IMPLEMENTING MANDATORY NETWORK SECURITY IN A POLICY-FLEXIBLE SYSTEM

by

Ajaya Chitturi

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

June 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Ajaya Chitturi

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Gary Lindstrom

John B. Carter

Jay Lepreau

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Ajaya Chitturi in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Gary Lindstrom
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

The use of networks is growing continuously, constantly increasing the vulnerability of the computer systems that use them. Current solutions for network security, such as firewalls, cannot support sophisticated trust relationships with external entities and lack a comprehensive approach to security. Research in security has shown the usefulness of mandatory security mechanisms for supporting sophisticated trust relationships and secure endpoints in addition to secure communication channels. Other efforts at incorporating mandatory security mechanisms into the network stack have a limited notion of access control policies. This work deals with the design and implementation of a more comprehensive and flexible network security architecture that enforces a mandatory access control policy on network-related operations and a mandatory cryptographic policy on network traffic.

The implementation involves modifying the FreeBSD TCP/IP stack within the Flask secure operating system. Access control decisions are made in a policy-flexible manner by consulting a security server and security attributes are interpreted only by the security server. The access control design maps access control requirements to checks made at different layers in the network stack according to the functionality provided by the layer. This approach has several advantages, which include less time spent on illegal packets and the ability to specify policy in a fine-grain manner. Network cryptographic protection is provided using the IPsec protocol for cryptographic support and the ISAKMP protocol for key management.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The widespread use of computers and networks has emphasized the necessity for secure operating systems and network security mechanisms. In general, computer security involves ensuring confidentiality, integrity and availability of information. In the context of computer networks, security threats such as wire tapping and message stream modification exist in addition to other threats present in stand-alone computer systems. Our work, like other similar efforts [1, 2, 3], addresses these problems by encrypting the information sent on the wire and by controlling access to all the network-related operations in a secure manner.

Current commercial solutions for network security, such as firewalls, protect against attacks by denying network services, other than those considered secure, to outsiders. They cannot defend against insider attacks, nor can they support sophisticated trust relationships with external entities.

Security in operating systems has long been a topic of research [1, 4, 5], which has led to the formulation of various security models and policies for operating systems. This research has shown that mandatory security provided by the operating system is essential for supporting secure applications. This thesis presents the design and implementation of a framework for mandatory security in the network stack. Our implementation involves modifying the FreeBSD TCP/IP stack within the Flask secure operating system to include mandatory access control and cryptographic protection.

The access control design involves identifying the different network services and resources granted at different layers in the network stack, associating permissions with them and imposing the corresponding checks at the points where they are granted. Whereas many of the earlier implementations [3, 6] are extremely policy dependent in the way access control is implemented, Flask follows the approach of separating policy from enforcement and all the policy enforcing components communicate with a security server for making all access control decisions. So, our access control implementation within the network stack performs no interpretation of security attributes and depends on the security server for implementing all the policy-specific logic. This design gives the flexibility of being able to impose a new policy by using a different security server which implements the policy. The architecture of Flask is described in detail in Chapter 3. Chapter 6 describes how the implementation deals with security threats in some common scenarios. Although our implementation is limited to the BSD TCP/IP stack, the access control design is quite general and can in principle be used to provide security for other network stacks too.

No changes in the functional semantics of the BSD socket operations are introduced. The implementation provides mandatory security enforcement for all applications in a transparent manner. Security aware applications can specify preferences using the new system calls we introduce.

In addition to requiring cryptographic techniques for protecting information on the wire, network security must deal with the problem of different machines using different security attributes and mechanisms. We also need protocols for exchanging cryptographic information and security attributes as well. Network cryptographic protection is implemented using the IPsec [7, 8, 9] and ISAKMP [10, 11] protocols, which are being proposed as standards by the IETF (Internet Engineering Task Force). This chapter gives a brief introduction to our work and the concepts related to network security.

1.1 Network Security Architecture

In a stand-alone computer system, both the users and the services provided by the computer are physically located on the same system. Thus, the services are invoked and served either by procedure calls or reliable interprocess communication. However in computer networks, remote services are carried out by communication protocols. In addition to the security problems common in a centralized computer, there are interprocess communication threats that can take place while services are being processed by the communication protocols. Communication protocols offer some minimal protection against these threats as part of their reliability features, but they were not designed to counter a hostile attack. In addition, different machines on the network could be employing different mechanisms, policies, and labels for security purposes, and these would have to be integrated with the mechanisms required to enforce a network security policy into a *network security architecture*.

Consider the example scenario shown in Figure 1.1. Nodes A, B, and C are connected by a network. Nodes A and B are trusted and node C is untrusted. Suppose node A does not want to allow any communication with node C. To accomplish this the network subsystem on node A should drop any packet sent by any process on node A to node C as well as any packet received from node C. Consider another scenario under which node A trusts a set of users on node B and wants to communicate with only those users. Supporting this requires the packets coming from node B to be labeled in some fashion to indicate the sending process. The network subsystem on node A can then decide based on the label whether to accept or discard an incoming packet. A label is similar to the Unix user identifier (uid) or group identifier (gid) and is used to specify data having specific access control requirements.

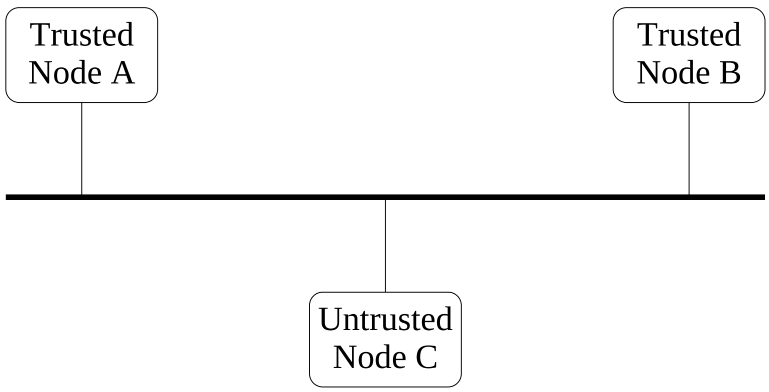

Figure 1.1. Example scenario

Note that since node A depends on the packet label for making the decision, it should also decide whether it can trust node B to deal with data having the corresponding sensitivity in a secure manner.

As the above discussion suggests, the basic requirement for secure communication is to enforce checks on all the network-related operations to ensure that they conform to the policy requirements. It involves extending principles used to build trusted computer systems to the network domain by using a few trusted components in the operating system to ensure that all information flow in the network is authorized. Access control mechanisms are discussed in sections 1.2, 1.4, and 1.5 and in Chapter 4.

Different nodes or administrative domains could use different labels, and each node must be able to interpret the label of incoming data. Resolving this problem involves the notion of a security perimeter where a group of nodes agree on a set of labels; this is described in section 1.6 and section 5.5.

As mentioned earlier, networks present distinct security problems because users must deal with remote entities and send data on the wire and through routers that may not be secure. Cryptography plays a major role in solving these problems. For example, if a customer wants to purchase a product over the internet by providing his credit card information, the node providing the service will first need to authenticate itself to the customer using cryptographic mechanisms. In addition, the credit card information and other sensitive information sent on the wire should be protected using encryption techniques. It is not necessary that all the information sent on the wire should be encrypted, especially since encryption is expensive. For example, web pages providing public information about the company such as the services provided could be sent out without any encryption. Cryptography is dealt with in section 1.3 and in Chapter 5.

Encryption is viewed by many as the salvation of network security and indeed it plays an essential role in protecting information from disclosure or modification from outside sources, but encryption by itself provides very limited protection from attacks within a system because secure communication requires secure endpoints in addition to a secure channel. Similarly, trusted computer system principles are vital to the correct operation of network components involved in solving the network security problem, including devices that employ encryption and control the security measures contained within network protocols, but trusted system concepts by themselves are insufficient to provide network security solutions [12]. Experience with secure networks has led many researchers to believe that three primary elements of network security exist: encryption, protocols and trusted components [13]. The various elements of network security must work together to properly protect information in transit and in storage on a secure network.

So broadly speaking, the network security architecture consists of two major components:

Network access control: This makes use of trusted components to constrain access to the network-related entities in a secure manner.

Network cryptographic protection: This provides reliable authentication of the peers and protection for the information sent on the wire using encryption techniques and protocols.

1.2 Subjects, Objects and Access Control

All activities within a system can be viewed as sequences of operations on **objects** like files, memory, interprocess messages, network packets and I/O devices. Active entities that can access or manipulate objects are called **subjects**. A subject is usually a user or a process acting on behalf of a user. Some entities could serve as both a subject and an object at the same time depending on the observer's view point. **Access Control** involves determining the access rights a subject has on a particular object and enforcing them.

The type of subjects and objects supported by a secure network depends on the layer at which security in the network is to be considered. Network subjects could be users, hosts, nodes, subnets, etc. and network objects could be messages, files, packets and connections. For a detailed discussion refer to the book by Gasser [14].

1.3 Cryptography

As discussed earlier, cryptography plays a key role in providing network security. Encryption algorithms such as RC2, RC4 and DES prevent release of message content that flows on the wire. Cryptographic hash algorithms such as MD2, MD5 and SHA act as a countermeasure to message stream modification. Authentication of principals is achieved using public key algorithms like RSA or sophisticated key exchange algorithms like Diffie-Hellman. Denial of service threats are taken care of to some extent by careful design of the protocols. For example, the cookie in the ISAKMP protocol [10] is aimed at protecting the computing resources from attack without spending excessive CPU resources to determine the authenticity of the packet.

For wide interoperability, protocols for cryptographic support are required. These protocols need to define how the nodes should negotiate the cryptographic mechanisms and also how the kernel can decide the cryptographic operations to be performed on inbound and outbound protocol data units (PDUs). IETF has proposed the IPsec [7] and TLS [15] protocols for providing cryptographic support at the IP and application layers respectively. Protocols like ISAKMP [10] and SKIP [16] are proposed as solutions for key management.

1.4 Covert Channels

Covert channels occur when some mechanism is used in an unexpected manner to provide a means by which information can flow to an unauthorized entity. Whereas overt channels on a secure computer system are prevented by access controls, authentication and other means, covert channels are more difficult to prevent since they occur on mechanisms that were never intended for information flow and may therefore not be protected by existing security mechanisms. An example is the file name space. On a normal UNIX system, a user can look at the name of a file he is not authorized to read and thereby gain some information from the presence of the particular file.

In the context of the network subsystem, the shared port number space creates a similar problem. Some information could be obtained by knowing that a process is bound to a particular port. We prevent this problem by creating a virtualized port number space as described in Chapter 4.

Other covert channels such as those created by analyzing the pattern of nodes with which a particular user is communicating, or the length of the data block being sent, or the time between successive transmissions [17] remain and dealing with them is beyond the scope of this work. Eliminating such channels would require introducing *noise* into the covert channel and could severely degrade network performance.

1.5 Security Policy

A security policy defines the conditions governing actions of and relationships between entities (users, files, network interfaces, etc) within a system. These conditions are usually expressed as a set of rules that define the requirements of the system. Appropriate mechanisms must be present to enforce the desired security policy.

In general, a security policy refers to an access control policy that describes which subjects can perform particular actions on an object. In fact, this is the major component of any security policy. Examples include multilevel security (based on the Bell-LaPadula disclosure model) [13], Clark-Wilson integrity model [18], Biba integrity model [13] and so on. Multilevel security concentrates on protecting confidentiality of information and the other models deal with protecting the integrity of information. Some mechanisms like Type Enforcement [19] and DTE [20] provide a framework for expressing and enforcing security policies. The security policy defines the security attributes associated with the entities on the system. Access control decisions are based on the security attributes of the subject and the object.

Our design also includes a cryptographic policy that specifies the cryptographic measures that need to be adopted based on the security attributes of the data sent on the wire.

