Assuring Distributed Trusted Mach

Todd Fine and Spencer E. Minear
Secure Computing Corporation
1210 West County Road E
Suite 100
Arden Hills, Minnesota 55112

Abstract

The Distributed Trusted Mach (DTMach) program is developing a design for a high-assurance, secure, distributed system based on Mach. To achieve this goal, it is first necessary to identify the general threats against which DTMach must protect. The next step is to identify control mechanisms that are sufficient to protect against each of the threats. The DTMach design makes extensive use of type enforcement in addressing the threats. This paper describes the general threats and the countermeasures provided by DTMach. Doing so provides more evidence of the usefulness of type enforcement in general and the high assurance provided by the DTMach type enforcement policy.

1 Introduction

Distributed Trusted Mach (DTMach) is an operating system currently being designed by Secure Computing Corporation. The goal of the project is to use the Mach 3.0 kernel as the base for a secure, distributed system. The DTMach design is an outgrowth of three related efforts: Mach [12], TMach [1, 2], and LOCK^{TM} [11].

As a first step in developing the DTMach security policy, a categorization of general security concerns was constructed. Concerns that were not adequately addressed by the Mach 3.0 kernel indicated potential security vulnerabilities. This paper describes these general security concerns, the manner in which the Mach 3.0 kernel addresses each concern, and the manner in which DTMach addresses each concern. This paper does not describe other aspects of the DTMach design. The interested reader is referred to [8] for a description of the complete design. Although familiarity with the Mach 3.0 kernel and DTMach design is helpful in reading this paper, Section 2 provides a brief overview of Mach and DTMach that contains enough detail to understand the issues discussed in subsequent sections.

DTMach uses *type enforcement* to address a broad range of security threats. Type enforcement is a flexible, general access control mechanism that was initially developed at Secure Computing for use in its LOCK Trusted Computing Base (TCB)[11, 13]. Section 3 provides a brief overview of the concept of type enforcement. On the DTMach project, the existing design for the TMach kernel [1] was modified to support type enforcement and to maintain maximal consistency with the Mach 3.0 kernel. Consequently, there are significant differences between TMach and DTMach. Although references are made to the TMach design, the focus of this paper is on the DTMach security policy and security mechanisms. Space does not permit a complete comparison of the solutions provided by each system, nor would it be appropriate to compare the two systems without further input from the TMach developers.

Section 4 provides motivation for the importance of protecting against more general security threats than those addressed by the conventional mandatory and discretionary access control policies. Next, in Section 5 we describe general classes of security threats and the manner in which DTMach protects against each threat. In Section 6 we sketch the DTMach approach to implementing these counter-measures. Finally, in Section 7 we summarize the main points of this paper:

- DTMach protects against a wide variety of threats in a highly assured fashion.
- Type enforcement is a powerful control mechanism that greatly facilitates the design of high assurance systems.

2 System Overview

This section provides a brief overview of Mach and DTMach. Further details of both systems can be found in [6] and [7]. To highlight the differences between Mach and DTmach, we first describe Mach and then separately describe DTMach extensions to Mach.

2.1 Mach Overview

The central concept in Mach is message passing. A Mach process can send a message to another Mach process by sending a message to a *port* from which the second process receives the message. Sending a message to a port causes the message to be enqueued in a message queue associated with the port. Receiving a message from a port causes a message to be dequeued from the head of the queue. Data in messages can be sent either by value (*in-line*) or by reference (*out-of-line*). Passing data out-of-line allows large amounts of data to be transmitted efficiently between processes. This is accomplished using *copy-on-write* semantics.

In other words, data that is passed out-of-line is not physically copied unless one of the processes accessing the data subsequently modifies the data. As long as processes are only observing the data, they can share a single copy. Once a process modifies the data, the data is copied and the modifications are only made in the copy visible to the modifying process. Copy-on-write semantics make it appear that each process has its own copy of the data while minimizing the amount of copying that actually occurs.

