A. First, we extend the Linux XFRM subsystem that implements IPsec to label IPsec policies and IPsec security associations, and use LSM/SELinux to authorize the assignment of labels. Second, we extend LSM/SELinux to authorize the selection of IPsec policies based on the label of the sending/receiving socket. Third, we use these IPsec policies to control the generation of labeled IPsec security associations, so they reflect

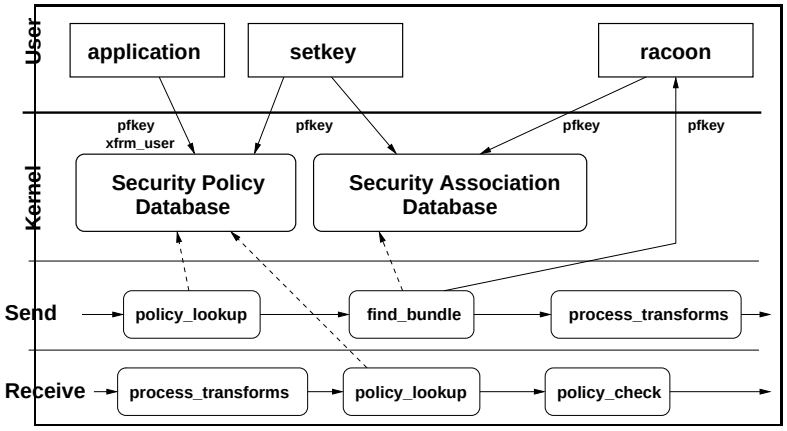

Figure 2: **Labeled IPsec implementation**: (1) Authorized IPsec policy labeling; (2) IPsec policy authorization; and (3) Labeled IPsec security association generation.

the authorized label choice, even in negotiation via the IKE daemon `racoon`.

In the remainder of the section, we discuss the details of the design of these three additions and the resolution of the design issues that were identified.

4.1.1 Adding Access Control Labels

The IPsec policy and the security association data structures in the XFRM subsystem that implements IPsec in Linux are extended by the addition of an access control label, `xfrm_sec_ctx`. This structure has the following fields:

```
domain_of_interpretation
algorithm
SID
context_name
```

The *domain of interpretation* is used by the IKE daemon to identify the domain in which the negotiation takes place. The *algorithm* specifies the LSM for which the label is generated (e.g., SELinux). The *SID* is an integer representation of the label that is interpreted by the LSM. The *context name* is a string representation of the label, also interpreted by the LSM. Storing both an integer and string representation for the label is done to speed authorization, which uses the integer, and dumping contexts on user requests, which uses the string.

When an IPsec policy or security association is input (e.g., via the `pfkey` interface by the `ipsec-tools` program `setkey`), an LSM hook must be added to allocate the label and perform any LSM-specific processing (e.g., computing the SID from the context name). We have one new LSM hook each for IPsec policies and security associations, `xfrm_policy_alloc` and `xfrm_state_alloc`, respectively. Since `xfrm_sec_ctx` is dynamically allocated, it must also be freed, by hooks `xfrm_policy_free` and `xfrm_state_free`.

The authority to label IPsec policy and IPsec security associations must be authorized. Otherwise, any program could change the labels of IPsec objects. For example, it is also possible for applications to set IPsec policies for their own sockets via `setsockopt`. The IKE daemon for ipsec-tools, `racoon`, must set a policy for its socket that handles negotiation, so that it can send packets without IPsec protections to avoid causing a recursive negotiation request. We must ensure that the label chosen for such IPsec policies is permissible. We do not want a low secrecy process to create an IPsec policy that permits it to read high secrecy data. In SELinux, this involves having permissions to `relabelto` the specified label.