1.5.1 Mandatory and Discretionary Policies

A mandatory security policy is defined to be any security policy where the definition of the policy logic and assignment of security attributes are tightly controlled by a system security policy administrator. Mandatory security can implement organization-wide security policies. This is also referred to as nondiscretionary security. A discretionary security policy is defined to be one where the specified mediation is enforced at the discretion of individual users. These mechanisms have user-definable parameters so that users can affect the mediation performed on the system. Discretionary security mechanisms place the burden for security on individual users and since they involve parameters that can be changed easily, they are more susceptible to Trojan horse attacks.

1.6 Security Perimeter

A security perimeter is a collection of nodes adhering to a particular security policy.

A security perimeter could have both trusted and untrusted nodes. Having untrusted nodes within a security perimeter is similar to having untrusted users in a secure computer system - the actions of the untrusted entity will need to be controlled. The security perimeter should be small enough so that the interactions and the policy governing the interactions between entities within the perimeter are manageable and large enough so that restrictions on interactions outside of the security perimeter do not constrain the operational needs of the entities within the perimeter [21].

Communication within a security perimeter involves the same set of security labels and cryptographic mechanisms that are known to all the nodes within the security perimeter. In the case of communication between different security perimeters, the sender and receiver will have different ideas of the meaning of a security context and the properties of the channel itself will have different meanings on each system. This requires a **policy mapper** component to perform translations so that the message can be passed from one security perimeter to another. A gateway between two LANs, for example, could be the only point of intersection between the security perimeters. Two systems in different security perimeters could also use some intermediate policy to talk to each other. Addressing the real problem of policy mapping would probably involve building an infrastructure for routing through security perimeters and is out of the scope of this project.

ISAKMP provides a field called the Domain of Interpretation (DOI) that helps in defining security perimeters. This is described in Chapter 5.

Consider the example shown in Figure 1.1. Consider a scenario in which a trusted file server process is running on node A and a trusted client process is running on node B. We examine how the mechanisms discussed above can be used to deal with the security threats from untrusted processes running on any of the nodes. Access control mechanisms based on a mandatory security policy prevent untrusted processes from reading or modifying files on the trusted file server. The security policies on nodes A and B could prevent any communication with node C. Authentication measures defend the threat of an untrusted user on node B masquerading as a trusted user. Encryption protects the confidentiality of the data on the wire. If an untrusted user on node C or some one who has access to a network component tries to modify the message stream between the trusted file server process and the trusted client process, integrity checks on the packet will fail leading to the packets getting dropped.

The following chapters describe the approach taken for implementing network security in the Flask operating system. Chapter 2 goes through some of the related work in network security. Chapter 3 describes the key concepts of the Flask operating system. Our network security architecture is a part of Flask and makes use of the Flask security server. Chapter 4 describes how security information is bound to the various network-related entities and the decisions involved in enforcing the security policy on these entities. It also describes how the problem of the covert channel due to the shared port number space is solved. Chapter 5 deals with the protocols we are making use of for providing cryptographic protection for the network traffic. It also describes how the cryptographic measures are decided based on the security attributes of the packet. Chapter 6 gives a high-level description of the implementation and presents the performance results obtained from the implementation. Chapter 7 provides conclusions and possible future directions for this work. Appendix A presents a glossary of the security relevant terms and acronyms used in this thesis. Appendix B gives the detailed access control model and how the functions in the network stack are modified to incorporate mandatory access control. Appendix C lists all the system calls, which have been added to allow security aware applications to specify preferences.

CHAPTER 2

RELATED WORK

Past efforts on integrating mandatory access control into the network stack were very policy-specific. The network stack was modified to directly interpret sensitivity labels and apply the access rules according to the policy. These implementations are oriented towards the specific policy they implement. For example MLS/TCP [3] is specific to the MLS policy, whereas the work by Romero et al. [6] is specific to the Bell LaPadula sensitivity model and the Biba integrity model. The DTE work on network security in [22] is specific to the Domain and Type Enforcement policy mechanism. Our design is novel in that it performs no interpretation of the actual security attributes or the policy rules, and can even support dynamic policies to some extent.

The Flask operating system follows the Synergy approach of separating enforcement from policy. So, our work is very closely related to DTOS [1], the prototype built earlier using the Synergy approach. In DTOS, network access control checks occur only at the socket layer, whereas they occur at all layers of the network stack in our model. Our view is that the amount of access control that can be implemented in the socket layer is limited, especially because of the layered nature of the network stack. Our access control model is comprehensive in that it covers different operations at different layers. It includes checks on network entities that range from the network interface to the socket itself. For a detailed discussion of the advantages of our access control model, refer to Chapter 4.

The access control model maps well onto the network architecture and preserves the normal BSD network programming semantics. For example, unlike the substream approach taken by the DTE work on network security [22], in our model, all the packets going out of a TCP connected socket have the same security attributes. We bind security attributes to network entities in the lower layers of the network stack such as network interfaces. This approach is also followed by MLS/TCP [3]. It has the advantage that illegal incoming packets can be detected earlier, thereby reducing the processing time spent on them.

Unlike most of the other efforts, we support extremely fine granularity of access rights. For example the DTE implementation [2] provides only two access rights specific to network access control. Our implementation associates access rights with almost every access control check it implements and provides 33 access rights specific to network access control as shown in Appendix B. Fine-grained access rights are useful both for policy flexibility and for supporting the principle of least privilege. However, there is a possible cost as it could be more complicated to correctly configure a policy when there are many access rights. In cases where a particular access right matches the requirements of the policy, policy specification is much simpler. For example, a client_associate check could be used to easily specify which clients a process is willing to accept.

We know of no earlier efforts, including DTOS, that attempt to solve the problem of the covert channel caused by the sharing of the port number space.

The LOCKed Workstation Expanded Environment Study report [21] gives information about the access control required for Socket IPC in a LOCKed workstation environment. Although the description is specific to the LOCKed workstation environment, the issues and solutions presented are very relevant to our implementation and our access control model is similar to their model. It also provides very useful discussion about the issues that arise when crossing a security perimeter.

The Trusted Computer System Evaluation Criteria (TCSEC) [23] specify the general principles to be applied in any computer system handling sensitive information and specifically detail the features and assurance mechanisms for general purpose host computers. The Trusted Network Interpretation (TNI) [24] of the TCSEC extends these principles to network components in order that they may be applied to specific devices with less functionality than full host computers, but the TNI fails to discuss either encryption or protocols [12]. The TNI in part II attempts to list a variety of security services, but when it tries to apply ratings to them, it manages to effectively illustrate the complexity of applying ratings to individual security features [12].

Some theoretical models [25, 26, 27] proposed earlier describe formal security models for access control in computer networks, but these models are not at the implementation level and are specific to the MLS policy [13].

Current commercial solutions for network security primarily consist of firewalls, encrypting routers, packet filters, etc. Some examples are SCC Sidewinder [28], and TIS Gauntlet [29]. The problem with these solutions is that they are not very flexible and in most cases offer an "all-or-nothing" solution. We allow partial trust based on the node security attributes and implement corresponding access control checks. Recent attempts have been made to make the firewalls more flexible [2, 30]. Whereas the goal of these systems is to make the firewall responsible for the access control, our approach is to modify the operating system and make each machine responsible for controlling the input and output operations that users on the machine can execute. So, our implementation is very useful in preventing insider attacks and supporting sophisticated trust relationships with external entities. Firewalls will still be required to some degree to reduce the burden on individual hosts.

Other solutions like HannaH [31] and SSL libraries [32] provide security functionality in the application layer on top of the network stack without requiring modifications to the operating system. As mentioned earlier, we believe that the amount of access control that can be implemented in such a layer alone is not adequate. In addition, our goal of providing mandatory policy support for access control and cryptography can be achieved only by modifying the network stack within the operating system.

CHAPTER 3

SYNERGY/FLASK APPROACH

The fundamental design principle of Synergy/Flask [33] is to separate enforcement from policy in order to achieve flexibility. The operating system knows when security-relevant decisions are needed and will perform the required enforcement, but relies on a **security server** to make the actual decision. The primary role of the security server is to isolate the rest of the system from the security policy by making all mandatory security decisions. The interface to the security server is the same regardless of security policy and changing the policy or the implementation of the security databases will have no effect on the rest of the system. As shown in the Figure 3.1, the microkernel and other entities which need to enforce policy such as the file server and the network server, communicate with the security server for making access control decisions.

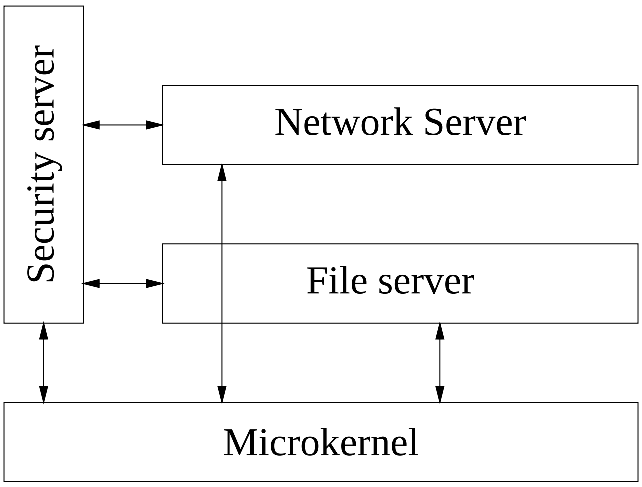

Figure 3.1. Synergy architecture

The data that the security server needs to make a decision are generally in the form of a **security context**, a data structure containing all of the security information that must be bound to an entity. For example this could include the sensitivity level and user identification. Security contexts are treated as opaque blocks of data by tasks other than the security server including the microkernel. From outside of the security server, a security context is referenced by an opaque handle called **security identifier (SID)**. A security identifier (SID) is bound to each entity controlled by the mandatory access control policy. Security identifiers are nonpersistent and are meaningful only on the local node. The security server provides functionality for converting a SID to its corresponding security context or a security context to its corresponding SID. All the entities for which security is being enforced are divided into **security classes**. A security class is a distinct type of object with a distinct set of legal operations, for example, a file, directory, network interface or a node.

When a security decision must be made, the microkernel (or any of the servers) passes the SIDs and the security classes of the subject and the object to the security server which computes the allowable **access vector** for the security contexts and classes associated with the SIDs. For efficiency, policy enforcers cache results obtained from the security server. To support adaptive policies, information about which permissions may be cached and when the access vector's validity will expire is also returned. In addition, the policy enforcers can register a notification port for cache flushes.

The prototype security server implements an access control policy that is a combination of multilevel security (MLS) and Type Enforcement.

CHAPTER 4

NETWORK ACCESS CONTROL

Network access control deals with preventing all the unauthorized network-related operations. It also needs to control covert channels, which could pass information through indirect channels in a way that violates the security policy being enforced. This chapter describes how security information is bound to the various network-related entities and the decisions involved in enforcing the security policy on these entities.

The access control design is layered to match the network architecture. The design involves identifying the various network services and resources granted at different layers in the network stack, associating permissions with them and imposing the corresponding checks at points where they are granted. Associating different permissions with different services also makes the job of specifying the access control policy easier. For example, we could easily specify which sockets could act as a client or a server to a particular socket using the client_associate and server_associate permissions respectively. The access control design is discussed in detail in [34].

In addition to the normal BSD socket calls, a few system calls (all of which end in `_secure`) are added for additional functionality. These calls are required for specifying the security contexts for an operation or for obtaining the security contexts of the peer. Some of these calls are described in the discussion below and a complete description is given in Appendix C.

4.1 Policy Enforcement

Flask provides the network access control policy to the security server, which is consulted by the network access control module to make access control checks.

