The following paper was originally published in the
Proceedings of the 7th USENIX Security Symposium
San Antonio, Texas, January 26-29, 1998

A Comparison of Methods for Implementing Adaptive Security Policies

Michael Carney and Brian Loe
Secure Computing Corporation

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Comparison of Methods for Implementing Adaptive Security Policies

Michael Carney
Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113
e-mail: carney@securecomputing.com

Brian Loe
Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113
e-mail: loe@securecomputing.com

Abstract

The security policies for computing resources must match the security policies of the organizations that use them; therefore, computer security policies must be *adaptive* to meet the changing security environment of their user-base. This paper presents four methods for implementing adaptive security policies for architectures which separate the definition of the policy in a Security Server from the enforcement which is done by the kernel. The four methods discussed include

- reloading a new security database for the Security Server,
- expanding the state and security database of the Security Server to include more than one mode of operation,
- implementing another Security Server and handing off control for security computations, and
- implementing multiple, concurrent Security Servers each controlling a subset of processes.

Each of these methods comes with a set of tradeoffs: policy flexibility, functional flexibility, security, reliability, and performance. This paper evaluates each of the implementations with respect to each of these criteria. Although the methods described in this paper were implemented for the Distributed Trusted Operating System (DTOS) prototype, this paper describes general research, and the conclusions drawn from this work need not be limited to that development platform.[1]

[1] This work was supported by Rome Laboratory contracts F30602-95-C-0047 and F30602-96-C-0210. Portions of the DTOS Overview found in Section 4 appeared in [SKTC96].

1 Introduction

Real organizations do not have static security policies. Rather, they have dynamic policies that change, either as a matter of course, or to allow them to react to exceptional circumstances. The computing resources of these organizations must reflect the organization's need for security while affording users the flexibility required to operate in a changing environment.

Any implementation of adaptive security presents its own set of advantages and disadvantages. While this paper compares four methods for implementing adaptive security policies, it is important to keep the needs of the organizations in mind in order to adequately compare implementations of adaptive security. Section 2 outlines some of the possible scenarios requiring adaptive security policies and provides a number of examples of adaptive policies that are useful to the later discussion. Section 3 describes the range of possible implementations for adaptive security given the basic security architecture of the DTOS prototype and provides a brief sketch of the implementations discussed in Section 5. Section 4 provides more background on DTOS, which was used to implement each of the four methods described in this paper. Section 5.1 describes the criteria against which implementations of adaptive security may be measured. The final subsections of Section 5 describes in greater detail the four specific implementations researched at Secure Computing Corporation and evaluates each with respect to the criteria from Section 5.1.

2 Motivating Examples for Adaptive Security

The first example of adaptive security consists of organizations that need to change their policies at regular intervals. For example, a bank may have one security policy enforced during business hours and another policy enforced after hours. The business hours policy would grant broad sets of permissions to various sets of employees in order complete normal banking transactions; however, a more restrictive policy would be in effect after hours to prevent system users from altering banking data in unintended ways.

Some organizations may need to release sensitive documents at specific times. For commercial organizations it may be a press release of new product information that must not be available from the webserver until a specified time. Military organizations may have similar needs to make information available to allies on a timed-release basis. Conversely, today's commercial partner or military ally may be an tomorrow's adversary, in which case they should not be allowed to receive various forms of information.

Other organizations may need to adapt their security policies based on the tasks performed by the users. For example, in the banking example cited above, some tasks may be critical to perform despite the more restrictive policy enforced after 5:00 PM. High-priority or urgent tasks may need to be granted special permissions to complete ongoing operations despite the general change of policy. Other task-based policies may make use of an assured pipeline, like that proposed by Boebert and Kain [BK85]. Assured pipelines address situations in which a series of tasks must be performed in a particular order and the control flow must be restricted. An adaptive policy might change the set of permissions associated with a single process as it completes a series of operations. As the process completes one operation, the permission set changes to allow the process to complete the next operation but to prevent it from revisiting any objects that it needed for earlier operations. A related security policy would be the Chinese Wall introduced by Brewer and Nash [BN89], which is intended to prevent conflicts of interest in commercial settings. Briefly, under a Chinese Wall security policy a subject may initially be allowed permission to an entire class of objects, but as soon as the subject accesses one element of the class, permissions to access any other object of that class are denied.

Role-based security policies form another class of adaptive security policies. A role is distinguished from a task in that an individual has an on-going need to complete a set of tasks. (See [SC96], [FCK95], and [Hof97].) In commercial settings, roles may be used to enforce separation of duties [CW87]. For example one role may be granted authority to issue purchase orders while another has authority to pay for those purchases. However, for small companies it may be necessary for one individual to perform actions in more than one role, though not necessarily at one time to provide the proper controls and oversight. In military operations it may be necessary for an individual to perform actions in more than one role simultaneously. For example, in the Navy the role of the Watch Officer on a ship may be performed by the Chief Engineer. This person may need to fulfill both roles simultaneously. Similarly, the Command Duty Officer may need to perform actions reserved for the Commanding Officer in times of emergency. Privilege to invoke these dual roles should be reserved for extreme situations.