4.1.2 Authorizing IPsec Policy

IPsec policy selection is the point at which authorization of network communication is done. In the XFRM subsystem, an IPsec policy that matches the source, destination, protocol, and ports (if specified) is retrieved. In addition, we require that the IPsec policy has a security label that the socket is permitted to use for the operation (i.e., send or receive) based on the MAC policy. An LSM hook is added to authorize any matching IPsec policy, called `xfrm_policy_lookup`. If the IPsec policy is not authorized, then the XFRM subsystem continues to search for another match.

The result is that the IPsec policy selected for a socket is always the first one retrieved that both matches the selection criteria and is authorized. This is consistent with the prior case where the first policy that matched the selection criteria is returned.

The XFRM subsystem actually retrieves IPsec policies in two passes: (1) socket-specific policies added via `setsockopt` and (2) general machine and port policies. Thus, the `xfrm_policy_lookup` LSM hook for authorization must be added in both places.

With the addition of labels, IPsec policies may be used to control access to individual processes. In SELinux, sockets inherit the labels of their process by default. For example, a high secrecy service in an MLS MAC policy may be labeled *high service* and its sockets would inherit the same label. When the socket aims to send a packet to a remote computer, our requirement of a verified trusted computing base (e.g., by remote attestation) enables us to limit communication to only those other computers that we trust to deliver the communication to authorized processes (e.g., other processes labeled *high service*. If the remote TCB meets these requirements, then we can define IPsec policies for communication with that machine labeled for *high communication*. Also, the SELinux MAC policy must permit the *high service* processes to send and receive using the *high communication* IPsec security associations. If the remote machine is truly compatible and trustworthy, it can restrict the delivery of packets only to sockets that can receive packets via the *high communication* IPsec security associations. Since only the sockets running in *high service* processes can receive *high communications* on the remote machine, no lower secrecy applications can intercept the data. We do not actively address covert channels in this design as we see it being outside the scope of hook placement (i.e., based on storage and timing channels in the drivers and protocol).

In addition, we must ensure that packets that are found not to require IPsec processing (i.e., are unlabeled) are only sent when authorized. The SELinux LSM has existing functions for filtering inbound or outbound packets, `sock_rcv_skb` and Netfilter `postroute_last`, respectively. Currently, these functions authorize socket access to ports, IP addresses, and network interfaces. We extend these functions to also authorize non-IPsec packets by evaluating whether the socket can send/receive in an unlabeled manner. Since both IPsec and non-IPsec packets use this hook, we distinguish between them by checking whether the packet has any security associations attached to it. Note that any IPsec communication has already been authorized by `xfrm_policy_lookup`, even if the IPsec policy had no label (i.e., is an unlabeled communication using IPsec).

4.1.3 Using Security Associations

Once an authorized IPsec policy has been selected, we must ensure that the security associations used in the communication have the same access control label as the policy. Since security associations are used differently on inbound and outbound communications, we examine each separately.

For outbound communications, the XFRM subsystem uses the previously cached *bundle* of security associations for the IPsec policy. We ensure that any bundle retrieved matches the access control label of that policy. If none are cached, then security associations may be retrieved individually from the security association database. Once again, we ensure that these security associations have a matching access control labels with the IPsec policy. Finally, if there is no matching security association in the database with the same access control label, then the IKE daemon will be requested to negotiate one. Note that this is the existing behavior of the XFRM subsystem, so no additional code is necessary to trigger the negotiation. Once the negotiation is complete, we check that the security association built by the IKE daemon has an access control label that matches the policy.

The IKE daemon (`racoon` in ipsec-tools) has been modified as well to ensure that the access control label is used in the negotiation. The IKE daemon on the initiator of the negotiation may generate multiple proposal payloads that consist of a set of transform payloads. The idea is that one of the initiator's proposal payloads should match the proposal payload generated by the responder. If so, that proposal is returned to the initiator for acceptance. We must modify the proposal payload generation on both sides to ensure that security associations proposed are built with the appropriate access control labels. On the initiator side, the access control label is extracted from the authorized IPsec policy submitted to the IKE daemon, and it is added as an attribute to the corresponding security association in the proposal payload. On the responder side, the responder is changed to peek at the initiator's proposal to extract the access control labels. There must be an IPsec policy in the initiator's database that matches this proposal and access control label, else the negotiation will fail.