Due to the layered architecture of the network server, unlike the file server, access control decisions must be made at different levels in the network stack because all the information needed to make the decision is not available at any single point. So, in most cases, the access control decision is split into a collection of checks made at different layers and on different nodes.

For example, when a connection is initiated by a process, the client node does not know what the security context of the recipient process will be, so it can only perform access control checks based on the security attributes of the objects that will be used to send the packet, such as the network interface, and the destination node address. The server node can perform access control checks based on the security attributes of the source process, the source node, the network interface on which the packet was received, and the recipient process. When a process attempts to accept a connection or receive a packet, if the policy prohibits the recipient process from receiving data from the source process or source node, the packet or connection is silently dropped.

A detailed list of all the permission checks that are made is presented in Appendix B and in detail in the Flask network access control design document [34].

4.1.1 Enforcement Location (Which Layer)

The access control design binds security attributes to network-related entities at different layers in the network stack as shown in Figure 4.1. Access control is enforced on operations at different layers as shown in Figure 4.2. This has several advantages as described below.

- By implementing the access control checks right at the point where a particular network service is being granted, we can avoid leaking any information to unauthorized processes or nodes. For example, the permission check on a connection request packet needs to be performed at the TCP layer. If the check is performed above the TCP layer, then a check can be performed only after the TCP connection is established, at which point the information that a socket was listening on that port would have been leaked. If the connection

Entity	Layer in the newtork stack
Sockets	Socket Layer
Port Names	Transport Layer
Nodes Packets Routing Table	Network Layer
Network Interface	Physical Layer

Figure 4.1. Binding security information to entities at different layers

System call	Layer in the network stack

`socket() bind() connect()`
`listen() accept()`
`getsockname() getpeername()`
`setsockopt() getsockopt()`
`send() sendto() recv() recvfrom()`
`shutdown()`
	Socket Layer

`in_pcbbind()` `tcp_input() udp_input()`	Transport Layer

`ip_input() ip_output()`
`network interface ioctls`
`routing table commands (RTM_*)`
	Network Layer

	Physical Layer

Figure 4.2. Enforcing access control on operations at different layers

request permission checks fail, then the TCP input routine can act in the same fashion as if no socket had been bound to that port, and thus the client obtains no information.

- Different applications and transport layers could implement different controls varying in strength. For example, TCP supports connection-oriented communication whereas UDP supports only datagram communication. Such differences create the necessity for different kinds of policies and enforcement. So, some of the access control decisions are bound to be extremely protocol dependent. Our design enables us to identify the services related to different protocols at the different layers and associate permissions with them. For example, we could easily specify which sockets could act as a client or a server to a particular socket using the client_associate and server_associate permissions respectively.

- Our model exports fine-grain access rights. It associates access rights with every access control check and each of these access rights could be used for policy specification. Fine-grained access rights are useful both for policy flexibility and for supporting the principle of least privilege.

- Resources may be consumed by unauthorized traffic. Rejecting packets at the application layer or even the transport layer is often too late. Doing so increases the vulnerability of a node to denial of service attacks. Our design allows us to minimize the time spent on unauthorized traffic by discarding illegal packets as early as possible.

- Some nodes provide transit services only and therefore must implement any desired controls as part of the routing protocol. Since routing is a network layer function, these controls must also involve network layer entities and cannot be left to transport layer endpoints.

- In addition, by performing the access control checks at the layer corresponding to the relevant entities, we avoid violating network protocol layering.

4.2 Binding Security Information

This section describes how security information is bound to the different network-related entities.

4.2.1 Network Interfaces

Each network interface on a node has a security context associated with it. This security context is used in access control checks when sending and receiving network packets through the interface. This design allows us to control the set of outgoing packets and the set of incoming packets that may be sent or received via the network interface. Using these checks, we can support policies that prohibit certain kinds of data from ever leaving the local node or prohibit certain kinds of data from ever entering the local node. If a node has multiple network interfaces, they can be assigned different security contexts. For example, a gateway that wishes to have different controls over the traffic to/from the internal network compared to traffic to/from the external network could assign different security contexts to the corresponding network interfaces.

Checking for incoming packets that violate the security policy at the network interface layer reduces the amount of processing time spent on illegal packets, which provides some protection against denial of service attacks.

The security context of a network interface is used in checks on bind operations that associate a socket with an address serviced by the network interface. If the access control check fails, it means that the socket cannot be bound to any reserved ports on that address and so it cannot perform any network communication using reserved ports on that address. The security context of the network interface is also used in permission checks on operations which observe or modify the interface configuration.

4.2.2 Nodes

Each node (e.g., host, router, etc.) has a security context. The node security context is used in permission checks on packets being sent to the node and on packets being received from the node. The security context of the destination host is used to verify whether packets can be sent to that host and similarly, the security context of the source host is used to check if incoming packets from that host can be accepted.

Performing these checks allows a system to avoid exposing resources to systems with which it does not want to communicate. For example, only a few nodes in an organization might need to communicate with external nodes. The ability, or lack thereof, for a node to communicate with external nodes could be specified in the security context assigned to the node. So all the nodes that do not want to communicate with external nodes simply drop the packets they receive from external nodes, by including such restrictions in the policy.

One more example would be one in which external communication should be restricted to a few subnets or nodes. In this case, the policy enforcing border router or gateway could just drop the packets that are not acceptable to the administrative domain as a whole and label the acceptable packets according to the policy, for example, based on the source host (this is a part of the policy mapping requirements). The nodes within the administrative domain could use the packet security attributes to decide whether they accept or discard the packets.

Even with hosts within the same organization, there could be some undesired dependencies, for example, misbehaving hosts generating broadcast storms, or poor transport protocol implementations degrading performance of a bottleneck resource, or hosts belonging to different groups such as accounting and research in the same organization. So, communication with other nodes in the same organization should also be controlled appropriately.

Similarly, controls are also needed for outgoing packets to reduce the risk of exporting undesired information.

In addition, the security context of a node signifies the level of trust associated with the node. When a packet is received from a node, checks need to ensure that the security attributes of the packet are consistent with the security context of the sending node. Similarly, when a packet is sent to a node, the sending node must also check the security context of the remote node to verify that it can be trusted to handle the packet appropriately.

In the current implementation, the local configuration of each network security policy server includes the security contexts for each node with which the local node may communicate. In the long term, the security contexts of nodes will be obtained from node certificates, distributed via a distributed name service such as secure DNS [35]. The network stack maintains a cache of the corresponding node SIDs for use in permission checks.

4.2.3 Routing Table

The network stack uses the routing table for obtaining information about the route on which a packet should be sent. This information includes the network interface that the packet should be sent out on and also the first hop destination along the route. User-level daemons insert routing information into the routing table, so access to the routing table by these processes must be controlled. The network security policy server's policy configuration specifies a security context for the entire routing table. The routing table SID is used to control the ability of processes to observe and modify entries in the routing table.

4.2.4 Sockets

A socket acts as a communication end point through which multiple processes can send or receive data. Hence security contexts are bound to sockets for use in send and receive operations on the sockets. In addition, sockets have private state that is user accessible (for example, using setsockopt/getsockopt). Hence, the security context of the socket is also used in permission checks on operations that manipulate socket private state. This includes operations initiated by the local process, such as bind, and operations initiated by the remote process, such as adding a newly received packet to the socket's receive buffer. In fact, send and receive operations on the socket can be viewed as operations that modify the socket's private state. A send operation adds packets to the socket's send buffer and a receive operation deletes packets from the socket's receive buffer. There are distinct socket security classes for each (domain, type, protocol) triple.

By default, a socket created via the socket call inherits the SID of the creating process. A socket_secure interface is provided to permit a process to specify the SID of the new socket.

By default, a new server-side socket created by a connection to a listening socket inherits the SID of the listening socket. A listen_secure interface is provided to permit a process to specify a SID for any sockets created as a result of subsequent connections to the listening socket. In this case, a permission check is performed during the listen_secure operation to verify that a new socket with the specified SID may be associated with the listening socket. The listen_secure interface also provides a mirror flag that may be used by processes to indicate that the SID of the remote socket should be used for the new local socket. In this case, an accept_associate check is performed when the new connection is received, to verify that a new socket with the remote socket's SID may be associated with the listening socket.

A process may obtain the SID of a local socket by using the new getsocksid interface. A process may obtain the SID of a remote socket to which a local stream socket is connected by using the new getpeersid interface. The SID of a socket may not be changed.

4.2.5 Packets

Each packet has a security context. The security context of a packet is used in permission checks on operations which send and receive packets. Additionally, for connection-oriented sockets (stream sockets), the SID of the connection request packet and the SID of the connection confirmation packet is used to indicate the SID of the remote socket.

Every packet sent via a stream socket (TCP) inherits its security context from the socket. So in our model, the socket used by the sending process acts as the *effective sender*. An alternative model is one in which the the packet inherits the security context from the sending process as done for controlling network communication with Domain and Type Enforcement [22]. Our model is consistent with the level of indirection provided by sockets in the BSD networking model. A TCP connection is not a connection between two processes; it is a connection between two sockets (in contrast, a Fluke reliable IPC connection is a connection between two threads). The set of sender and receiver processes for a TCP connection is dynamic, and the process that will receive a packet is decided only at the time of the actual receive call. Since message boundaries of data sent on stream sockets are not preserved, if we were to use the model in which the packet inherits the security context of the sending process, we would need to either prohibit multiple senders with different SIDs from using a connected stream socket or support substreams with different security contexts in a TCP connection as done in the DTE work [22]. With our current model, we can support multiple senders with different SIDs on a connected stream socket, as long as the policy permits each of them to generate packets with the socket SID. Supporting substreams with different security security contexts in a TCP connection should not be required.

By default, a packet sent via a nonstream socket also inherits its SID from the socket. Alternatively, new interfaces (send*_secure) may be used by processes to specify a SID for a packet sent via a nonstream socket. This is possible because nonstream sockets preserve message boundaries. A process may obtain the SID of a packet received via a nonstream socket by using the recvfrom_secure interface.

Packets are labeled implicitly by security association; each security association includes a packet security context.

If an unlabeled packet is received from a node, then the packet is assigned a default packet SID based on either the network interface or the source node. Default packet SIDs may be specified in the network security policy server's configuration.

4.2.6 Port Names

Though a port name is simply a name associated with a socket, it could be considered an object in the sense that applications expect particular services to be provided through particular ports. For example, one might want to forbid an untrusted sendmail server process from binding to any port other than 25 or forbid any process other than a sendmail server from binding to port 25.

The network security policy server's policy configuration specifies a default port security context for reserved ports within each transport layer protocol's port space, and may specify particular port security contexts for individual reserved port names or ranges of reserved port names within each port space. The security context of a port is used to control which sockets may be bound to the port.

Security contexts are only associated with port names that are either reserved for privileged processes (ports $<$ IPPORT_RESERVED) or reserved for servers (ports $>$ IPPORT_USERRESERVED). These constitute the ranges for well-known service ports, in contrast to the port names between IPPORT_RESERVED and IPPORT_USERRESERVED, which may be automatically selected for sockets when no explicit bind is performed prior to an operation on the socket.

4.3 Virtualized Port Number Space

As with the file name space, the UDP and TCP port number spaces provide a shared resource that can be used as a covert channel [13]. We allow any number of sockets to be bound to the same UDP or TCP port number as long as they are associated with different security contexts, thereby preventing the covert channel.

To prevent processes from using port spaces as a covert storage channel, we create two port space security classes: regular port spaces and security union port spaces. A regular port space is a collection of (socket, port name) pairs, in which each port name is unique within the port space. A security union port space is a collection of regular port spaces, where each member port space may be uniquely identified by its SID. Each transport layer protocol's port space is a security union port space, and has a corresponding security context defined in the network security policy server's configuration.