Mach supports multi-threaded processes through *tasks* and *threads*. A task consists of a *port name space*, an address space, and a set of threads. A thread consists of things such as machine registers and an instruction pointer. Threads are the unit of scheduling while tasks provide environments in which threads can execute. Each thread in a given task executes in the environment provided by that task. A process in Mach is a task together with the threads executing within the task. For example, a process reading input from a keyboard and a mouse might be implemented as a task containing a thread handling keyboard input and another thread handling mouse input.

Just as an address space provides access to memory, a port name space provides access to ports. Mach uses *port rights* to control the access a task has to a port. A task cannot receive a message from a port unless its port name space contains a *receive* right for the port. Similarly, a task can only send a message to a port if its port name space contains a *send* or *send-once* right for that port.

A port is created when a task *allocates* the port. The allocating task is given rights to the port. The only way for other tasks to obtain a right for a port is for the right to be *passed* in a message sent by a task already holding the right. Mach port rights are analogous to *capabilities*[9]. Although many tasks are permitted to hold a *send* right to a port, at most one task holds a *receive* right for a port at any given time; this task is the port's *receiver*. Whenever a port's receiver passes its *receive* right to another task, it forfeits its ability to receive messages from the port.

All entities in Mach are represented by ports. In other words, ports are used to provide a uniform name space for services and resources. Except for a small number of requests implemented as traps, all requests in Mach are implemented as messages sent to ports. For example, each task has an associated *task port* that is used to identify the task. Operations on a task are invoked by sending messages to the task port associated with the task. The Mach kernel is the receiver for all task ports. Upon receiving a message through a task port, it performs the necessary processing.

As a specific example, tasks are killed by sending a **task_terminate** message to the associated task port. Upon receiving the request, the kernel terminates the task associated with that port.

An interesting consequence of the decision to use ports to provide a uniform name space is that it is quite simple to interpose tasks between a task and a service or resource. For example, kernel requests invoked on a given task, *Target*, can be rerouted through a debugger, as shown in Figure 1. The kernel records both a *self* port and an *sself* port for each task. While the self port is the actual task port for the task, the sself port can be any port. Whenever a task asks the kernel to provide it the task port for a task *Target*, the kernel returns *Target*'s sself port rather than *Target*'s self port. This feature allows the debugger to interpose by allocating a port P and instructing the kernel to set *Target*'s sself port to P. Although tasks might "believe" they are invoking kernel requests on *Target* when they send messages to P, they are actually sending messages to the debugger. By forwarding the messages received over P to T, *Target*'s self port, the debugger can gather information about kernel requests in a transparent manner.

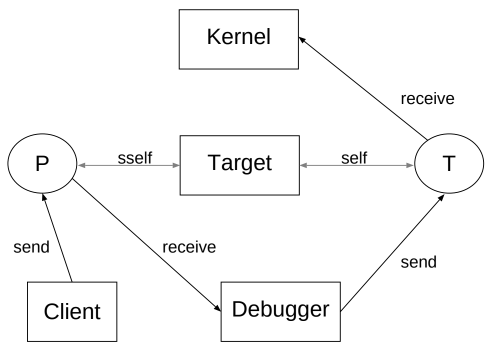

Figure 1: Task Interposing in Mach. All requests sent to P are received by the debugger and forwarded to T.

It is important to understand that Mach is not an operating system, it is a kernel. In other words, rather than providing all of the services typically included in an operating system, Mach provides services that can be used to implement the traditional operating system services. For example, Mach provides facilities to map a file into an address space even though it does not provide a file system. Some mechanism outside the kernel must be used to associate a file with a port. The receiver of this file port is a task that acts as the pager for the file. Then, the port representing the file is provided as a parameter to the Mach **vm_map** request. This causes the kernel to establish a binding between the file's pager and a region of memory in the address space of the task invoking the **vm_map** request. When a page fault occurs, the kernel sends a message requesting data from the pager. When the kernel needs to swap a dirty page from memory, it sends the contents of the page to the pager in a message.

An interesting feature of Mach is that the task serving as a pager does not have to be a system task. In other words, users are free to implement their own pagers. Such tasks are called *user pagers* or *external memory managers*. This allows an operating system emulation or an application to use its own paging strategy rather than restricting it to using a single paging strategy implemented in the kernel.