For inbound communications, the XFRM subsystem compares the security associations of the received packets to the authorized IPsec policy for the socket. First, we extend the XFRM subsystem to ensure that only authorized IPsec policies are selected. The same function is used to select policies for both the inbound and outbound direction, so no new LSM hooks are required. Second, we compare security associations and the authorized IPsec policies is based on a runtime identifier, `spi`. When the security associations are built, templates are created and associated with the IPsec policy from which they originated. Thus, the template spi from the authorized IPsec policy and the packet's security association spi must match for the security association to have originated from that IPsec policy. We extend this compar-

ison to also verify that the access control labels match. This may not be strictly necessary since the kernel verifies consistency between security associations and policy on negotiation, but this does ensure correctness of behavior at runtime for low cost.

4.2 Label Extraction

We want to enable applications, such as `xinetd`, to extract the security label for the peer process with which they are communicating. Since the peer process is on a remote machine, we do not see the label of the process, but we do know the security label of the security association that the process is using to communicate. Since the remote process must have been authorized to use security associations with this label, it is indicative of the client's label. We discuss how to use security association labels to identify process labels even more accurately in Section 5. The challenge is that network communication may be implemented using different protocols (TCP or UDP) which have markedly different behavior. In both cases, we have extended SELinux to provide the security label on request. This code is in the process of being upstreamed for Linux.

For TCP sockets, the label of the security association they are using to communicate with a remote peer (e.g., the `xinetd` client) is extracted using the `getsockopt` system call with the `SO_PEERSEC` option. Since TCP is connection-oriented, the Linux kernel can determine whether the socket is currently connected. The kernel caches the security associations used by a socket via its `sk_dst_cache` field (specifically, in the `sock` data structure), so the security label can be retrieved from these cached entries. Note that interpretation of a security label must be done by a LSM; however, an LSM hook to request a peer label in the same manner for UNIX domain sockets (local sockets) already exists. We modified the SELinux implementation to also enable retrieval for TCP sockets.

For connectionless UDP sockets, we do not have a cached connection. In this case, wewant to determine the security label of the peer from the UDP packet that we are receiving. Using the system call `setsockopt` for a UDP socket with the socket options *SOL_IP* and *IP_PASSSEC* tells the kernel to provide the security label in an ancillary message of type *SCM_SECURITY*. Again the LSM (e.g., SELinux) must be invoked to determine the actual label. A new LSM hook is necessary to enable the attachment of the security label to the UDP packet receipt process.

4.3 Worker Process Generation

We extend `xinetd` with a new configuration option called `secsock_adopt` and the supporting code that implements this option. When the `secsock_adopt` option is selected, `xinetd` creates a worker processes for clients based on the security label of the IPsec security association used for the communication.

In this case, when a request is received by `xinetd`, it retrieves the security label for the security association using `getsockopt` as described in the previous subsection (for a TCP connection). As the mechanism uses security labels from LSMs, the implementation must be aware of the particular LSM. In this case, we have used the SELinux LSM. Using the retrieved SELinux security label, we use the SELinux library call `setexeccon` which sets the security label for the next process executed from this parent. So, the worker process will be executed with the retrieved SELinux security label for the security association. We envision that the clients will establish security association using their label, so that the label used will directly identify the label of the client.

A more general version of `xinetd` might launch server programs on one of several machines based on load balancing. If we want `xinetd` or any other server to be able to submit the request to be processed on a remote machine, this can be enabled using labeled IPsec as well. For `xinetd`, it can create a socket to pass the request to the remote machine, and assign this socket the label of the client using the `setsockopt` system call. When labeled IPsec selects an authorized IPsec policy for this socket, only one with the client's label will be chosen (based on the SELinux policy). Of course, the mechanism above requires the reply to return through the first `xinetd`, so alternative networking approaches for load balancing, such as Network Address Translation (NAT), may be preferred. Using UDP encapsulation enable IPsec packets to be used with NAT, so NAT and labeled IPsec can be used together.