Two permissions are defined for regular port spaces:

Permission check	Entities involved	Description
Portspaces		
spacebind	portspace x socket	control association between this portspace and this socket
receive	portspace x packet	control ability of a port in this portspace to receive this packet

During a bind operation, if the port name is already in use within any of the member port spaces with which the socket may be associated, then EADDRINUSE is returned. Otherwise, the network server determines the default member port space for the (socket,sup) pair by invoking the member_sid interface. If the default member port space does not exist, then it is created. The socket is then bound into the default member port space.

When an inbound packet is received by the transport layer, the network server first checks the default member port space for the (packet,sup) pair. If the default member port space for the (packet, sup) pair does contain a binding for the port name, then the packet is delivered to that socket. Otherwise, the network server checks each of the member port spaces which may receive the packet. If exactly one of these member port spaces contains a binding for the port name, then the packet is delivered to that port. Otherwise, in the TCP case, the connection is refused, and in the UDP case, the packet is dropped.

In addition to solving the problem of the covert storage channel, security union portspaces could also be used for supporting the polyinstantiation of security-unaware server applications at different security contexts. For example, we could instantiate a database server that listens on port 2047 at each security level, and transparently directs client requests to the correct instantiation.

CHAPTER 5

NETWORK CRYPTOGRAPHIC PROTECTION

As a basic premise, access control mechanisms require reliable identification and authentication of users in the network. In addition, we need to protect the information transmitted over the network to make wire tapping ineffective. This requires cryptographic techniques to provide confidentiality, integrity, and authentication. This section deals with the cryptographic protocols we are planning to use and the rationale behind the choices made.

5.1 Which Layer

Placement of the cryptographic service within a specific layer of the network stack has far-reaching implications concerning the nature and extent of the service implementation. This section describes the different options available.

Data link layer: Performing encryption at the level of the data link layer has the advantages of being transparent to applications running at higher layers and being implementable in hardware, but it also suffers from several disadvantages [36]. If an intermediary is used for transmission between different nodes, then a clear-text version is available for perusal at the intermediary node. Also the cost of key distribution could be huge as all intermediate nodes will need to exchange keys. In addition, end-to-end measures are more naturally suited to users' perceptions of their security requirements.

Network layer: IPsec [7, 8, 9] provides a standard for IP layer authentication and encryption. The problem with IP layer cryptographic processing is that it does not support security associations between end users in a direct manner. A security association specifies how the communicating entities will use cryptographic mechanisms to protect network traffic. It is discussed in detail in section 5.2. Bellovin in his discussion on the weaknesses of the IP layer security protocols [37] recommends moving the cryptographic processing towards the transport layer. He suggests that the semantics for creating security associations are quite clear at the transport layer. Transport layer offers process level communication and hence guarantees a secure pipe between two users rather than between two machines.

Some IPsec implementations, such as the one by NRL [38], support such security associations, but its design violates traditional layering by making use of transport layer information at the IP layer.

Transport layer: Netscape's Secure Sockets Layer(SSL) [32] provides a standard for authentication and encryption at the application layer (i.e., above the transport layer). The Transport Layer Security (TLS) working group is working on standardizing this protocol with minor changes [15]. SSL has the advantages of being application protocol independent and requiring no modification to the operating system.

The SSL protocol has the problem that the security provided is not transparent to the application and all the networking applications that need security will need to be rewritten to comply to the standard. Our intention is to provide cryptographic functionality transparently to all applications in accordance with a mandatory cryptographic policy. So, if SSL were to be used, the SSL functionality would have to be implemented within the transport layer.

Also, since SSL is implemented above the transport layer, it does not ensure the integrity of the transport layer protocol header or the binding between the header and payload. In addition to this, the SSL protocol does not provide a way to distinguish SSL packets from non-SSL packets. Also SSL does not provide the equivalent of a Domain of Interpretation (DOI) provided by ISAKMP for interoperability purposes. DOI is described in section 5.5.

> Solving these problems would mean that we would need to modify the SSL protocol significantly and this would make interoperability with SSL extremely difficult, if not impossible.

After considering the above we decided to use NRL's implementation of IPsec for providing the cryptographic mechanisms. Although this is probably not an ideal solution, we feel this is the one that best suits our needs. We use the ISAKMP protocol [10] for providing the key management functionality required by IPsec.

5.2 IPsec

We use the IPsec implementation provided by NRL [38] for providing the encryption and authentication mechanisms. NRL's code was written for NetBSD, BSDI and other operating systems. We have ported this to the FreeBSD TCP/IP stack in Fluke.

The Security Architecture for the Internet Protocol [7] provides a high-level definition for two IP security mechanisms, the IP Authentication Header (AH) [8] and the IP Encapsulating Security Payload (ESP) [9]. IP AH is a mechanism for providing integrity and host authentication for IP datagrams. IP ESP is a mechanism for providing confidentiality for IP datagrams. IP AH and IP ESP are not bound to specific cryptographic algorithms. However, at a minimum, implementations must support keyed MD5 [39] for IP AH and DES-CBC [40] for IP ESP.

When two or more entities wish to use the IP security mechanisms to protect network traffic between them, they must first establish a **security association** (SA). A security association specifies how the communicating entities will use the security mechanisms to protect network traffic. Two security associations will need to be established on each host to specify how inbound and outbound traffic should be protected. Each inbound security association on a host may be uniquely identified by a local **security parameter index** (SPI). Each outbound security association on a host may be uniquely identified by the pair (remote host, remote SPI).

A security association consists of two primary components: a network situation and a set of attributes. The **network situation** contains the information that is relevant to the network usage policy, which is described below. The situation consists of a (source address, destination address, source socket security context, destination socket security context, packet security context) tuple. For an inbound security association, the set of attributes includes a local SPI and for an outbound security association, the set of attributes includes a remote SPI. In both cases, the set of attributes includes the set of mechanisms (e.g., none, ESP, AH or both), and cryptographic information (e.g., algorithms, keys, initialization vector length, and cryptographic checksum length).

These security associations could be set up manually using a command or on the fly using negotiation servers such as ISAKMP. The NRL implementation supports the PF_KEY Key Management API [41], that application-layer key management daemons, such as Cisco's ISAKMP daemon, can use to configure IP security associations (including keys) into the operating system. IPsec depends on ISAKMP for establishing the security associations.

5.3 ISAKMP

The Internet Security Association & Key Management Protocol (ISAKMP) [10] provides a framework for Internet key management and specific protocol support for negotiation of security attributes. ISAKMP in combination with the Oakley Key Determination Protocol [11] provides full security association attribute negotiation, as well as authentication methods that provide both repudiation and nonrepudiation.

We use Cisco's ISAKMP implementation for key management. This operates as a user-level daemon as shown in Figure 5.1 The Cisco ISAKMP daemon uses the PF_KEY Key Management API [41] to register with an operating system kernel (which supports this API) and the surrounding key management infrastructure. Security associations that have been negotiated by the ISAKMP daemon are inserted into the kernel's key engine and are then available for use by the system's

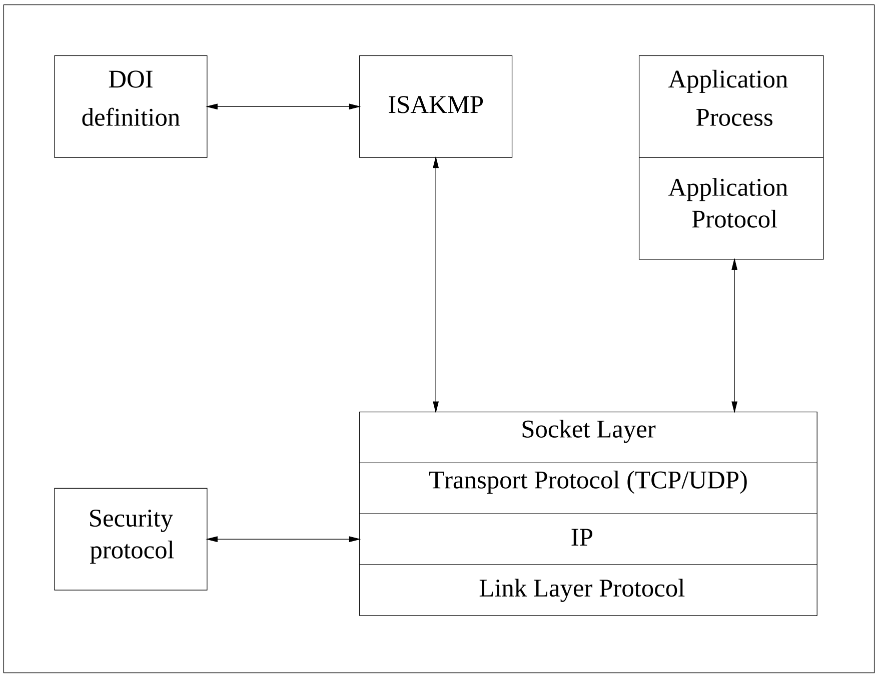

Figure 5.1. Placement of ISAKMP in the network architecture

standard IPsec security mechanisms (i.e., AH and ESP).

5.3.1 Triggering ISAKMP

Negotiations are triggered when a network situation occurs for which there is no established security association. When a process invokes connect or send, and a security association does not already exist for the situation, the IPsec layer performs an upcall to the negotiation server (ISAKMP) requesting an outbound association. After the key exchange is complete, a security association is available for use by the outgoing and incoming traffic.

When a protected packet is received by a node, and no association exists for the SPI in the packet, then the IPsec layer sends a notification to the negotiation server requesting an inbound association and drops the packet. This may indicate that this node was engaged in secure communications with the remote node and rebooted, thereby losing its security associations, which must be reconstructed.

5.4 Network Usage Policy

The Network Usage Policy determines which cryptographic services are required and preferences for the cryptographic algorithms for a particular **network situation**. The network situation contains all the security-relevant information that a system considers necessary to decide the security services required to protect the session being negotiated. It consists of the (source address, destination address, source socket security context, destination socket security context, packet security context) tuple.

The process of selecting a cryptographic service based on the network situation is generally transparent to the applications. However, an interface is provided using which applications may specify stronger cryptographic preferences. The negotiation server consults the network usage policy while negotiating the protection suites.

5.5 Domain of Interpretation (DOI)

Groups of communicating nodes need to agree on some common interpretation of security label and policy information as well as the cryptographic mechanisms. Within ISAKMP, a Domain of Interpretation is used to group related protocols using ISAKMP to negotiate security associations. Security protocols sharing a DOI choose security protocol and cryptographic transforms from a common namespace and share key exchange protocol identifiers. They also share a common interpretation of DOI-specific payload data content, including the security association and identification payloads. All the nodes using the same DOI lie within the same security perimeter. Some nodes could probably implement multiple DOIs and act as gateways between different security perimeters.

The Internet IP Security Domain of Interpretation for ISAKMP document [42] defines the naming scheme and syntax of the DOI specific fields for the IPsec protocols that use ISAKMP to negotiate their security associations, but this does not satisfy all our requirements such as including the packet security context and other security contexts in the network situation. So we define a new Flask DOI which is based on the IPsec DOI. This DOI allows the exchange of the security contexts required by our access control model.

Implementing this DOI requires changes to the ISAKMP and IPsec implementations. Our implementation supports both the Flask and ISAKMP DOIs and so interoperability with machines using the IPsec DOI is preserved.

CHAPTER 6

IMPLEMENTATION AND RESULTS

The implementation is a part of the Flask secure operating system. The base network stack we use is the FreeBSD TCP/IP stack on Fluke. We use NRL's IPsec implementation [43] and Cisco's ISAKMP implementation [44]. Both have been modified to deal with the network situation as required by our access control model.

The communication between different components in the network security architecture occurs as shown in Figure 6.1. Access control checks are performed on different network-related operations and at different layers as required by the design. Appendix B shows how the functions in the network stack are modified for implementing the access control checks. Access control checks are implemented by making calls to the security server. Security attributes of network-related entities such as nodes and ports are obtained by querying the network security policy server.