Multi-level security (MLS) rules as applied in the military and intelligence communities form a final class of examples of security policies that must be adaptive. Adaptive policies may allow either a relaxation or selective hardening of confidentiality restrictions. Under MLS rules all objects are labeled according to the sensitivity of the data they contain (e.g., Top Secret, Secret, Confidential, and Unclassified). By the simple security rule, users and subjects are allowed access to observe objects only if their clearance level is equal to or exceeds the sensitivity of the object (see [BL73]). During an emergency it may be necessary to consolidate levels into two levels: one for Secret and Top Secret files, and another for the remainder. Thus, under the relaxed rules, someone formerly cleared for Secret could access files formerly labeled as Top Secret. For example, military officers may only have clearance to the Secret level, but once their troops are under fire, they may need to access Top Secret information such as the location or capabilities of enemy forces. Conversely, confidentiality rules and other security measures could be "hardened" based on DEFCON alert status or following detection of a possible intrusion. There are a number of ways to "harden" a system. For example, one could increase internal controls, perform full audits rather than selective audits, or require additional authentication measures.

3 Implementation Space

The DTOS prototype provides a security architecture that separates the enforcement of the security policy from its definition. Since this type of security architecture is not unique to the DTOS prototype, results from this paper apply to a variety of systems with similar architectures.

Elements available to adapt the security policy include the following:

- the number or complexity of the databases that a Security Server uses to initialize its internal state
- the number of Security Servers available to the microkernel for security computations
- the control over which Security Server makes security computations on behalf of the microkernel

Although the number of possible implementations is large, this paper describes the following representative implementations:

- One Security Server and multiple databases — adapting the policy by forcing the Security Server to re-initialize from a new security database.
- One Security Server and one database — adapting the policy by expanding the internal state of the Security Server and increasing the complexity of the security database to describe more than one set of security policy rules and by providing the Security Server with a mechanism for changing its mode of operation.
- Multiple Security Servers with a single active server providing one point of control over security computations — adapting the policy by providing a mechanism to hand off the responsibility of computing access decisions from one server to another. Thus, one and only one Security Server defines the policy at any given time.
- Multiple, concurrent Security Servers with responsibility for security computations partitioned by tasks — adapting the policy by assigning a pointer to a specific Security Server to each new process. In this method, whenever a process makes a request to the microkernel for service, the microkernel submits requests for access computations to the Security Server that is associated with that process and which defines the security policy with respect to that process.

4 DTOS Overview

This section provides an overview of DTOS, the operating system used to implement the four methods discussed in this paper.

DTOS was designed around a security architecture that separates enforcement from the definition of the policy that is enforced. This architecture allows the system security policy to be changed without altering the enforcement mechanisms. The policy is defined as a function that maps a pair of security contexts to a set of permissions. Each pair of security contexts represents the security context of a subject and the security context of an object that the subject attempts to access. Currently, DTOS implements security contexts consisting of level, domain, user, and group, but the set of attributes that form a security context is configurable. Enforcement consists of determining whether the permissions specified by the policy are adequate for an access being attempted. The generality of the DTOS security architecture has been studied as part of the DTOS program [Sec97]. The conclusion of this study is that a large variety of security policies, useful for both military and commercial systems, can be implemented.

The basic DTOS design consists of a microkernel and a collection of servers. The microkernel implements several primitive object types and provides InterProcess Communication (IPC), while the servers provide various operating system services such as files, authentication, and a user interface [FM93, Min95]. Of particular interest is a *Security Server* that defines the policy enforced by the microkernel and also possibly by other servers. When a request is made for a service provided by the microkernel, the microkernel sends identifiers for the security contexts of the subject and object to the Security Server. These identifiers are referred to as *security identifiers* or *SID's*. A context contains attributes about a subject or object that are necessary for making security decisions. For example, the context may contain the domain of a subject or the type of an object, or the level of a subject or object. The information that makes up the context is dependent on the policy. The actual contexts are

local to the Security Server and are not available to the microkernel. The Security Server then computes permissions for the context pair, as defined by the policy that it represents, and replies to the microkernel. The microkernel is ignorant of the context of each entity since it only enforces the permissions that the ß computes on its behalf. Finally, the microkernel determines if the permissions required for the request were present in the reply. Other servers can communicate with the Security Server in a similar fashion.

For example, a Security Server implementing an MLS policy might maintain subject and object contexts consisting of a level. For the microkernel to enforce the simple security and *-property of the Bell and LaPadula model [BL73], the Security Server will grant a write permission only if the level for the object security identifier dominates that of the level for the subject security identifier, and it will grant read permission only if the level for the subject identifier dominates that for the object identifier. Both permissions may be granted if the levels are equal. A file server would check for write permission before allowing a request to alter a file. An alternative Security Server might provide UNIX-like access controls by maintaining a user and a group for each subject context and an owner, group, and access control bits for each object context. This type of Security Server will grant permissions based on the access control bits depending on whether the user in the subject context matches that of the owner and whether the groups match.

A prototype DTOS microkernel and Security Server has been built by Secure Computing. The microkernel is based on Mach, developed at Carnegie Mellon University [Loe93, Ras91]. A version of the Lites UNIX emulator, modified by the government, provides secured UNIX functionality.

The object types implemented by the microkernel include *task*, *thread*, and *port*. Tasks and threads represent the active subjects, or processes, in the system. Each task has a security context that is used for security decisions involving that task. The state of each task includes virtual memory consisting of a set of disjoint memory regions, each of which is backed by a server that is used to swap pages of the region in and out of physical memory. Each task contains a collection of threads, each of which is a sequential execution, that share the task's virtual memory and other resources. A server is implemented as one or more tasks.

The ports are unidirectional communication channels that the tasks use to pass messages. Tasks use *capabilities* to name ports, and these are kept in an IPC name space on a per task basis. Each capability specifies the right to either receive from or send to a particular port. These capabilities may be transferred to another task by sending a message. For each port there is exactly one receive capability. Therefore, at most one task can receive messages from the port. IPC is asynchronous in that messages are queued in the port and the sending task does not wait until its message has been received. An exception is when the microkernel is the receiving task, in which case the sender waits until the microkernel finishes processing the message.