4.4 Other Applications

We briefly discuss two systems that use the IPsec controls described here to enforce their access control requirements: (1) a distributed storage system and (2) a distributed virtual machine monitor.

4.4.1 Distributed Storage

To reduce management costs and overhead, many organizations are moving to centralize their storage resources. By employing MAC labeling, we aim to enforce multi-level security without separate infrastructures for each category. Figure 3 shows an example of multiple web servers serving clients with different authorization levels, connected in turn to back-end storage through a proxy. In this example, the client establishes a security association labeled C_x with web server x. When the web server's worker process requests access to the common storage,

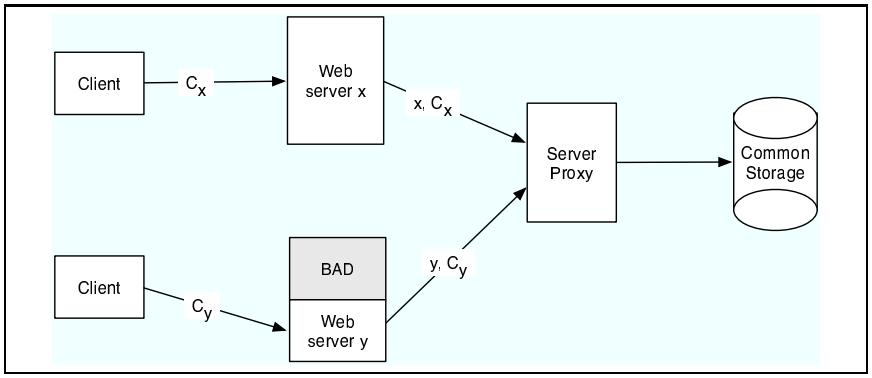

Figure 3: **Distributed processing with centralized storage.**

its security label reflects that it is the web server working on behalf of the client, rather than some more arbitrary client processing as in the `xinetd` case. SELinux enables extended labels, such as type enforcement for the web server label and multi-level security for the client. The storage proxy makes access control decisions based on both the server accessing the storage and the client ultimately making the request (e.g., a client with low security level should not be able to access storage marked as high security). Further, the use of client and web server labels enables the storage proxy to control access based on the combination of server and client, such that a user may have storage access through process x but not through process y. Finally, labeled IPsec enables denial of access at the process level, so the BAD process will have no access to this storage.

4.4.2 Virtual Machine Controls

We have used labeled IPsec as a basis for controlling network communications between virtual machines distributed among remote systems [26]. As shown in Figure 4, a virtual machine trusted to enforce system policy (MAC VM) uses the labeled IPsec mechanism to control communications between untrusted virtual machines. For example, we implemented BOINC [1] servers and clients in VMs such that the BOINC server could only communicate with its clients. The main difference between this implementation and the `xinetd` example is the use of tunnel mode with labeled IPsec. The two MAC VMs use the labels to control communication between untrusted virtual machines. Also, remote attestation was used to verify the integrity of the authorization infrastructure and policy.

5 Discussion

In this section, we discuss a few issues related to building the systems described above.

IPsec Flows Labeled IPsec is based on IPsec, so the granularity of its controls are limited by the granularity of the possible IPsec security associations. Typically, IPsec is used to establish security associations between individual machines although it is possible to establish security associations at the port-level. Machine-level security associations are more coarse-grained than desirable, and in some cases port-level security associations are still insufficient. We can imagine a case where two client processes on the same machine with different security labels contact the same server process. Because the client processes will use unpredictable port numbers, the server has no basis for describing an IPsec policy for its communication with the client based on ports. Therefore, both clients cannot communicate with the server.