Each outgoing packet has information about the IPsec protocols to be used for the packet and the **network situation** associated with the packet. The network situation consists of the source socket security context, destination socket security context, and the packet security context. The IPsec layer looks for a security association that can be used for this packet. If no security association can be found, it sends a request for setting up a security association to the ISAKMP negotiation server and enqueues the packet. It also sends the network situation along with the request. The ISAKMP negotiation server contacts the ISAKMP daemon on the remote host and key exchange takes place. They also decide on a SPI (security parameter index) for the security association and send the information to the IPsec layer. If the remote host is a Flask node, the network situation is sent to the remote

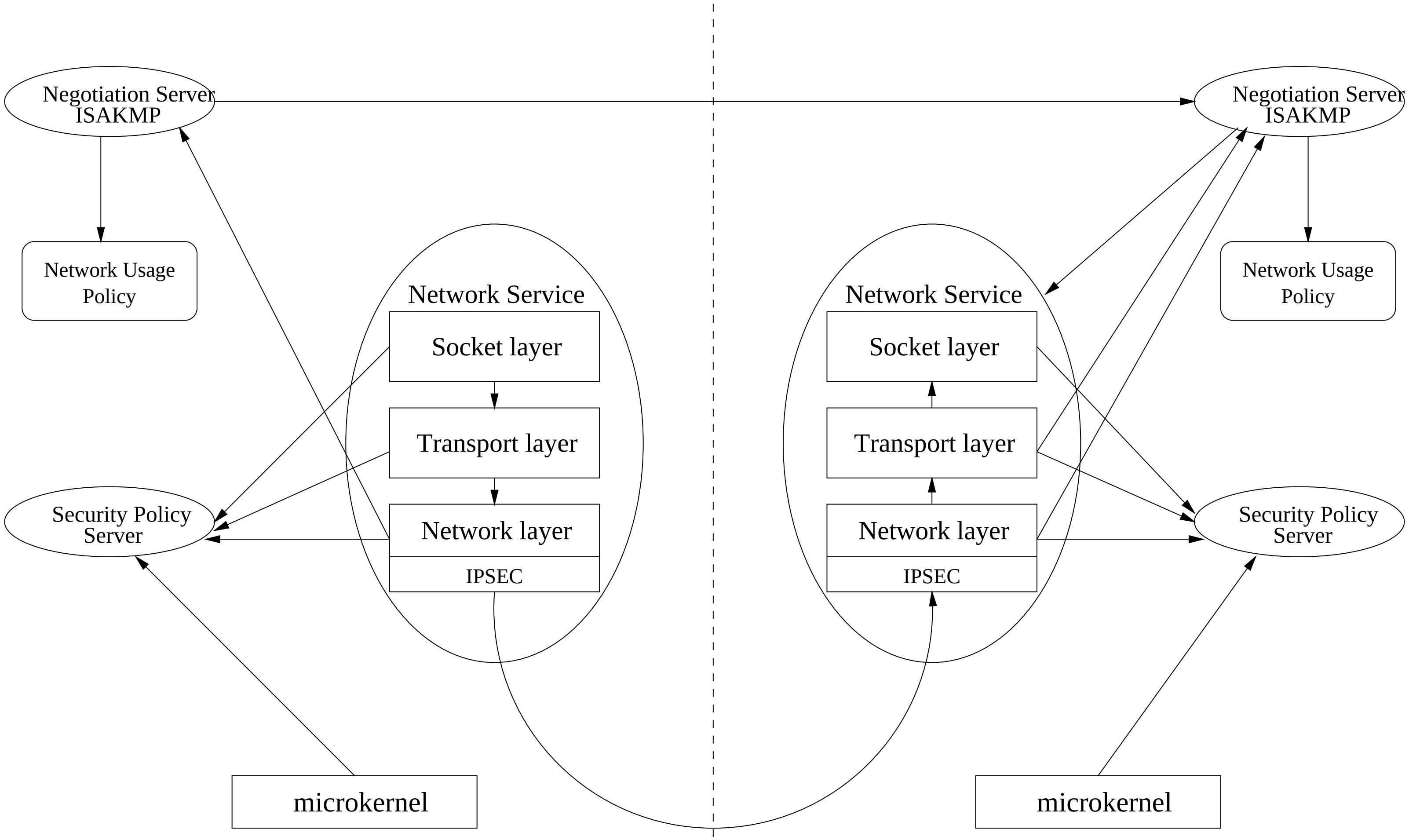

Figure 6.1. Network security architecture

host during this exchange and is stored along with the security association on the remote node.

Once the security association is set up, the packet can be sent after applying the required cryptographic transforms and adding the headers. The remote host identifies the security association using the SPI in the packet header. It then applies the required transforms to obtain the decrypted packet. When this packet is sent to the upper layers for processing, the network situation associated with the packet is also sent along with the packet. The security contexts in the network situation are used for access control checks in the layers above. They are also used for giving information about the security contexts to applications requesting such information using the new system calls we provide.

One of the important goals is to preserve interoperability with normal IPsec and TCP/IP hosts. In order to achieve this goal, a Flask DOI (Domain of Interpretation) is introduced in addition to the IPsec DOI supported by the implementation. If the remote host is a Flask node, the FLASK DOI is used. As mentioned earlier, in this case, the network situation is exchanged during the security association set up process. The network situation is stored along with the security association in the IPsec layer. So, the situation corresponding to an incoming packet can be obtained using the SPI from the IPsec headers in the packet. If the remote host is not a Flask node but supports IPsec and ISAKMP, then the IPsec DOI is used. In this case, the network situation is not exchanged and a default network situation corresponding to the security context of the node and the default packet security context of the node is used. Even in the case of communicating with a normal TCP/IP host that does not support IPsec, the default network situation is used. In this case incoming and outgoing packets do not require any IPsec-related processing.

If a permission request for receiving a data packet or a connection request fails, the message or connection is silently dropped without any notification to the receiver. This is to avoid a channel. In the case of a connection request packet, the sender will receive a reset. In the case of a data packet, the sender will not receive any notification.

In the case where a packet is received with an SPI that does not correspond to any security association in the kernel's key engine, a key acquire message is sent to the negotiation server. This is done because the SPI could have been referring to a valid security association that was deleted because of a reboot.

A cache of the access vectors is maintained to avoid repeated calls to the security server. There should be benefits due to caching because usually the same socket is read from or written to repeatedly and if caching is not done, this would lead to a number of calls to the security server.

6.1 Results

Although extensive performance analysis was not performed, this section describes and analyzes the results of certain measurements that allow estimation of the performance of connection establishment and latency, as well as cryptographic costs. We also outline some of the tests we performed to validate the implementation's provision of access control.

Methodology: All tests were performed on 200 MHz Intel Pentium Pros with 128MB of memory, using 100 Mbit switched ethernet. The IPsec protocol used was ESP with the DES encryption algorithm. In terms of kernel and nester configuration, we used only the Flask microkernel,[1] security policy server, the secure network server, and the process manager; note that the virtual memory manager was not used.

6.1.1 Performance

Connection establishment: Six separate runs were made to gather connection establishment data. Performance results show very high overhead, but only when it is necessary to set up security associations. The median time for an entire TCP connection setup that requires the construction of two security associations is about 6.1 seconds. A breakdown of the cost is presented in Table 6.1. Two steps

[1] Flask configuration options were "–enable-fdev –enable-flask –enable-fluke-inlining –enable-inlining –enable-opt-mutex –enable-opt-acount –enable-opt-xfer i486-linux" with a kernel updated as of January 25, 1998.

Table 6.1. Times for connection establishment

Operation	SAs and access vectors not in cache (time in milliseconds)			SAs and access vectors in cache (time in milliseconds)		
	Min	Max	Median	Min	Max	Median
server SA setup	1870	3370	2760	0	0	0
client SA setup	1870	3370	2760	0	0	0
server IPsec queue check	31	494	355	0	0	0
client IPsec queue check	31	494	355	0	0	0
access control checks	4.8	5.3	5.0	0.33	0.46	0.41
IPC calls for SUPs	3.3	3.6	3.5	3.3	3.6	3.5
Base cost (with no network security measures)	3.2	3.5	3.4	3.2	3.5	3.4
Encryption	1.09	1.63	1.34	1.09	1.63	1.34
Total Connection establishment time	3814	7742	6243	7.92	9.19	8.65
Observed Connection establishment time	5920	6250	6130	8.2	8.9	8.6

are responsible for almost all (99.8%) of the cost: SA setup and the associated IPsec queue checking before an SA is known to exist.

Since the same code path is executed for SA setup on both server and client sides, we combined our individual measurements of both client and server times to arrive at a median SA setup time of 2.76 seconds.

ISAKMP security association setup is to some degree an inherently expensive process, especially on a wide-area network, requiring seven message exchanges between the ISAKMP daemons to agree on a set of protocols and transforms. There are also cryptographic costs associated with key generation and exchange. However, we believe our observed costs are not representative of reasonable ISKAMP implementations. In particular, the ISAKMP daemon unnecessarily uses asynchronous timeouts to drive its internal finite state machine, introducing both large amounts of latency and timing variability. Initial investigation suggests that this unusual asynchronous design is responsible for the majority of ISKAMP costs. Since this thesis work concentrates on the implementation of network access control and treats ISAKMP simply as an externally-provided service, further pursuit of its performance, design and implementation awaits a future investigator.

If the required security association for a packet is not present, the packet is queued and the kernel periodically checks, currently every 0.5 second, whether the security association has been established. Similarly to the SA setup times, we considered both the server and client queue check times in arriving at a median cost of 355 milliseconds for each check.

Significant time is lost due to the asynchronous design of this check. If the kernel is configured to check for the security association less frequently, for example, every 3 seconds as was tried earlier, the impact of this asynchrony is much higher. If configured to check significantly more frequently, for example, every 0.1 seconds as was also tried, the overall cost rises slightly, perhaps due to the expensive implementation of timeouts in the current Fluke system. The interval of 0.5 second was chosen as a middle ground. A better implementation of IPsec could probably eliminate this asynchrony completely, reducing connection establishment time by up to 1.0 second.

In comparison to the costs associated with security association set up, other security-imposed costs on connection establishment are very small, totaling only 13 milliseconds. As shown in the table, these costs include fetching all of the access vectors from the security server into the access vector cache, costing 5 milliseconds. Security union portspaces (SUPs) impose a cost of 3.5 milliseconds by requiring IPC calls at connection establishment. "Encryption cost" of 1.3 milliseconds is the overhead due to encrypting the connection packets. Finally, the "base cost" of 3.4 milliseconds is the time required for connection establishment on a similar Flask-based configuration, but without any of the code implementing the network security measures described in this thesis. (We also made four runs of a vanilla Fluke configuration, finding a similar connection cost of 3.7 milliseconds.)

The performance of TCP connection establishment improves approximately three orders of magnitude if the required security associations are already present in the kernel's key engine; it drops to about 8.6 milliseconds. A breakdown of the cost is presented on the right-hand side of Table 6.1. The differences are twofold. First, the costs associated with SA setup drop to zero. Second, access control costs are reduced over an order of magnitude due to access vector caching, dropping from 5.0 to .41 milliseconds. In this situation, security-imposed costs cause connection establishment time to increase from 3.4 milliseconds (in Flask with no network security) to 8.6 milliseconds, an additional delay that is not perceptible to humans.

In summary, connection time takes from 2.5 to 1800 times longer than a version of Flask containing no network security measures, depending whether security associations are already set up and the access vectors cached. It should be noted that setting up a security association is required only if a security association for the particular network situation does not already exist. For example, once the security associations required for a user to communicate with an HTTP server on another node are set up, neither data transfer nor new connections between that user and the HTTP server will require new security associations to be set up. In fact, if some other server such as the FTP server has the same security context as the HTTP server, the user can communicate with such a server without requiring new security associations to be set up.