2.2 DTMach Overview

The DTMach extensions to Mach can be categorized as those that extend Mach to a TCB and those that allow finer access control. As discussed previously, Mach does not provide all of the traditional operating system services. Although things such as file systems and devices are external to the kernel in both Mach and DTMach, in DTMach they must be included in the TCB if they are to be secure.

2.2.1 DTMach Servers

Although there are other servers in the DTMach TCB, the only TCB servers discussed in this paper are name servers, file servers, network servers and security servers. Name servers are used to implement name spaces. To access a service or resource, a task provides the name of the service or resource to a name server. In response, the name server returns a port representing the service or resource. File servers are similar; to access a file, a task first obtains a port representing that file from the file server.

Just as the kernel does not provide things such as file systems, it does not provide any networking capability. The DTMach approach for distributing Mach is based on the approach described in [10]. Tasks called network servers are implemented that transparently extend the Mach message passing mechanism across a network. In this approach, each node in a network must contain a network server. To access a port on a remote node, a task must obtain access to a port on its node that the local and remote network servers bind to the port on the remote node. Figure 2 illustrates the process of sending a message across a network.

1. The sending task sends a message to the local port bound to the remote port.
2. The local network server receives the message from the local port and transmits it through the network to the network server for the node containing the remote port.
3. The remote network server takes the data received through the network and sends it in a message to the remote port.
4. The receiving task receives the message from the remote port.

The fact that the data was transmitted through the network is transparent to the sending and receiving tasks.

While file, name, and network servers are common operating system extensions to Mach, the security servers are unique to the DTMach design. The security servers are responsible for making access control decisions; in this sense, they are analogous to the LOCK SIDEARM[1][11]. Each task and port is assigned a *security context*. When a task attempts to access a port, the kernel determines whether the task

[1]The SIDEARM is a security coprocessor that performs all of the access control computations in the LOCK system.

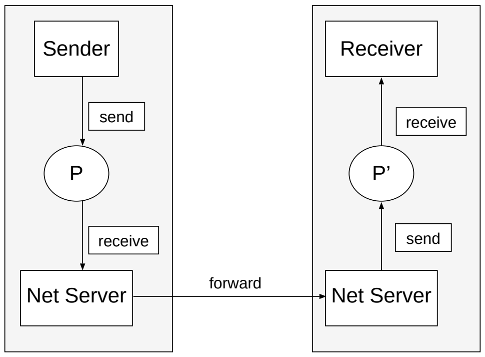

Figure 2: Network IPC in DTMach. Network servers extend IPC across the network.

is permitted to access the port by sending a message containing the relevant security contexts to a security server. In response, the security server sends the kernel a message indicating the accesses allowed by the security policy. For efficiency, the kernel may cache decisions made by a security server. If sufficient information is present in the cache, the kernel does not need to query a security server to determine whether an access is permitted.

One of the key benefits of using a security server to make security policy decisions is that there are few dependencies between the security policy and the kernel. The kernel is responsible for enforcing access decisions, but it relies on the security server to make the access decisions. In particular, the security policy can be changed with minimal or even no changes to the kernel. By adopting this approach, DTMach systems can be used to enforce a rich set of security policies that address both confidentiality and integrity and that can be tailored to fit the needs of the enterprise supported by the system.

2.2.2 Permission Vectors

The major refinement DTMach makes to the Mach access control mechanisms is the separation of port rights and *permissions*. While Mach allows a task to make any use of a port right it holds, DTMach requires a task to have permission in addition to holding the right. For example, a task can send a message to a port only if it both holds a *send* or *send-once* right to the port and a security server indicates that the task has *send* permission to the port. For performance reasons, permission vectors are cached in the kernel once they have been computed. By separating the operational semantics of Mach port rights from the DTMach security policy, it is possible for Mach and DTMach to have the same semantics for port rights and to control the transfer and revocation of port rights in a secure manner.