Sending or receiving a message is a Mach microkernel operation to which DTOS has added security controls that enforce the security policy. Thus, possession of the appropriate capability for a port is necessary but not sufficient in order to send or receive a message from that port. The security contexts of the task and the port must also permit the operation. The policy also constrains what capabilities may be passed in a message sent or received by a task.

The Security Server receives requests from the microkernel through the *microkernel security port* and from other servers through a *general security port*. Requests contain four elements:

- an operation identifier — allowing the Security Server to specify history-based policies that depend on the sequence of operations made on an object,
- a subject security identifier (SSI) — representing the security context of the subject,
- an object security identifier (OSI) — representing the security context of the object, and
- a send capability for a reply port.

The Security Server replies by sending the permissions for that pair to the reply port (Figure 1). Not shown in this figure is the fact that the Security Server both defines and enforces a policy for the requests that it receives. It might allow security determination requests from some subjects, but not from others. Similarly, it might allow security determination requests from a particular subject only for certain (SSI,OSI) pairs.

Security enforcement as described above would be very expensive due to the large number of messages that must be exchanged between the microkernel and the Security Server. The solution in DTOS is

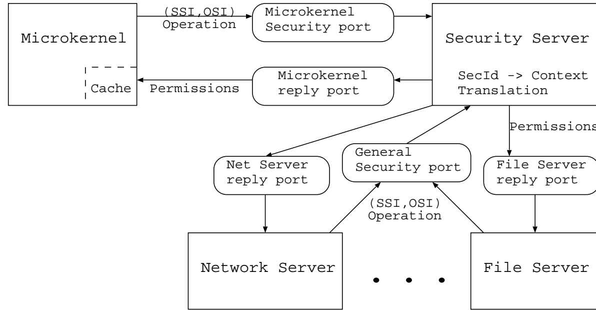

Figure 1: Security Server Interaction

to cache (SSI,OSI) pairs with their permissions in the microkernel [Min95]. When the microkernel receives a request, it first looks in the cache for the appropriate (SSI,OSI) pair. If that pair is in the cache, the microkernel uses the cached entries. Otherwise, it sends the pair to the Security Server to determine the permissions, usually also caching the reply. (Part of the permission set returned is permission to cache the reply — caching would not be permitted for permissions granted for a single operation by a dynamic policy.) Since sending to and receiving from a port are microkernel operations controlled by the policy, the cache must be preloaded with permission for the Security Server to send and receive from the designated ports.

In order to implement a different policy (either by changing the current ß or by referring to a new ß) there must be a mechanism for flushing permissions from the microkernel's cache. Otherwise, if the new policy removes permissions from the system for a specific (SSI, OSI) pair, and the microkernel has already cached the permissions for that pair, then the microkernel would continue to enforce the old policy rather than consult the ß defining the new policy. Therefore, the ß must issue a command to the microkernel, and any other servers registered as caching permissions determined by the ß, telling it to flush its cache. However, it would be impractical for the microkernel to flush every permission in its cache; if it did, then the entire system would come to a halt. Therefore, some permissions are hard coded. These include some of the basic permission required for IPC between the subjects comprising the operating system itself.

The separation between policy and enforcement in the DTOS prototype make it attractive for studying adaptive security. The work described in this report discusses refinements to the design that are important for these policies.

5 Comparison of Implementations

This section of the paper describes each of the methods for changing the security policy in greater detail along with the capabilities and limitations presented by each. However, we begin by describing the criteria against which the four implementation methods are evaluated in Section 5.1. The methods themselves are described in Sections 5.2 through 5.5

5.1 Criteria for Evaluation

An adaptive security policy for a computer system must have the flexibility to meet the security requirements of the organization that fields the system. There are two types of flexibility to consider:

- Policy flexibility — the range of policies that a system can support before and after a transition between policies.
- Functional flexibility — the ability of users to complete tasks despite the transition of policies.

However, greater flexibility may come at the expense of security, and the greater complexity required for some types of transitions may also have an impact on the reliability of the system.

The criteria identified here are not independent of one another; in fact, examining various implementations of adaptive security leads to a series of trade-offs with respect to these criteria. The conclusions that are drawn from the analysis of the four implementations reflect the nature of the dependence of the criteria upon one another.

Policy Flexibility In the context of adaptive security, the concept of policy flexibility could be measured by the amount of change one is allowed to make and whether the system can enforce an arbitrary new policy. Thus, policy flexibility depends on the number (or lack) of constraints that must be satisfied by the successor policy for a given predecessor policy.

Functional Flexibility Functional flexibility addresses whether the policy transition is graceful or harsh with respect to the applications that are running at the time of the transition. A harsh transition might be like turning off the power and re-booting the system, whereas a graceful transition may appear seemless to the user and most applications on

the system. A harsh policy transition may prevent users from performing necessary, possibly urgent, tasks, rather than allowing them to complete their tasks in an evolving security environment. The ideal is to allow necessary tasks to complete while terminating tasks that are not only disallowed under the new policy, but which represent a security risk in the new environment.

Security The existence of a mechanism or method of changing policies may introduce security vulnerabilities. In assessing a method of policy adaptation, one must consider the security risks that are inherent in that method. Furthermore, each type of policy transition must be assessed for the relative difficulty of providing formal assurance evidence in support of the policy transition.

Reliability Each method of policy transition introduces a measure of complexity into the system. Changing policy may expose the system to certain risks which decrease the stability of the entire system.