Latency: We assessed latency by measuring, over 11 runs, application-to-application roundtrip time of a TCP packet carrying one byte of data. We obtained a median round trip time of 5.2 milliseconds; the cost breakdown is presented in Table 6.2. On a similar Flask kernel and server configuration, but with all network security code "ifdef'ed" out, the round trip time was 3.7 milliseconds. However, for

Table 6.2. Round-trip time for a 1 byte TCP message

Operation	Time (in milliseconds)
base (with no network security measures)	3.7
access control (including SUPs)	0.4
encryption	1.1
Total	5.2

reasons unknown at this time, round trip times on vanilla Fluke were approximately half twice as fast, averaging 1.9 milliseconds. Security union portspaces (SUPs) and access control checks together introduce an overhead of 0.4 milliseconds, on average split approximately evenly between the two. In individual runs the contribution of each to this total cost varies widely, between 0.1 and 0.3 milliseconds. SUPs are not costly because the TCP data packets require no IPC calls. The cost due to access control is also low because all of the access vectors are in the cache. Finally, we estimated the cost due to encryption by using a "null" security association, which specifies that no IPsec protocol should be used. The cost due to two encryptions (out and back) of the 20 byte TCP header and 1 byte payload was found to be 1.1 milliseconds.

In summary, the network security measures in this Flask implementation increase latency by about 40% for minimum size packets. However, since a vanilla Fluke configuration exhibits a latency less than half that of Flask configured without any network security, the true security-imposed effects on latency may be much higher.

Cryptographic costs: We made some attempt to isolate the costs of encryption, since cryptographic hardware is likely to be available in the future. In addition to the 1 byte tests reported above, which showed a latency ratio of 1.4, we made four runs each with 512 and 1024 byte packets. For 512 byte packets, round trip time increased from 3.8 milliseconds to 5.6 milliseconds when encryption was enabled, a ratio of 1.5. For 1024 byte packets, encryption caused round trip time to increase from 4.3 milliseconds to 7.8 milliseconds, a ratio of 1.8. For long packets it appears that the effect on latency, and therefore speed of data transfer, would approach a factor of two on this hardware. If encryption costs were to drop to near zero, the remaining costs (access control and SUPs, taking 0.4 milliseconds) would impose only about a 10% penalty above the average latency of around 4 milliseconds.

Finally, we note that Fluke is an experimental prototype and has not yet been optimized. In addition, both the IPsec and ISAKMP implementations are experimental prototypes imported from elsewhere and involve unnecessary overhead, especially ISAKMP. Therefore, determining the true inherent overhead due to security would involve optimizing these implementations.

6.1.2 Example Scenarios

Several example scenarios have been tested to show the effectiveness of our access control model. Some of these are described below.

One of the scenarios tested shows how the inetd daemon could be modified to take advantage of our implementation. The listen_secure interface is used to specify that the server socket created for the TCP connections should have the same security context as its peer. The execve_secure interface, provided by the process manager, is used to specify that the server process created should also use the same security context. Using this approach, the server side process created to service the client request is confined to the security context of the peer, thereby limiting the operations it can carry out. For example, in the scenario tested, it is shown that a restricted file cannot be read in response to a request from an unprivileged client.

One more test shows how a simple policy permits all outbound TCP connections from a process and prevents any inbound TCP connections, as done commonly in network firewalls. This mitigates the risk of allowing untrusted outsiders to initiate communication with an insider.

Another test shows that a process cannot bind to a port forbidden by the access control policy. This, for example, prevents an arbitrary untrusted program from binding to the telnet port and answering incoming telnet requests.

Another test shows how the checks based on the security context of the network interface can be used to ensure that a process cannot send data out of the node, if the data with that security context is not permitted to be sent out of the node. If such a process attempts to send data or initiate a connection with a remote node, the operation is terminated with an error indicating denial of access. This helps prevent a process from exporting data unless the destination is a host trusted to handle that data in a secure manner. For example, a gateway with two network interfaces could ensure that sensitive data are not permitted to go out of the node via the external network interface.

The usefulness of security union portspaces (SUPs) is demonstrated by instantiating a security-unaware server application at different levels on the same port and transparently directing the client requests to the correct instantiation, based on the security context of the incoming packets. This approach has the advantage of using existing server applications without any modifications, allowing the server applications to be completely untrusted.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the contributions of this work and concludes with a discussion of unresolved issues and future work.

7.1 Contributions

The necessity for network security is widely appreciated. This work follows the approach of providing security in the network stack of the operating system, as shown to be required by research in security. This thesis presents the design and implementation of a framework for mandatory access control and cryptography in the network stack. The implementation supports transparent mandatory security enforcement for all applications. Security aware applications can specify preferences using the new system calls we provide for this purpose.

We follow the Synergy approach of separating policy from enforcement. In our implementation, the security attributes are interpreted by the security server alone so it can support a variety of policies including dynamic policies to some extent. The access control model is layered to match the network stack. The access control requirements are mapped onto different layers in the network stack, depending on the functionality provided by the layer. This gives several advantages including less time spent on illegal packets and support for fine-grain access rights. Fine-grain access rights are important for supporting both policy flexibility and the principle of least privilege.

We use standard cryptographic protocols for wide interoperability. The IPsec protocol is used for cryptographic support at the IP layer and the ISAKMP protocol is used for key management. Though our implementation requires passing some additional security information, interoperability with the original protocols is preserved.

We know of no other efforts that attempt to solve the problem of the covert storage channel due to the shared port number space. In addition, the security union portspaces that we create to solve this problem could also be used for supporting the polyinstantiation of security-unaware server applications at different security contexts.

We show experimental results to demonstrate the costs of security services in the network stack. The cryptographic costs and access control costs are identified separately. Interoperability with standard IPsec and TCP/IP implementations is also demonstrated.

7.2 Future Work

This section points to directions for useful extensions of this work.

One of the goals of the Synergy approach is to have a cryptographic subsystem that makes decisions about the cryptographic transforms to be used based on the security policy. In the current implementation, the code that selects the cryptographic measures to be adopted is not separated out from the code that enforces the cryptographic requirements. This needs to be done to meet the Synergy goals.

Multicast communication is gaining increasing importance. The ISAKMP protocol does not deal with setting up security associations for multicast communication. Recent work [45] describes a framework for securing multicast communications. A future version of ISAKMP will probably support security associations for multicast communication.

Our design does not deal with routing between administrative domains and secure policy routing in general. We would like route selection to be based on the security policy. This would require support for multiple routes for the same host/network and a particular route is selected based on the packet security context, but the framework for this is not very clear currently. An implementation of mobile-IP [46] has extended NRL's IPSEC implementation to allow IPSEC security associations to be bound to routes. The motivation for this was to allow virtual private networks to be created by simply configuring the routing table appropriately. We could bind security attributes to routes in a similar fashion and allow for selection of a route based on the security policy.

Protection against denial of service always seems to be one of the most difficult to address. ISAKMP's cookie (or Anti-Clogging token) aims at protecting the computing resources from attack without spending excessive CPU resources to determine the authenticity of the packet. Prevention against denial of service is not always possible. The practical alternative is intrusion detection and auditing symptoms of possible attacks [47, 48, 49, 50].

Our implementation assumes that different Flask nodes on the network use a common set of security contexts. The security contexts are exchanged during the security association setup, and checks on incoming packets are based on these security contexts. In the real world, policy mapping services are required for mapping these security contexts to attributes that the host system understands. This could be done either by a table which gives the mappings or in a more dynamic fashion.

In our implementation, the security contexts of the nodes a host communicates with are obtained from network security policy configuration. Once stable DNSSEC [35] implementations become available we need to make use of the DNS services for obtaining the security contexts of the nodes in a secure manner.

Experience with real-world applications and policies will decide whether changes are required to the access control design. For example, though we currently do not see the need to support data substreams with different security attributes in TCP connections, we have not been able to prove conclusively that they are not required either. So, if a real-world scenario requires substreams, then we will need to support them in our design and implementation. A rigorous analysis of the security semantics required by real applications in general and commercial off the shelf (COTS) applications in particular is bound to improve the design. Each organization would probably use just one or very few policies as the administrative costs of managing multiple policies could be very high. The external communication would be resolved by the border routers which do the necessary policy mapping function also.

Though our design and implementation are limited to the TCP/IP (Internet) world, the design could easily be viewed in a protocol independent manner and it could be extended to other network architectures such as the OSI.

The ISAKMP framework uses an out-of-band approach where security association setup is done by user-level daemons upon request by the kernel. Recent work [51] shows that an in-band approach which piggybacks the keying headers on the TCP connection setup packets could be more efficient. This approach has not yet been tested as widely as ISAKMP.

Other Flask services could be made use of to demonstrate the advantages of the Flask mandatory access control framework. For example, we could use the services provided by the process manager to control the execution of services spawned in response to the requests from clients. We could use the memory manager services to place keys in a special memory location, so that some unauthorized process cannot get access to this this key, because it will not have read permissions for the location where the key is stored.

Misconfigured or malicious hosts on the same network could pose threats to ARP (Address Resolution Protocol) security. A solution could be to use an authenticated database for obtaining the ethernet address corresponding to an IP address and discarding all the ARP replies that could be sent by other hosts.

Our implementation and design do not deal with the covert channels possible due to the pattern analysis of the network traffic. Doing this requires generating random network traffic and thereby impairs performance. This is not one of our goals.

We use the ISAKMP exchanges required for setting up a security association to convey security-related information. An alternative approach [3, 22] is to use CIPSO (IP-security options) [52, 53] to label the packets with the appropriate security label. This approach provides greater flexibility of separating the expensive security association setup from the exchange of security information. A combination of both these approaches could be used where the IPsec security association could be used to exchange the range of security contexts and the packet label could be used to specify the exact security context.

APPENDIX A

GLOSSARY

Access vector A vector of all possible accesses that one security context can make on another. This is returned by the security server and is cached by the policy enforcer

Confidentiality threat Dissemination of information to an individual for whom that information should not be seen.

Covert channels Indirect channels, which are not normally protected by mandatory controls and through which information could flow to an unauthorized entity

Denial of service threat A threat arising when access to resources is blocked as a result of malicious action taken by another user.

Discretionary access control (DAC) The procedures and mechanisms that enforce the specified mediation at the discretion of the individual users

Domain of interpretation (DOI) The conventions for naming security-relevant information such as security policies or cryptographic algorithms and modes across different machines

Flask A secure operating system built from Fluke

Fluke A microkernel-based operating system being built by the Flux team

Integrity threat Any unauthorized change to information stored on a computer or in transit between computer systems

IP authentication header (AH) A security mechanism of IPsec for providing integrity and host authentication for IP datagrams.

IP encapsulating security payload (ESP) A security mechanism of IPsec for providing confidentiality for IP datagrams

IPsec An IP layer security protocol from IETF

Internet security association & key management protocol (ISAKMP) A protocol for the establishment of security associations and key management; used along with the OAKLEY key exchange protocol

Mandatory access control (MAC) The procedures and mechanisms that enforce the specified mediation at the discretion of a centralized system administration facility and not at the discretion of individual users

Multilevel security (MLS) A military security policy with linearly ordered security levels. Read operation is only allowed from a higher level entity to a lower level entity and write operation is only allowed from a lower level entity to a higher level entity

Network situation All the security-relevant information that a system considers necessary to decide the security services required to protect the session being negotiated. It may include the addresses, security classifications, modes of operation, etc.