We have investigated enabling IPsec flows to be based on labels as well as machines. IPsec negotiation already enables a server to create a security association if it has an authorized policy, so we can have multiple security associations for the same flow differentiated by label. The problem is that when the server receives the packet it only uses the first security association that matches the flow, regardless of the label associated with the received packet. It is straightforward to extract the label from the packet and use this in selecting the corresponding security association, but this requires a conceptual change in IPsec flows that the community is not ready for yet. The use of a virtual machine per client as in Section 4.4.2 would remove the need for finer-grained flows in many cases.

Actual Peer Labels Rather than using the label of the security association as a surrogate for the label of the

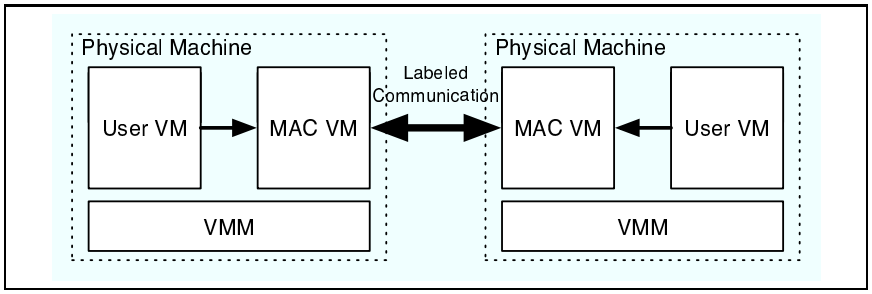

Figure 4: **Virtual machine network controls.**

peer, we are examining whether we can determine the actual label of the peer. Because IPsec security associations are unidirectional, it is possible to have a different label for each direction of the communication. For example, the `xinetd` client packets are sent via a security association that is labeled for the client process, and the `xinetd` packets are sent via a security association labeled for the `xinetd` service. What we have done thus far is to use the client's label for the IPsec security association. Using a label in each direction would enable the client to also ensure that it is talking to a specific server, not just any process that can send a packet to the client.

The current implementation of IPsec negotiation is limited by the `racoon` semantics that result in the creation of a symmetric connection. While this makes some sense for cryptographic processing, it is limiting for authorization. We are investigating a modification to `racoon` to enable the negotiation of different labels for each direction of the IPsec communication which would enable per process identification.

Internet-Scale Systems The next step would be to build a distributed system based on several machines that are connected, where communication is necessary, by labeled IPsec security associations. The basic labeled IPsec mechanism would be the same in such a scenario, but the management problems would be exacerbated by the scale. For example, verifying the trusted computing base and consistency of labeling would challenge scalability. Also, the distribution of IPsec credentials would be much more difficult to manage. We envision that more dynamic management of IPsec policies will be necessary. Dynamic modification of IPsec policies is not often done, but the complimentary work of Yin and Wang [23] shows that dynamic modification of policies is practical and useful. We plan to investigate these issues in future work.

6 Conclusions

In this paper, we describe an architecture for distributed, process-level authorization based on labeled IPsec security associations. The architecture enables the operating system to control communication between processes on machines with compatible trusted computing bases (e.g., verified by remote attestation). Further, the architecture enables applications to work with the security labels to further control processing on behalf of clients.

The foundation for the architecture is the labeled IPsec mechanism we built that is now available in Linux 2.6.16. The implementation extends the Linux kernel, SELinux LSM, and ipsec-tools management programs to use the IPsec security labels in a coherent manner. Labeled IPsec enables SELinux to control Linux network communication by authorizing the selection of IPsec policies based on their security labels entered and negotiated by ipsec-tools. In this paper, we describe how this implementation fits in the overall architecture and demonstrate its use in several example systems.

Acknowledgments

Anonymized for review.

References

[1] David P. Anderson BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the* 5^{th} *IEEE/ACM International Workshop on Grid Computing*, November 2004.

[2] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. MITRE Technical Report MITRE MTR-2997, 1976.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Los Alamitos, CA, USA, Nov. 1996.