Performance The ability to change policies quickly has impact on the needs of the user for security, functionality, and reliability. A complex hand-off may allow greater flexibility between policies enforced before and after the transition, but it may also present greater security risks. A less complex hand-off may provide performance gains at the expense of functional grace or flexibility.[2]

5.2 Loading A New Policy Database

One possible method for implementing a new security policy is to change the way that the Security Server defines it by creating a second database and re-initializing the Security Server. A method for doing this existed on the DTOS prototype. During the boot process, the microkernel operates on a hard-coded cache of permissions until the Security Server is ready for operation. Once the Security Server has initialized, the microkernel places the command **SSI_load_security_policy** on the security port of the Security Server. This command causes the Security Server to read the security database to construct a table in its internal memory that maps SSIs and OSIs to permissions. The Security Server then tells the microkernel to flush its cache of permissions, and from that point onward, the policy defined by the Security Server is the policy enforced by the microkernel. The same command can be used to replace one table with another. Once the Security Server has loaded the new policy, it tells the microkernel to flush its cache, and the new policy is enforced by the microkernel.

The command to reload the policy can be encapsulated in a user-invoked program or in some automated process which changes the policy at the triggering of some event. Thus, the policy can be changed at regular intervals using a process like the UNIX utility *cron*, or by a background process which monitors the system for intrusion events.

Policy Flexibility This method relies heavily on the tables that can be loaded into the Security Server from the security database. Since the tables are indexed by the SSI and OSI, the management of the system is easiest if the Security Server loads a new policy which is similar to the old one; thus limiting the granularity of the allowable policy changes. A radical change of policy requires that each entity in the system have a security context which can be recognized by the active Security Server both before and after the policy change.

For initial policies based on Type Enforcement[3] (see [BK85]) or MLS access rules, it would be difficult to make radical changes in the policy. Every entity that has a type or domain associated with it must also have the attributes necessary for enforcing the different policy. Thus, to change from a Type Enforcement policy to a UNIX-like security policy, it would be necessary for objects and processes to have attributes necessary for both security mechanisms. For objects it is necessary to maintain contexts for the sensitivity level of the object as well as the users and groups which may have access to the object. For subjects, it is necessary to maintain the clearance level of the subject as well as the user of the subject. It is also necessary to maintain a database listing the group membership.

[2] Performance seemed to a natural criterion to include. Unfortunately, the performance data is incomplete. Despite this problem, the authors have chosen to include partial data although it is somewhat inconclusive.

[3] A thumbnail sketch of Type Enforcement describes it as a type of mandatory access control policy in which each object has a security attribute known as a *type* and each subject has an attribute known as a *domain*. Subjects are granted access to read, write, or execute objects based on the domain-type pair. Roles can be constructed for users by forming sets of domains in which users may have subjects operating.

Functional Flexibility Since the transition between policies during the loading of a new policy is nearly atomic, this implementation is quite harsh on running applications. Any application which ceases to have permission to perform any task under the new rules is essentially orphaned. This abrupt change of behavior is probably acceptable, and may even be desirable in some contexts (e.g., military emergencies). However, in some contexts this abruptness would cause considerable difficulty. In the banking example presented in Section 1, there may be occasions when a particular user must complete a specific transaction before the end of the day. However, if the policy transition time occurs at 5:00 PM sharp and the user needs an additional fifteen to twenty minutes to complete the task, then the policy may hinder bank employees from completing vital tasks. This would doubtlessly be unacceptable under this scenario.

Security The immediacy of the transition of this method provides for the greatest security; the users always know exactly which policy is the current policy. As will be shown below, this is not always the case with other methods.

Although the security database is a critical object that should be protected from unauthorized modification, the security database could be changed while the system is in operational mode. Assuming that the system is fielded with adequate physical and procedural security constraints, the security database is more susceptible to replacement or modification during operation than the database (and system) would be to attacks conducted between successive boots of the system. If subverted software could replace the intended database with a different file, the system would enforce the wrong policy.

The issue of who can authorize, authenticate, and execute the policy change is a clear security concern. In the DTOS prototype, authority to reload the security policy is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain processes, to roles, or to sets of individuals with other security mechanisms.

As for the assurance of such a system, there would be some concerns about the flow of information across transitions. Consider the following example: Suppose that under one policy a subject in domain D_1 has no permission to observe data in objects of type T_1 but does have permission to observe data in objects of type T_2. Furthermore, no domain that may observe objects of type T_1 may write to any object of a type that D_1 may observe (e.g., T_2). Under this policy alone, there is no possible flow of information from type T_1 to domain D_1. Under a second policy, subjects in domain D_2 may observe data in T_1 and write to objects of type T_2, but D_1 is no longer authorized to observe objects of type T_2. Under the second policy, no information may flow from T_1 to D_1. However, after a transition from the second policy to the first, it would be possible for a subject in domain D_2 to pass information from objects of type T_1 to a subject in domain D_1.

Similarly, when dealing with MLS policies and dynamic security lattices, one is necessarily concerned about loss of confidentiality and potential contamination of files during periods of relaxed security. Returning to a more stringent MLS policy or changing a policy using Type Enforcement requires extensive audit to effect such "roll backs." Unfortunately, these concerns exist for all methods.

While there do not appear to be any theoretical frameworks nor any tool support for conducting covert channel analyses (and simialr analyses) for adaptive security policies, some of the formal modeling and proofs might be relatively easier for a policy in which the database is simply reloaded than for more complex policy transitions.

Reliability A tangible concern is that if the database file has become corrupted, then the Security Server will not be able to read it. The effect of this is that the Security Server dies, and the system is left without any Security Server at all. Not only would the system not be able to enforce the new, intended policy, but the system would have difficulty running at all. The microkernel and other processes that can cache permissions computed by the Security Server would rely solely on the permissions that had been cached up to the time that the Security Server went down.