Send and *receive* are only two types of permissions; permissions are also used to control what kinds of messages a task may send to a port. As a specific example, the DTMach *service vector* represents a subclass of permissions that controls the services that a task may request through a port. For each service that a task may request through a port, the task's service vector for the port contains a bit indicating whether the service is permitted. Since each of the kernel requests represents a kernel provided service, the service vector can be used to control which kernel requests can be issued by each task. For example, suppose that a task $task_1$ needs to terminate $task_2$, but it does not need to perform any other operations on $task_2$. By requiring that the only bit that is set in $task_1$'s service vector for $task_2$'s task port is the bit corresponding to the kernel request **task_terminate**, $task_1$ is prohibited from performing operations other than **task_terminate** on $task_2$. Similarly, the service vector can be used to control the services that a task may request from servers.

Other extensions to provide finer access control include tagging messages with the security context of the sender and (optionally) the security context of the intended receiver.[2] More detail on permission vectors is provided in Section 6.

2.2.3 The DTMach Security Model

Before proceeding with the description of the threats to DTMach and security mechanisms in DTMach, we first provide an overview of the DTMach security model. We follow the traditional approach of describing the security model in terms of the *subjects* and *objects* in the system.

A subject is an active entity in the system. Technically speaking, the only active entities in Mach are threads. In other words, only threads execute instructions on the CPU. Since threads always exist within tasks, we view tasks as subjects. We assign a security context to each task and define the security context of a thread to be the security context of the task in which it is contained. To reduce the dependencies between the kernel and the security server, the contents of a security context are known only to the security server. In the security server currently being developed, the security context contains:

- a level attribute that is used by the MLS (multilevel security) policy
- a domain attribute that is used by the type enforcement policy described in the next section
- a subject identifier that is used by the identity based access control policy

An object is a passive entity in the system. There are three types of objects in DTMach: ports, memory objects, and persistent objects. As described earlier, ports are entities that are maintained and protected by the kernel and that allow unidirectional communication between tasks. Memory objects are also kernel protected entities. Tasks access them through their virtual address spaces. Unlike ports and memory objects, persistent objects are external to the kernel. Although persistent objects are not protected by the kernel, they are TCB protected. The servers inside the TCB that provide access to persistent objects protect them. In this paper, persistent objects may be thought of as files. Each object has an associated security context. As with subject security contexts, the contents of object security contexts are known only to servers. In the servers currently being developed, the security context contains:

- a level attribute that is used by the MLS policy
- a type attribute that is used by the type enforcement policy described in the next section

The *security database* for the system contains subject and object security context tables. These are tables, indexed by subjects and objects respectively, that contain all the security relevant information about subjects and objects in the system. There are similar tables for other classes of system entities such as users, groups, devices, etc. This distributed database is maintained by the security servers and is consulted whenever a security server makes an access control decison related to the system's MLS or type enforcement policy. The MLS policy subsumes the traditional Simple Security Property and $\star$-Property. This is the primary confidentiality policy enforced by the current DTMach security server. The type enforcement policy, discussed in the next section, is the primary integrity policy enforced by the current DTMach security server.

3 Type Enforcement

Many of the solutions DTMach provides to security problems are based on *type enforcement.* Type enforcement is an access control policy that constrains access based on *domains* and *types*. Each subject is assigned a domain attribute and each object is assigned a type attribute. Rather than using an ordered set of sensitivity levels as an MLS policy does, type enforcement uses relations defining which access modes are permitted for each domain-type pair and each domain-domain pair.[3] Then, the policy is expressed as this set of type enforcement relations indicating the modes in which each subject is permitted to access each entity in the system.

While an MLS policy and type enforcement both are based on subject and object attributes, there are two significant differences. First, type enforcement is generally an intransitive policy. For example, the type enforcement relation might allow sbj_1 to communicate with sbj_2 and sbj_2 to modify obj while not allowing sbj_1 to directly modify obj. Intransitivity is quite useful in supporting integrity. For example, if obj contains critical data and sbj_2 is a server responsible for managing obj, it is desirable to have a policy that allows sbj_2 to modify obj while preventing clients sending requests to sbj_2 from modifying obj directly. Figure 3 illustrates this use of type enforcement to protect such a *trusted subsystem* from untrusted system components.