Network usage policy The policy determining the cryptographic services and mechanisms for a particular network situation

Nonrepudiation A security property that ensures that a participant in a communication cannot later deny having participated in the communication. This property may apply to either the sender or the recipient of communicated data or both

OAKLEY A key exchange protocol used with ISAKMP

Policy enforcer Any entity in the Synergy/Flask architecture such as the microkernel, file server, network server, etc. which enforces the security policy. It makes the access control decision by consulting the security server

Protection suite A list of security services that must be applied by various security protocols. All of the protections in a suite must be treated as a single unit

Secure sockets layer (SSL) A security protocol above the transport layer from Netscape

Security association (SA) A simplex (unidirectional) logical connection, created for security purposes. All traffic traversing an SA is subjected to the same security processing at the transmitter and receiver.

Security class A distinct type of object with a distinct set of legal operations: examples, thread, port, segment, file, filesystem

Security context The set of attributes bound to an object which are relevant to the security policy

Security identifier (SID) An opaque handle to a security context

Security label A label for an entity for which security is being enforced. It indicates the security level. For example MLS uses labels like unclassified, confidential, secret, and top secret.

Security parameter index (SPI) An identifier for a security association, relative to some security protocol.

Security perimeter A collection of nodes adhering to a particular security policy.

Security policy The conditions governing actions of and relationships between entities (users, files, network interfaces, etc.) within a system.

Security server A server in the Synergy/Flask architecture which makes all the security policy relevant decisions. All the entities which enforce the policy such as the microkernel, file server, network server, etc. consult the security server for making the access control decisions.

Security union port space (SUP) A collection of regular port spaces, where each member port space may be uniquely identified by its SID. This is basically a virtualized port number space where processes with different security contexts can use the same port. This prevents the covert channel which exists when processes with different security contexts share the same port number space.

Synergy A computer security architecture made use of by Flask. DTOS was the earlier prototype built using this architecture.

Transport layer security (TLS) A security protocol above the transport layer based on SSL. IETF is currently working on making this a standard

Type enforcement A security policy "mechanism" which classifies subjects into domains and objects into types. Access control is achieved using two tables: Domain Definition Table (DDT) and Domain Transition Table (DTT). DDT represents what operations subjects in a specific domain can perform on objects of a specific type. DTT indicates the domains to which a subject in a given domain can pass execution

APPENDIX B

NETWORK ACCESS CONTROL PERMISSIONS

B.1 Permissions for Each Network Security Class

Permission check	Entities involved	Description
Network Interfaces		
getattr	subject x netif	control ability to get the attributes of this network interface
setattr	subject x netif	control ability to set the attributes of this network interface
preceive	packet x netif	control ability to receive this packet on this network interface
psend	packet x netif	control ability to send this packet on this network interface
nreceive	node x netif	control ability to receive from this node on this network interface
nsend	node x netif	control ability to send to this node on this network interface
Nodes		
receive	packet x node	control ability to receive this packet from this node
send	packet x node	control ability to send this packet to this node

Permission check	Entities involved	Description
Routing table		
observe	subject x rtable	control ability to get information from the routing table
modify	subject x rtable	control ability to set information in the routing table
All Socket security classes		
create	subject x socket	control ability to create a socket with this SID
getlocal	subject x socket	control ability to get the name of this socket
setlocal	subject x socket	control ability to set the name of this socket
getremote	subject x socket	control ability to get the name of this socket's peer
setremote	subject x socket	control ability to set the name of this socket's peer
getopt	subject x socket	control ability to get the options of this socket
setopt	subject x socket	control ability to set the options of this socket
setprotoopt	subject x socket	control ability to set protocol options for packets sent via this socket
send	subject x socket	control ability to add data to this socket's send buffer
receive	subject x socket	control ability to remove data from this socket's receive buffer
disable_send	subject x socket	control ability to disable sends on this socket
disable_receive	subject x socket	control ability to disable receives from this socket
portbind	portname x socket	control association between this portname and this socket
addrbind	netif x socket	control association between this network interface and this socket

Permission check	Entities involved	Description
Stream sockets		
listen	subject x socket	control ability to enable this socket for receiving connections
accept	subject x socket	control ability to accept the next connection on this socket
accept_associate	socket x socket	control which new sockets may be created from this listening socket
client_associate	socket x socket	control ability to set the client of a socket to this socket
server_associate	socket x socket	control ability to set the server of a socket to this socket
unidir_client_associate	socket x socket	control ability to set the client of a socket to this socket allowing data flow only from client to the server
unidir_server_associate	socket x socket	control ability to set the server of a socket to this socket allowing data flow only from client to the server
Nonstream sockets		
recv_associate	packet x socket	control which packets may be added to this socket's receive buffer
send_associate	packet x socket	control which packets may be added to this socket's send buffer

B.2 Permission Checks for Each Network Operation

B.2.1 Routing Table Commands

1. RTM_GET

   observe in allowed(process,rtable)

2. RTM_ADD, RTM_DELETE, RTM_CHANGE

   modify in allowed(process,rtable)

B.2.2 Network Interface Ioctls

1. SIOCG*

   getattr in allowed(process,netif)

2. SIOCS* SIOCAIFADDR SIOCDIFADDR

   setattr in allowed(process,netif)

B.2.3 Socket Layer

1. socket(domain,type,protocol,sid,out_so)

   create in allowed(process,(type)process)

2. socket_secure(domain,type,protocol,sid,out_so)

   create in allowed(process,(type)sid)

3. accept(so,out_addr,inout_addrlen,out_so)

   accept in allowed(process,so)

4. bind(so,addr,addrlen)

   setlocal in allowed(process,so)

5. connect(so,addr,addrlen)

   setremote in allowed(process,so)

6. connect_secure(so,addr,addrlen)

   setremote in allowed(process,so)

   so→so_remote_sid = ssid

7. getsockname(so,out_addr,inout_addrlen)

   getlocal in allowed(process,so)

8. getpeername(so,out_addr,inout_addrlen)

   getremote in allowed(process,so)

9. getsockopt(so,level,optname,out_optval,inout_optlen)
   getopt in allowed(process,so)

10. setsockopt(so,level,optname,optval,optlen)

    if (level == SOL_SOCKET)
            setopt in allowed(process,so)
    else
            setprotoopt in allowed(process,so)

11. listen(so,backlog)
    listen in allowed(process,so)

12. listen_secure(so,backlog,sid,mirror)

    listen in allowed(process,so)
    if (sid was specified and mirror is TRUE)
            invalid arguments
    if (sid was specified)
    {
            accept_associate in allowed((socket)sid,so)
            so→so_newconn_sid = sid
    }
    so→so_newconn_mirror = mirror

13. recv(so,buf,len,flags,actual)
    recvfrom(so,buf,len,flags,addr,addrlen,actual)
    recvfrom_secure(so,buf,len,flags,addr,addrlen, sid,actual)
    receive in allowed(process, so)

14. send(so,buf,len,flags,out_actual)
    sendto(so,buf,len,flags,addr,addrlen,out_actual)

```
sendmsg(so,msg,flags,out_actual)
 send in allowed(process, so)
if (so→protocol == TCP &&
        (so→so_dirflags & TCPF_OUT == 0))
        error
```

15. sendto_secure(so,buf,len,flags,addr,addrlen,sid,out_actual)

```
send in allowed(process, so)
send_associate in allowed((packet)sid,so)

if (so→protocol == TCP &&
        (so→so_dirflags & TCPF_OUT == 0))
        error
```

16. shutdown(so, how)

```
switch(how)
{
        case 0:
                disable_receive in allowed(process,so)
                break
        case 1:
                disable_send in allowed(process,so)
                break
        case 2:
                disable_receive in allowed(process,so)
                disable_send in allowed(process,so)
                break
}
```

B.2.4 Transport Layer

1. in_pcbbind

```
if (nam)
{
        sin = nam
        if (sin→sin_port < IPPORT_RESERVED ||
              sin→sin_port > IPPORT_USERRESERVED)
        {
              portbind in allowed(port_number,so)
              netif = ifa_ifwithaddr(sin)
              addrbind in allowed(netif,so)
        }
}
```

2. tcp_input

Receive connection request packet on so (server side):

```
        so2 = sonewconn
        so2→so_remote_sid = packet→sid
        if (so→so_newconn_mirror)
        {
                accept_associate in allowed(so2→so_remote_sid,so)
                so2→so_sid = so2→so_remote_sid
        }
        else if (so→so_newconn_sid)
                so2→so_sid = so→so_newconn_sid;
        else
                so2→so_sid = so→so_sid;
        if (client_associate in allowed(so2, so2→so_remote_sid))
                so2→so_dirflags = TCPF_IN | TCPF_OUT
```

else if (unidir_client_associate in allowed(so2, so2→so_remote_sid))

so2→so_dirflags = TCPF_IN

else

drop connection request packet

Receive connection confirm packet on so (client side):

so→so_remote_sid = packet→sid

if (server_associate in allowed(so, so→so_remote_sid))

so→so_dirflags = TCPF_IN | TCPF_OUT

else if (unidir_server_associate in allowed(so, so→so_remote_sid))

so→so_dirflags = TCPF_OUT

else

drop packet and send reset to server

Receive data packet on connected so (server or client):

if (packet→sid ! = so→so_remote_sid)

log a warning

drop the packet

3. udp_input

Receive packet on so:

recv_associate in allowed(packet, so)

4. rip_input

Receive packet on so:

recv_associate in allowed(packet, so)

B.2.5 Network Layer

1. ip_output

psend in allowed(packet,netif)

nsend in allowed(node,netif)

send in allowed(packet,node)

2. ip_input

   preceive in allowed(packet,netif)

   nreceive in allowed(node,netif)

   receive in allowed(packet,node)

APPENDIX C

NEW SYSTEM CALLS

C.1 `socket_secure()`

Synopsis

`#include <sys/socket.h>`

`int` **socket_secure**(*int domain*, `int` *type*, `int` *protocol*, `security_id_t` *sid*);

Description

By default a socket created by the socket call will inherit the SID of the creating process. This socket_secure interface is provided to permit a process to specify the SID of the new socket. A `create` check is performed to ensure that the process is allowed to create a socket with this sid.

Parameters

domain: The **domain* parameter specifies a communications domain within which communication will take place. This selects the protocol family. Refer to the `socket`(2) call for details.

type: The socket has the indicated *type*, which specifies the semantics of communication. Refer to the `socket`(2) call for details.

protocol: The protocol specifies a particular protocol to be used with the socket. Refer to the `socket`(2) call for details.

sid: This is the specified sid. If the call is successful, this will be the sid of the created socket.

RETURNS

A -1 is returned if an error occurs; otherwise the return value is a descriptor referencing the socket.

C.2 `connect_secure()`

SYNOPSIS

`#include <sys/socket.h>`

`oskit_error_t` **connect_secure**(*int s*, `struct` *sockaddr *name*, `int` *namelen*, `security_id_t` *dst_so_sid*);

DESCRIPTION

This call permits a process to specify a desired remote socket SID. This could be useful when sockets with different sids are bound to the same port on the remote host. The destination node is responsible for enforcing this restriction. The desired destination socket security context for a packet will be a part of the security association.

PARAMETERS

s: *s* is a file descriptor referring to a socket

name: *name* refers to the address of the remote socket.

namelen: The size of the `sockaddr` structure passed in *name*.

dst_so_sid: This is the specified destination socket sid.

RETURNS

If the connection or binding succeeds, 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in errno.

C.3 `listen_secure()`

Synopsis

`#include <sys/socket.h>`

`int` **listen_secure**(*int s*, `int` *backlog*, `security_id_t` *so_sid*, `int` *mirror*);

Description

By default, the new server-side socket created by a connection to a listening socket will inherit the SID of the listening socket. The `listen_secure` interface is provided to permit a process to specify a SID for any sockets created as a result of subsequent connections to the listening socket. In this case, an accept_associate check is performed during the listen_secure operation to verify that a new socket with the specified SID may be associated with the listening socket. The listen_secure interface also provides a mirror flag which may be used by processes to indicate that the SID of the remote socket should be used for the new local socket. In this case, an accept_associate check is performed when the new connection is received, to verify that a new socket with the remote socket's SID may be associated with the listening socket.