Both the security and reliability concerns could be ameliorated by placing a checksum (or a digital signature) over the security database. The Security Server would not read in the new database unless the checksum can be verified.

Of course the discussion in the previous two paragraphs assumes that the specification of the new database is correct. Even if the policy changes are small, an entirely new database must be constructed, and it must be correct to avoid problems (e.g., either a corrupt file or deadlock conditions between the Security Server and microkernel). Since

it can be difficult to specify one database correctly, attempting to make more extensive changes reduces the reliability of this method.

Performance This is the second fastest method for changing policies. During performance testing, a typical transition time (median) required 2.985 seconds, and no transition required more than 3.970 seconds. Although this might not be as fast as necessary in a real-time embedded system, this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

5.3 Expanding the Database and Security Server State

In this method of transition between policies, when the Security Server loads its initial security database, all of the permissions allowed under all modes of operation are initialized in the Security Server's internal memory. A mechanism internal to the Security Server allows it to change policy without having to read a new security database. Thus policy changes could be triggered by a variety of events. The policy could change based on several events: the time of day, when a process completes a certain task or invokes a certain permission, or when an alarm is set off (e.g., by a possible intrusion event). This method is similar to the mechanism described in Section 5.2; however, because of the ability to change policies based on triggering event it has a number of advantages which are listed below.

Policy Flexibility This method has the same restrictions that loading a new database has. It is easiest for the Security Server to alternate among policies which are similar. For initial policies based on Type Enforcement or MLS access rules, the new policy must also be based on Type Enforcement or MLS access rules. However, the mechanisms for changing policy definition give this method greater flexibility than the previous method.

For example, for policies which change on a regular periodic basis (recall the banking example in which a more stringent policy is enforced for after-hours operation), a timing mechanism that triggers the change of policy could be added to the Security Server.

Another adaptation mechanism could be triggered by the use of particular permissions. For example, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions. This would render the permission as a one-time only permission. For example, in a commercial application a one-time permission to issue payment for a purchase order would prevent double payment.

Similarly, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions, and one or more could be added. Such adaptations could be chained together. For example, if the Security Server were applying Type Enforcement, a process operating in one domain might be granted access to a new type and denied access to an old one. Thus a set of operations could be performed by a single process in a secure pipeline. Such secure pipelines are already possible with Type Enforcement, but each operation is performed by a separate process, each running in a unique domain (see [BK85] and [GHS97] for more details). This type of mechanism would also be ideal for enforcing the security policy known as Chinese Wall (see [Pfl97] for a definition).

Functional Flexibility Since the transition between policies during the loading of new policy is the most atomic, this implementation could be as harsh on running applications as reloading the database. However, the database could be expanded to include several policies so that a policy transition could take place with several intermediate policies during the transition. A phased transition of this sort might allow some tasks to complete processing within fixed time limits.

Security The security concerns here are the same as in Section 5.2 with the exception that the security database is read once and only once at initialization, and thus the possibility that an untrusted user or process has been able to corrupt it is removed from concern.

With the expanded state of the Security Server, changes of policy may be regulated automatically by the time of day, as in the banking example, or by events, as in the Chinese Wall policy. By moving the authority for changing the policy from subjects to events, the methods by which hostile users could alter the enforced policy change. If a hostile user tampers with the system clock, or forces a triggering event, or counterfeits a triggering event, then he

could control changes of policy.

The ability to "harden" system defenses automatically in the event of a possible intrusion also seems to be a particular advantage not present in reloading the database.

Reliability This method is more reliable than reloading the policy because we are not concerned about the second policy being corrupted after boottime. However, like the previous method, changes to the policy are limited in their granularity by practical concerns. This method makes the coding of the Security Server more complex, which may cause unforeseen problems; so small policy changes, this method would be superior to specifying and reloading an entirely new database. However, one should not be tempted into making too many changes to the policy using this method because of the potential complexity.

Performance Explicit performance numbers are not available for this method. However, since it avoids the time-consuming step of reading a new database, it is anticipated to be faster than reloading the database, and expected transition times should be less than one second. Thus, it is expected to be the fastest of the four methods under discussion.

The microkernel and other processes can cache permissions to improve performance; so changing policy and flushing the cache frequently could cause a minor performance drag. However, permissions in the database can be flagged as non-cachable. Thus, transient permissions as described above could be flagged in that way so that the microkernel would not have to flush its entire cache as it does for reloading the database. Similarly, permissions in the database can be flagged as those which cannot be flushed. Thus, persistent permissions could be flagged so that the microkernel would not have to flush those permissions from its cache at all, and performance would not be adversely affected by the adaptation of policies.

5.4 Handing Off Control to a New Security Server

In the Security Server hand-off, the current Security Server passes the receive capability for its security port to another Security Server that implements a new policy. In order to accomplish this, the new Security Server is initialized while the current Security Server is still in control of the policy decisions. The new Security Server uses the command **get_special_port** to obtain the send right to client port of the current Security Server and then issues the **transfer_security_ports** to the current server. The current Security Server packages the receive rights for its security port along with two other tables of information. One table contains the mapping between security contexts and SIDs that it uses to interpret incoming requests, and the other table lists the ports of processes that may be caching security permissions. The new Security Server needs the former to interpret requests that it receives regarding any processes or objects that exist prior to the hand-off. It needs the latter because it may eventually need to tell these other processes to flush their cached permissions. The last action of the current Security Server is to tell all processes with cached permissions to flush their caches. At this point the new Security Server can compute access permissions, and the microkernel and any other processes that enforce these permissions can enforce the new security policy.