[2] The tagging of messages with security contexts is a TMach extension that has been retained in DTMach.

[3] On the LOCK project, the relation describing access on the basis of a domain-type pair is called the Domain Definition Table (DDT), and the relation describing access on the basis of a pair of domains is called the Domain Interaction Table (DIT).

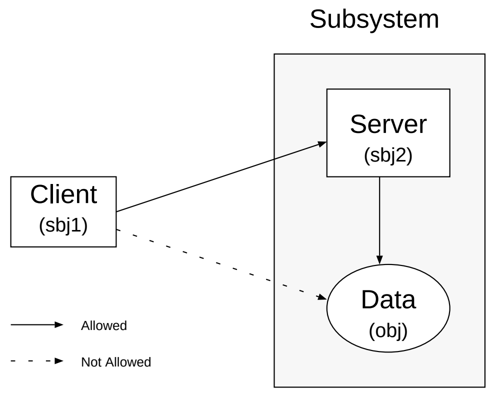

Figure 3: Type Enforcement Protecting a Subsystem. Type enforcement constrains which clients can send requests to the server. In addition, type enforcement prevents clients from bypassing the server and directly accessing subsystem data.

The second significant difference is that type enforcement can provide a finer degree of control. For the purposes of MLS, there are only two types of access modes: observe and modify. There are no such restrictions on the set of access modes used with type enforcement. For example, the LOCK type enforcement policy uses access modes such as *create*, *destroy*, *read*, *execute*, *write*, and *append*. In addition to these access modes, the DTMach type enforcement policy uses access modes such as *receive* and *send* to control access to ports. The following sections describe how DTMach uses the finer granularity access modes to provide a much higher degree of assurance.

As noted previously, entities in Mach are generally named and accessed through ports. This Mach design decision simplifies the incorporation of type enforcement into DTMach since almost all decisions can be made by the security server on the basis of a task security context and a port security context. In particular, the service vector controls the requests that can be invoked on a port in exactly the same manner as reading or writing a file is controlled.

The only difficulty introduced by the use of ports to represent all entities is that it is necessary to ensure consistency between the domain of a task/thread and the type of the port representing it. This is accomplished by associating a unique type with each domain and requiring this type be used for the task/thread port.

Now that we have discussed Mach, DTMach, and type enforcement, we provide motivation for the DTMach approach to providing security.

4 Layered Security Policies and Mechanisms

Although much of the prior work in computer security has concentrated on MLS systems, there is much more to computer security than just MLS. In general, the authorities responsible for a computer system have certain control objectives that are specific to the system. In the case of an MLS system, the primary control objective is that data never be disclosed at an inappropriate level. For a banking system, the primary control objective might be that balances of accounts are never modified inappropriately. A weapons system might have control objectives such as:

- The weapon is only fired when requested by an authorized party.
- When an authorized party requests the weapon be fired, the weapon is fired in a timely fashion.

Besides the wide variation in objectives across systems, the control objectives for a particular system can evolve over time. They might change between the design and the deployment of the system or even as the deployed system is being used.

Consequently, it is unrealistic to expect that a system can be shown to be "secure" before deployment and need never be re-examined. On the other hand, it is impractical to perform a complete security analysis every time a system's control objectives change. The DTMach solution to this problem is to use type enforcement to provide general table driven access control within the base TCB. Then, the table can be configured to support and protect higher layers of enforcement mechanisms for a wide range of more specific, higher level policies.

We use the term "trusted subject" to denote a subject that is responsible for ensuring that some control objective is satisfied. The key to the computer security problem is to simplify the task of assuring that trusted subjects operate correctly.

Ideally, the executable code of a trusted subject would be demonstrated to operate correctly. Although there has been some limited success in verification of machine code for algorithms, analyzing the executable code of every trusted subject in a system is currently infeasible. Another possibility is to verify the source code for a trusted subject. Currently, this is on the cutting edge of verification technology. Although it is important to perform research to advance the state of this technology, it is currently not practical to rely solely on the verification of the source code for trusted subjects.