Parameters

s: *s* is a file descriptor referring to a socket.

backlog: The backlog parameter defines the maximum length the queue of pending connections may grow to.

so_sid: This is the sid specified for any sockets created as a result of subsequent connections to the listening socket.

mirror: If this flag is nonzero, the new socket created will have the sid of the remote socket.

Returns

A 0 return value indicates success; -1 indicates an error. If the mirror flag is nonzero and a sid is also specified, -1 is returned to indicate error.

C.4 `sendto_secure()`

Synopsis

`#include <sys/socket.h>`

`int` **sendto_secure**(*int s*, `const` *void *msg*, `int` *len*, `int` *flags*, `const` *struct sockaddr *to*, `int` *tolen*, `security_id_t` *pkt_sid*, `security_id_t` *dst_so_sid*);

Description

By default, a packet sent via a nonstream socket inherits its SID from the socket. This interface permits the process to specify a SID for a packet. A `send_associate` check is performed to ensure that this socket is allowed to send this packet.

This call also permits a process to specify a desired remote socket SID. This could be useful when sockets with different sids are bound to the same port on the remote host. The destination node is responsible for enforcing this restriction. The desired destination socket security context for a packet will be a part of the security association.

Parameters

s: *s* is a file descriptor referring to a socket.

msg: *msg* is a pointer to the buffer which contains the data to be sent.

len: *len* is the length of the buffer pointed to by *msg*.

flags: *flags* indicate the flags associated with the data. Refer to `sendto`(2) for details.

to: *to* refers to the address of the remote socket.

tolen: The size of the `sockaddr` structure passed in *to*.

pkt_sid: This is the specified packet SID.

dst_so_sid: This is the specified destination socket sid.

RETURNS

The call returns the number of characters sent, or -1 if an error occurred.

C.5 `recvfrom_secure()`

SYNOPSIS

`#include <sys/socket.h>`

`int` **recvfrom_secure**(*int s*, [out] `void` `*`*buf*, `int` *len*, `int` *flags*, [out] `struct` *sockaddr *from*, [in/out] `int` `*`*fromlen*, [out] `security_id_t` `*`*pkt_sid*);

DESCRIPTION

A process may obtain the SID of a packet received via a nonstream socket by using the recvfrom_secure interface.

PARAMETERS

s: *s* is a file descriptor referring to a socket.

buf: *buf* is a pointer to the buffer which contains the data to be received.

len: *len* is the length of the buffer pointed to by *buf*.

flags: *flags* indicate the flags associated with the data. Refer to `sendto`(2) for details.

from: If *from* is non-nil and the socket is not connection oriented, the source address of the message is filled in.

fromlen: *fromlen* is a value-result parameter, initialized to the size of the buffer associated with from and modified on return to indicate the actual size of the address stored there.

pkt_sid: On return from the call **pkt_sid* contains the sid of the packet received.

RETURNS

This call returns the number of bytes received, or -1 if an error occurred.

REFERENCES

[1] S. E. Minear, "Providing policy control over object operations in a Mach based system," in *Proc. Fifth USENIX Unix Security Symposium*, (Salt Lake City, UT), pp. 141–156, June 1995.

[2] L. Badger, K. A. Oostendorp, W. G. Morrison, K. M. Walker, C. D. Vance, D. L. Sherman, and D. F. Sterne, "DTE firewalls initial measurement and evaluation report," TIS Report #0632R, Trusted Information Systems Inc., Mar. 1997.

[3] D. Futcher, R. Sharp, and B. Yasaki, "Building a multi-level secure TCP/IP," in *Proc. 14th National Computer Security Conference*, pp. 78–87, 1991.

[4] T. Fine and S. E. Minear, "Assuring Distributed Trusted Mach," in *Proc. 1993 IEEE Symposium on Research in Security and Privacy*, (Oakland, CA), pp. 206–218, May 1993.

[5] C. Flink and J. Weiss, "System V/MLS: Labeling and mandatory policy alternatives," in *Proc. 1989 Winter USENIX Conference*, Feb. 1988.

[6] S. Romero, C. Schaufler, and N. Bolyard, "BSD IPC model and policy," in *Proc. 16th National Computer Security Conference*, (Baltimore, MD), pp. 97–106, 1993.

[7] R. Atkinson, "Security architecture for the Internet Protocol," RFC 1825, Internet Engineering Task Force, Aug. 1995.

[8] R. Atkinson, "IP authentication header," RFC 1826, Internet Engineering Task Force, Aug. 1995.

[9] R. Atkinson, "IP encapsulating security payload (ESP)," RFC 1827, Internet Engineering Task Force, Aug. 1995.

[10] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet security association and key management protocol (ISAKMP)," Internet draft, Internet Engineering Task Force, Feb. 1997.

[11] D. Harkins and D. Carrel, "The resolution of ISAKMP with Oakley," Internet draft, IPsec Working Group, July 1997.

[12] S. T. Walker, "Network security: The parts of the sum," in *Proc. IEEE Symposium on Security and Privacy*, pp. 2–9, 1989.

[13] E. Amoroso, *Fundamentals of Computer Security Technology.* Prentice Hall, 1994.

[14] M. Gasser, *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.

[15] T. Dierks and C. Allen, "The TLS protocol," Internet draft, Transport Layer Security Working Group, May 1997.

[16] G. Caronni, H. Lubich, A. Aziz, T. Markson, and R. Skrenta, "SKIP - securing the internet," in *Proc. Fifth Workshop on Enabling Technologies, (WET ICE '96)*, 1996.

[17] C. G. Girling, "Covert channels in LAN's," *IEEE Trans. Software Engineering*, vol. 13, pp. 292–296, Feb. 1987.

[18] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *Proc. 1987 IEEE Symposium on Security and Privacy*, (Oakland, CA), p. 184, 1987.

[19] W. Boebert and R. Kain, "A practical alternative to hierarchical integrity policies," in *Proc. 8th National Computer Security Conference*, (Gaithersburg, MD), pp. 18–27, 1985.

[20] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical domain and type enforcement for UNIX," in *Proc. 1995 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 66–77, May 1995.

[21] Secure Computing Corporation, "LOCKed workstation program LWS expanded environment study report," Technical Report, Secure Computing Corporation, Jan. 1996.

[22] D. L. Sherman, D. F. Sterne, L. Badger, S. L. Murphy, K. M. Walker, and S. A. Haghighat, "Controlling network communication with Domain and Type Enforcement," in *Proc. 18th National Information Systems Security Conference*, p. 211, Oct. 1995.

[23] National Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria.* 1985. DoD 5200.28-STD.

[24] National Computer Security Center, *Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria.* 1987. NCSC-TG-005.

[25] V. Varadharajan, "Network security policy models," in *AUSCRYPT90* (J. Seberry and J. Pieprzyk, eds.), vol. 453 of *Lecture Notes in Computer Science*, pp. 74–95, Springer-Verlag, 1990.

[26] J. McHugh and A. P. Moore, "A security policy and formal top level specification for a multi-level secure local area network," in *Proc. 1986 IEEE Symposium on Security and Privacy*, pp. 34–39, 1986.

[27] W.-P. Lu and M. K. Sundareshan, "A model for multilevel security in computer networks," *IEEE Trans. Software Engineering*, vol. 16, pp. 292–296, June 1990.

[28] Secure Computing Corporation, "Secure computing Sidewinder security server - product white paper." Available as http://www.securecomputing.com/SWFwhitepaper.pdf.

[29] Trusted Information Systems, "Gauntlet functional summary." Available as http://www.tis.com/docs/products/gauntlet/tech.html.

[30] T. J. Fraser, M. J. Petkac, W. G. Morrison, M. L. Badger, B. Uecker, E. L. Cohen, J. Grillo, K. A. Oostendorp, K. C. Djahandari, T. P. Horvath, C. Ko, D. L. Sherman, and C. D. Vance, "DTE firewalls phase two measurement and evaluation report," TIS Report #0682, Trusted Information Systems Inc., July 1997.

[31] Security First Technologies, "A white paper on HannaH and network security." Available as http://www.secureware.com/products/hannah/brochure/hannahpaper.html.

[32] A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL protocol," Internet draft, Transport Layer Security Working Group, Nov. 1996.

[33] O. S. Saydjari, S. J. Turner, D. E. Peele, J. F. Farrell, P. A. Loscocco, and W. Kutz, "Synergy: A distributed trusted microkernel-based operating system," Technical Report, Department of Defense, Mar. 1993.

[34] S. D. Smalley, "A network access control model for Flask," Technical Report, U.S. Department of Defense, July 1997.

[35] D. Eastlake and C. Kaufman, "Domain name system security extensions," RFC 2065, Internet Engineering Task Force, Jan. 1997.

[36] V. Voydock and S. Kent, "Security mechanisms in high level network protocols," *ACM Computing Surveys*, pp. 135–171, June 1983.

[37] S. M. Bellovin, "Problem areas for the IP security protocols," in *Proc. Sixth Usenix UNIX Security Symposium*, July 1996.

[38] R. J. Atkinson, D. L. McDonald, B. G. Phan, C. W. Metz, and K. C. Chin, "Implementation of IPv6 in 4.4 BSD," in *Proc. USENIX 1996 Annual Technical Conference*, Jan. 1996.

[39] P. Metzger and W. Simpson, "IP authentication using keyed MD5," RFC 1828, Internet Engineering Task Force, Aug. 1995.

[40] P. Metzger, P. Karn, and W. Simpson, "The ESP DES-CBC transform," RFC 1829, Internet Engineering Task Force, Aug. 1995.

[41] R. J. Atkinson, D. L. McDonald, B. G. Phan, C. W. Metz, and K. C. Chin,

"A socket-based key management API (and surrounding infrastructure)," in *Proc. USENIX 1996 Annual Technical Conference*, Jan. 1996.

[42] D. Piper, "The Internet IP Security Domain of Interpretation for ISAKMP," Internet draft, Network Working Group, Cisco Systems, Feb. 1997.

[43] Naval Research Labs, "NRL IPv6+IPsec software distribution alpha release 5 (March 1997)." available as http://web.mit.edu/network/isakmp/nrlkmp-Mar97.html.

[44] Cisco Systems Inc., "Cisco systems ISAKMP distribution version 0.5." available as http://web.mit.edu/network/isakmp/ciscokmp.html.

[45] S. Mittra, "Iolus: A framework for scalable secure multicasting," in *Proc. ACM SIGCOMM '97*, (Cannes, France), pp. 277–288, Sept. 1997.

[46] Portland State University, "The Portland State University secure mobile networking project." available as http://www.cs.pdx.edu/research/SMN/ index.html.

[47] D.Denning, "An intrusion detection model," in *Proc. 1987 IEEE Symposium on Security and Privacy*, 1987.

[48] N.MacAuliffe, "Is your computer being misused? A survey of current intrusion detection system technology," in *Proc. Sixth Annual Computer Security Applications Conference*, 1990.

[49] S. I. Schaen and B. W. McKenney, "Network auditing: Issues and recommendations," in *Proc. Seventh Annual Computer Security Applications Conference*, pp. 66–79, 1991.

[50] L. T. Heberlein, K. N. Levitt, and B. Mukherjee, "A method to detect intrusive activity in a networked environment," in *Proc. 14th National Computer Security Conference*, pp. 362–371, 1991.

[51] B. Sommerfeld, "Inline keying within the ISAKMP framework," Internet draft, IPSEC Working Group, Mar. 1997.

[52] S. Kent, "U.S. Department of Defense security options for the Internet Protocol," RFC 1108, Internet Engineering Task Force, Nov. 1991.

[53] M. S. Johns, "Draft revised IP security option," RFC 1038, Internet Engineering Task Force, Jan. 1988.