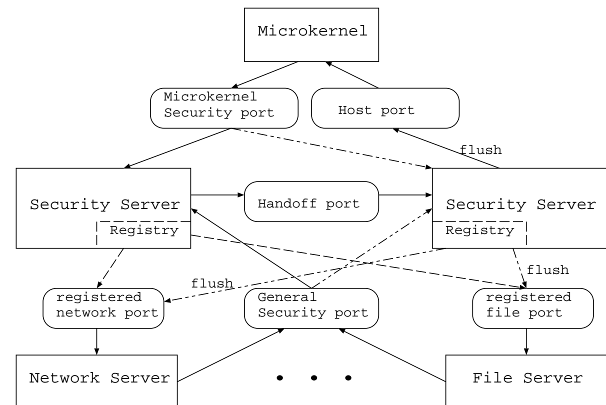

Figure 2: Security Server Hand-Off

In order to be able to process new requests for permission computations, the new Security Server must be able to interpret the requests. As mentioned above, the old Security Server sends the appropriate information for the new Security Server to match contexts to SIDs. However, the new Security Server has some knowledge of security contexts prior to receiving this information from the old Security Server; so it must reconcile its understanding of contexts with the mapping information received from the old Security Server. It also must create new SIDs for any new contexts which were not recognized by the old Security Server. For example,

if the both the new Security Server and old Security Server are implementing Type Enforcement and there are new domains as part of the new policy, the new domain must receive a SID. Similarly, if the hand-off occurs in order to implement dynamic lattices as part of an adaptive MLS policy, any new levels must receive SIDs. Once the new Security Server has completed this reconciliation, the old Security Server can shut down.

Policy Flexibility The greatest strength of the hand-off method is that one can enforce a global, radical change of policy. The new Security Server can implement a very different policy from the one that is enforced before the hand-off. Not only can the new Security Server initialize from a new security database, it can implement an entirely different set of algorithms for making its security computations. This may be especially important for implementing dynamic lattices as part of an adaptive MLS policy.

As discussed in Section 5.2, the only impediment to changing the policy in a radical way is the labeling of objects and processes with the appropriate set of attributes which can be interpreted by both the new and old Security Servers. In other words, radically different policies may require essentially disjoint sets of attributes which the system designers glue together for the context of any single entity.

Functional Flexibility In essence this method is not different from reloading the database. Changes to the security policy are global and atomic. The same problems exist in this method as for reloading the databse for situations where a harsh change of policy is undesirable, as in the banking example.

Security Some of the same security advantages and concerns exist here as for the Reload Policy method. As with the Reload Policy method, the users always know exactly which policy is the current policy. However, if the new Security Server has to initialize from some static file or security database, there is always the risk that it could be subverted. Another possibility is that the code for a new Security Server could be subverted as well and that a malicious Security Server could end up in control of the permission decisions.

There remains the issue of who can authorize, authenticate, and execute the policy change. The Security Server will hand off the security port to the new server when it receives the command *SSI_transfer_security_ports* on its security port. Just as in the case of the authority to reload the policy, the permission to issue this command is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain roles or to sets of individuals with other security mechanisms. The additional concern here is that the security port is transferred to the correct subject, the new Security Server.

Reliability The hand-off is a more complicated procedure than the preceding two methods from two points of view: of the operation itself and of the development of the system.

From the latter point of view, the handoff requires that a second, fully functioning Security Server be implemented, as well as a second security database. This solution should not be used for trivial changes to the policy.

Unfortunately, from the former point of view, the hand-off procedure on the DTOS prototype is relatively delicate, and this is its greatest weakness. While these type of problems are inherent in all four methods, they are more likely in this implementation. The unreliability is an artifact of the Lites server which provides the microkernel with services that allow one to use UNIX applications on DTOS. The combination of the microkernel, Lites server, and the Security Server is prone to paging errors and deadlocks. To avoid these errors, the microkernel must have a sufficient set of permissions hard-coded into its cache (these permissions are not flushed from the microkernel). Some of the permissions required by the new Security Server to complete the hand-off must be in the hard-coded cache before the transition is initiated.

For example, the Security Server has page-able memory. During the hand-off, the Security Server may start using new areas of memory while processing a security request from the microkernel. If a page fault occurs, then the Security Server itself will request service from the microkernel. If the microkernel has not cached the permission required by the Security Server, it must in turn request a security computation from the Security Server. However, the Security Server is blocked on the request to the microkernel for service, and the microkernel cannot complete its request without the security computation from the Security Server. What makes these types of events unpredictable is the existence

of other processes on the system that may request services from the Lites server while the security port rights are in transit. The new Security Server depends on the Lites server for services, but a thread of execution in the Lites server can be waiting for a security computation creating the deadlock.

Performance This is the slowest of the methods tested. During performance testing, a typical transition time (median) required 4.900 seconds, and all transitions fell with the range of 4.820 to 5.010 seconds. This might not be as fast as the Reload Policy method, but once again this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

5.5 Adding Security Servers for New Tasks

The final method for changing the security policy is to create a set of task-based Security Servers. In the three previously described methods, all tasks operate under a single, monolithic policy. With this method there may be more than one Security Server computing access decisions for the microkernel and other clients, each defining a separate set of security rules. While the microkernel is enforcing multiple policies, each task on the system is associated with one and only one Security Server which serves as its primary Security Server. In fact, each task is associated with a list of Security Servers ordered by precedence. There is a well-defined policy for each task because there is only one way for each access request to be computed by the entire set of Security Servers.