[4] M. Blaze, J. Ioannidis, and A. Keromytis. Trust management for IPsec. In *ACM Transactions on Information and System Security (TISSEC)*, 5(2), May 2002.

[5] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8^{th} National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[6] A. Chitturi. Implementing mandatory network security in a policy-flexible system. *Master's thesis, University of Utah*, 1998.

[7] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, Oakland, CA, USA, 2001. IEEE Computer Society.

[8] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, Feb. 2003.

[9] LIDS Organization. LIDS: Linux Intrusion Detection System. `http://www.lids.org`, 2005.

[10] S. Kent and R. Atkinson. Security Architecture for the Internet protocol. *RFC 2401, Internet Engineering Task Force*, 1998.

[11] S. Kent and R. Atkinson. IP authentication header (AH). *RFC 2402, Internet Engineering Task Force*, 1998.

[12] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). *RFC 2401, Internet Engineering Task Force*, 1998.

[13] K. Miyazawa *et al.* IPv6, IPsec, and Mobile IPv6 implementation of Linux. In *Proceedings of the 2003 Ottawa Linux Symposium*, July 2004.

[14] National Security Agency. Security-Enhanced Linux (SELinux). `http://www.nsa.gov/selinux`, 2001.

[15] Novell, Inc. Novell AppArmor, `http://www.novell.com/products/apparmor`, 2006.

[16] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A Trusted Open System. In *Information Security and Privacy: 9th Australasian Conference, ACISP 2004*, Sydney, Australia, July 2004.

[17] D. Piper. The Internet IP Security domain of interpretation. *RFC 2407, Internet Engineering Task Force*, 1998.

[18] M. St. Johns. Draft revised IP Security Option. *RFC 1038, Internet Engineering Task Force*, 1988.

[19] S. Smalley. Configuring the SELinux policy. NAI Labs Report #02-007, available at www.nsa.gov/selinux, June 2002.

[20] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Oct. 2002.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The Flask Security Architecture: System support for diverse security policies. In *Proceedings of the 8^{th} USENIX Security Symposium*, August 1999.

[22] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11^{th} USENIX Security Symposium*, August 2002.

[23] H. Yin and H. Wang. Building application-aware IPsec policy system. In *Proceedings of the 14^{th} USENIX Security Symposium*, August 2005.

[24] H. Yoshifuji, K. Miyazawa, Y. Sekiya, H. Esaki, and J. Murai. Linux IPv6 networking. In *Proceedings of the 2003 Ottawa Linux Symposium*, July 2003.

[25] Anonymous. Anonymized IPsec document. Tech report, 2005.

[26] Anonymous. Anonymized VMM document. Tech report, 2006. In submission.

[27] Anonymous Another Anonymized Document. Conference paper, 2006.

Appendix A: Background

Here we provide background on the Linux Security Modules (LSM) framework, IPsec implementation in Linux (the XFRM subsystem), and ipsec-tools services.

Appendix A.1: Linux Security Modules

The Linux Security Modules (LSM) framework defines a reference monitor interface in the Linux kernel. A *reference monitor interface* defines where authorization decisions are requested of an *authorization module*, which implements the actual authorization mechanism using its access control policy. An access control decision determines whether a particular *subject* (e.g., process, user, etc.) may access a particular *object* (e.g., file, socket, etc.) to perform a specified *operation* (e.g., read, write, etc.). The LSM reference monitor interface is implemented as function pointers that are placed at the location that such authorizations are required. 140 of such function pointers are defined in Linux 2.6.12.

A wide variety of modules have been implemented as reference monitors behind the LSM interface (e.g., [15, 9, 14]). These modules define the policy model and authorization mechanism that will be invoked when the LSM interface is used. In this work, we extend the SELinux module [14] to enable network control. The SELinux module implements the LSM interface comprehensively using an access control policy representing in an extended Type Enforcement policy model [5, 19]. SELinux support 29 different kernel object types with about 10 different operations per type. SELinux is included in the Linux kernel mainline distribution, and it is enabled by default in RedHat's Fedora Core 3.