For this method we introduce a new global variable: the Security Server stack. Each entry in the stack consists of a data structure containing the security and client ports for each Security Server. At boot time, the initial Security Server uses the **set_special_port** command to enter the security and client ports to the initial entry in the stack. Another global variable, **curr_ss**, points to that entry in the stack to indicate that the initial Security Server is the *current* Security Server. When another Security Server is created, it also enters its ports to the stack at the first available entry, and **curr_ss** is incremented to the next position in the stack.

Each task has a pointer labeled **ss_ptr** that identifies the Security Server that defines the policy under which the task is running. When tasks are created, **ss_ptr** is set to **curr_ss** by default, though the

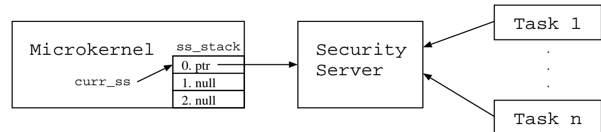

Figure 3: Security Server Stack Before "Push"

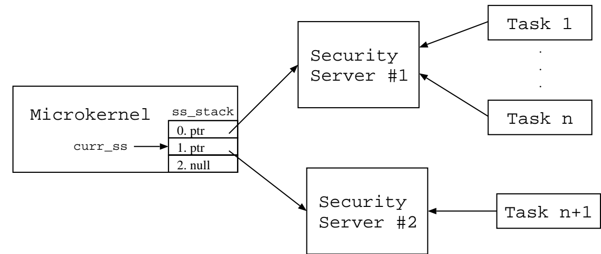

Figure 4: Security Server Stack After "Push"

parent task may cause the value of **ss_ptr** for the new task to be set to the parent's Security Server. Like any other process, each new Security Server itself operates under the policy defined by a Security Server which precedes it in the stack (the Security Server immediately preceding it would be the default). If each Security Server in the stack refers to its immediate predecessor in the stack, then it is truly a stack-like implementation. If the Security Servers in the "stack" refer to servers older than their immediate predecessors, then a graph of the dependencies could be more accurately described as a "tree." This tree-like structure for the dependencies between the Security Servers give this implementation an interesting set of properties.

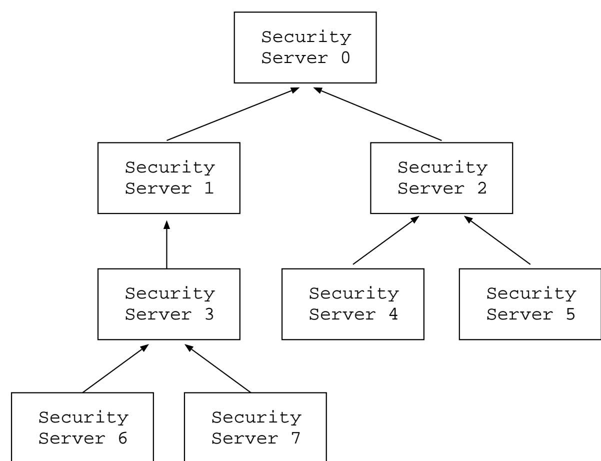

Figure 5: The tree-like dependency among a set of Security Servers

When the microkernel receives a request, it checks

its cache for the permission. If the permission is not in the cache, it sends a request to the Security Server assigned to the requesting task. The Security Server computes the requested security access, unless it receives a request with a context that it does not understand. If the Security Server cannot resolve the SIDs into security contexts, it forwards the request to its own Security Server. The request is passed down the tree until some Security Server, possibly the "root" Security Server, is able to resolve the SIDs into contexts and a security computation can be made.

This method for changing the security policy is the most robust and possibly the most flexible method of the four methods discussed in this paper. However, the additional flexibility and reliability of enforcing multiple security policies may come with an increased cost for assuring the security of the system.

Policy Flexibility This method for changing policy provides the capability for considerable flexibility for changing the policy. However, as new Security Server s are created, only new tasks operate under the new policy rules; so changes to the system-wide policy are local rather than global. In other words: you can't teach an old dog new tricks, because old tasks will continue to run under the policy defined by the old Security Server.

There is the possibility that the stack could be augmented by using one of the other policy changing mechanisms to force old tasks to run under a new policy. For example, if there are two servers in the stack at positions 0 and 1, the Security Server at position 0 could hand off to a third Security Server which is identical to the Security Server in position 1. Thus, both servers operating would define the same policy, and the microkernel would be enforcing only one policy rather than two. [In fact, the first two servers could then exit, all tasks with pointers to the second Security Server would be re-directed to the server at position 0 (the third server), and the system would only have one Security Server as well as one policy.]

Functional Flexibility Functional flexibility is the greatest strength of the Security Server stack. Allowing running processes to run under their original policy is a way of "grandfathering" in their allowed accesses. Thus, in our banking example, if some user is actively working on a process at 5:00 PM which must be completed, but the bank's security policy is set to change to a more restrictive policy at that time, the user would be allowed to continue his task because the task is operating under the less restrictive policy. However, any attempt by a user to create new tasks after 5:00 pm would be subject to the new, more restrictive policy.

Security This method is a double-edged sword. It is possible that certain tasks which need to be highly constrained could operate under more restrictive policies than is generally required. This could be an advantageous design for increasing security. However, coordinating the necessary elements could be a nightmare for system designers and for any attempts to provide formal assurance evidence. In effect there would be multiple, overlapping security policies. One could not make broad global statements about the behavior of the system and the rules in place at any given time.

Also, once a task is granted a permission to perform some operation, it is allowed to keep that permission, even if another more restrictive Security Server is pushed onto the stack. Thus, in the event of an intrusion, a rogue process which has gained unauthorized access to system resources may be able to continue unchecked. Thus, the gains made for functional flexibility allow for a loss of security. In order to harden the defenses of a system like this, it would be necessary to graft another method of policy change on top of this one.

Reliability This method improves upon the hand-off for reliability because there is no vulnerable moment when the rights for the security port are in transit. It is also more reliable than the Reload Policy Method because the top Security Server in the stack will still be able to make security computations even if new Security Servers fail to initialize due to corrupted security databases.

Performance Explicit performance numbers are not available for this method. However, it is anticipated to be as fast or faster than the hand-off, and expected transition times should be between four and five seconds. The greatest factor in the performance for the stack is the loading of the large executable for creating a new server to push onto the stack, which is also true of the hand-off. The hand-off is slower because the rights to the security port have to be transferred from one server to the other. The Security Server stack is quicker than

loading the executable for the new server, but adds an extra wait.

6 Conclusions

There are a number of ways of implementing adaptive security policies for security architectures which separate the definition of the policy from its enforcement. From the entire range of such implementations, this paper has examined four possible methods all of which have been implemented for the DTOS prototype by Secure Computing. Each implementation has strengths and weaknesses, and the trade-offs are encapsulated in Table 1 below. From the table, the server stack and the extended state appear to be the most attractive options for implementing adaptive security, but which choices one makes depends on the eventual application for the implementation as suggested below.

	Implementations			
Criteria	Reload Policy	Extended State	Hand-Off	Server Stack
Policy Flexibility	Fair	Good	Fair	Excellent
Functional Flexibility	Poor	Good	Fair	Excellent
Security	Good	Excellent	Fair	Poor
Reliability	Fair	Excellent	Poor	Good
Performance	Good	Excellent	Poor	Fair

Table 1: Summary of Trade-Offs

When applied appropriately, reloading the policy and the expanded state methods are the lightest weight implementations and provide good features for a narrow set of applications. In particular, the key features of these two methods are that they allow the Security Server to change the database without changing the algorithms from which the Security Server makes its security computations. The database and Security Server implementations for the expanded state have the potential to become complex. The additional complexity posed by this work may make alternate methods for implementation more attractive. The expanded state method is best left to small, incremental changes in the policy. By comparison, reloading the policy is probably not a an attractive option for system in which there are a large number of small changes to the databases since each change of policy would require its own database, and the issue of scalability may be burdensome.

The other two methods, the hand-off and server stack, allow for changes to the algorithms for computing permissions, and this is what accounts for the greater degree of policy flexibility. Because of the multiple points of control, the security server stack offers the greatest functional and policy flexibility, and the inheritance structure of the parent-child relationships between Security Servers offers the ability to grandfather permissions for running applications. However, that very same asset is a liability. Policy changes using the stack are local, not global. Thus, it is not possible to revoke permissions using that method alone. Furthermore, depending on the number of policies supported by the system, the security server stack holds the potential for being the heaviest weight implementation.

Not addressed in Table 1 is the possibility of mixing and matching the four methods described in this paper to capture the best security features of one method with the best flexibility features of another. For example, one might combine the security server stack and with the hand-off method in the following way. Tasks would operate under task-based policies with the server stack up to a certain point in time, allowing for local changes to the policy based on roles and tasks, and then a server might hand off to its parent and shut down. For example, in the banking application in which the more restrictive nighttime policy is the child of the less restrictive daytime policy (i.e., the stricter Security Server is pushed onto the stack at 5 PM), the nighttime server could hand off to its parent the following morning at 8 AM and shut down. Similarly, one might follow the hand-off or server stack by reloading the policy to change the internal tables of a Security Server without changing the fundamental algorithms by which it operates.

Current work on adaptive security has focused on theoretical aspects of adaptive security policies and on various mechanisms for implementing adaptive security. Future work on adaptive security policies should turn from the theoretical to the applied, hopefully by implementing a demonstration system. For example, one might implement a set of banking applications that would operate under policies for daytime and after-hours processing. A demonstration system of this type should also be accompanied by formal assurance evidence such as a formal security policy. However, until there is a real system to examine, formal assurance for adaptive security can only be speculative.

7 Acknowledgments

The authors would like to thank Rome Laboratory for sponsoring the contract under which this research was completed. Thanks also go to Todd Fine, Terry Mitchem, Duane Olawsky, and Ray Spencer for technical assistance, to Dan Thomsen and Joe Andert for comments on the manuscript, and to Cornelia Murphy who served as program manager.

References

[BK85] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, September 1985.

[BL73] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, May 1973.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[FCK95] David F. Ferraiolo, Janet A. Cugini, and D. Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Computer Security Applications Conference*, New Orleans, LA, dec 1995.

[FM93] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[GHS97] Paula Greve, John Hoffman, and Richard Smith. Using Type Enforcement to Assure a Configurable Guard. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997. To appear.

[Hof97] John Hoffman. Implementing RBAC on a Type Enforced System. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997. To appear.

[Loe93] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

[Min95] Spencer E. Minear. Providing policy control over object operations in a Mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[Pfl97] Charles P. Pfleeger. *Security in Computing*. Prentice Hall, Inc., Upper Saddle River, NJ, 2 edition, 1997.

[Ras91] Richard F. Rashid. Mach: A case study in technology transfer. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, chapter 17, pages 411–421. ACM Press, 1991.

[SC96] Ravi S. Sandhu and Edward J. Coyne. Role-based access control models. *IEEE Computer*, pages 38–47, February 1996.

[Sec97] Secure Computing Corporation. DTOS Generalized Security Policy Specification. DTOS CDRL A019, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[SKTC96] Edward A. Schneider, William Kalsow, Lynn TeWinkel, and Michael Carney. Experimentation with adaptive security policies. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1996. Final Report for Rome Laboratory contract F30602-95-C-0047.