Using the LSM interface enables control of most objects in a fine-grained manner. For example, the extended attributes of the ext3 filesystem can be used to store labels on individual inodes, so that access to each file may be controlled independently. For most types of kernel objects, fine-grained labeling is possible, but not for network packets. When a network packet is sent or received, the current LSM hooks enable authorization based on the socket and the packet (i.e., `sk_buff`). However, IP packets do not contain a lot of information that is

used for authorization currently. SELinux authorizes network communications as follows: (1) it authorizes the process's access to the socket and (2) it authorizes the socket's access to the network interface used and remote IP address of the packet. For the receiver of a packet, this information is not particularly helpful in authorization. IP addresses may be spoofed, and the source IP address does not indicate the source application. In general, more than the IP address is necessary for two SELinux machines to work together to manage information flow.

Appendix A.2: Linux IPsec

IP Security protocol [10, 11, 12, 17] (IPsec) provides per-packet authenticity and confidentiality guarantees between peer machines communicating over an untrusted network. An *authentication header* (AH) may be used to provide a strong cryptographic checksum for a packet. An *encapsulating security payload* (ESP) encrypts packets to protect the confidentiality of their contents. IPsec is a kernel protocol that is implemented as part of IPv4 and IPv6. The choice of modes and algorithms is determined by an *IPsec policy*. Policy entries may refer to a directed pair of machines and, optionally, the communication ports. The kernel determines if an IPsec policy is present for a particular communication, and if so, it retrieves an *IPsec security association* for the communication compatible with that policy. If no security association has been created, the kernel can initiate a *negotiation* with the remote party which is implemented by a user-level Internet Key Exchange (IKE) daemon which creates a security association if the negotiation is successful.

In Linux 2.6, an IPsec implementation is part of the mainline kernel. IPsec is implemented in the XFRM (pronounced "transform") subsystem which is based on the USAGI prototype [13, 24]. The idea is that prior to transmitting a packet or prior to delivering a packet to higher protocol layers, the packet may be transformed by a specific algorithm.

First, the `pfkey` and `xfrm_user` interfaces are defined which enable IPsec policies and security associations to be input to the kernel. The kernel stores IPsec policies in the Security Policy Database (SPD) and the security associations in the Security Association Database (SAD). Second, when a packet is sent, the kernel determines whether there is an IPsec policy for the endpoint pair. If so, it is used to retrieve an existing transform bundle which is a set of security associations. These correspond to security associations in the traditional IPsec sense. If no bundle can be found, then the kernel tries to *acquire* a bundle using the IKE daemon. If one is generated, then a bundle is created that includes all the security associations for this endpoint pair. Prior to transmitting the packet, the bundle of security associations is applied to transform the packet.

On the receiving end, the security associations in the transform bundle provided are applied first, then the IPsec policy is retrieved to check whether it is consistent with the bundle applied. The IPsec policy is retrieved using the same code as for the transmission case. Then, the retrieved IPsec policy is compared to the security associations in the bundle to verify that each algorithm in the policy is implemented by a security association with the same identifier (i.e., spi).

Appendix A.3: ipsec-tools

We must also consider changes to the user-level daemons that to support authorization via IPsec. We use the *ipsec-tools* programs `setkey` and `racoon` to manage IPsec policies and negotiate security associations, respectively. `setkey` takes policy specifications and generates IPsec policy and security associations objects that it submits to the kernel for entry into the SPD and SAD, respectively. `racoon` accepts negotiation requests from the kernel including an IPsec policy. It then negotiates transform bundles and enters them into the kernel SAD if the negotiation succeeds. Both daemons use the `pfkey` interface to communicate with the kernel. Note that the addition of access control information will require changes to both daemons: (1) to specify access control labels in the `setkey` configuration and (2) to account for access control labels in negotiation in `racoon` and generate security associations with appropriate access control labels.