Thus, common practice is to manually inspect the source code and provide informal correctness arguments. In the case of A1 systems, the arguments are made more rigorous by providing a mapping between the source code and a formal specification of the system. The errors inherent in manual inspection can easily result in a security flaw in the implementation not being detected. To reduce the likelihood of this occurring, it is important to simplify the proof obligation placed on the analyst as much as possible. The DTMach security policy does so by increasing the number of threats that are addressed by the base TCB. Consequently, the number of threats that must be addressed by each trusted subject is reduced and the proof obligation associated with that trusted subject is simplified. In addition to simplifying the correctness arguments for trusted subjects, the policy enforced by the base TCB also simplifies correctness arguments for the TCB itself. In other words, the TCB control mechanisms themselves are used to help ensure that the TCB components correctly implement the base TCB. Figure 4 illustrates this layering of the security analysis.

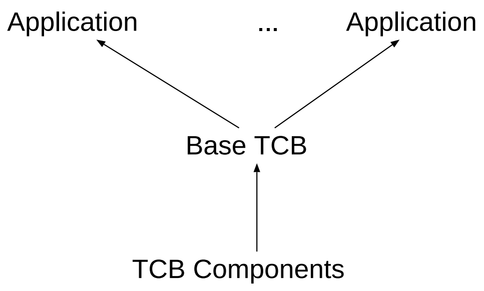

Figure 4: Layering of Security Analysis

The next section describes general threats that trusted subjects must protect against and the mechanisms that DTMach provides trusted subjects to protect against these threats. In addition to providing guidance to the design of the DTMach security server, this also provides justification for the claim that the DTMach security server addresses a wide range of threats.

5 Threats and Solutions

The threats are divided into the following classes:

- Violation of Least Privilege
- Execution of Incorrect Code
- Improper Process Control
- Attacks against Interprocess Communication (IPC)
- Attacks against Data

A separate section is devoted to each of the first four threats. Attacks against data are not discussed since most of the issues related to it are addressed in the discussion of attacks against IPC.

5.1 Least Privilege

First, we consider the *Principle of Least Privilege*. Trusted subjects are often given privileges that are not available to other subjects. For example, MLS trusted subjects are privileged to downgrade information. As another example, only the security database maintenance subjects are privileged to modify the security database.

Part of the analysis of a trusted subject involves ensuring that the trusted subject does not use its privileges inappropriately. For example, an MLS trusted subject should only downgrade certain information; it should not just downgrade arbitrary information. Similarly, the security database maintenance subjects should maintain the security database correctly.

The more privileges a trusted subject has, the greater the analysis that must be performed to demonstrate that the privileges are not misused. Thus, it is desirable to limit the privileges granted to each subject to minimize the required analysis.

The first way that DTMach supports least privilege is through the fine degree of control provided by its access control policy. The DTMach type enforcement policy provides very fine granularity in specifying privileges. For example, it is possible to specify a downgrader that is permitted to downgrade the contents of certain types of objects while having no access to other types of objects. The task of demonstrating that the downgrader only downgrades what is safe to downgrade is much simpler when type enforcement has been used to restrict the accesses of the downgrader.

An example of this would be a downgrader that is intended to downgrade ciphertext prior to export into a hostile environment such as an unprotected network or some form of removable media. On DTMach, special object types, *ciphertext_type* and *ciphertext_control_data_type*, could be defined. Tasks in the *ciphertext_downgrader* domain would only be allowed to read data of these two types, and only tasks in the *encryption* domain would be allowed to write into objects of *ciphertext_type*. Then, it would only be necessary to demonstrate that subjects in the *ciphertext_downgrader* domain only downgrade data read from *ciphertext_type* objects. This is in sharp contrast to the assurance obligation on systems that do not provide type enforcement. In such systems, a downgrader can typically downgrade anything at its level.

As another example, the security database maintenance subjects might be granted read and write permission to the security database while not being granted destroy permission. This would allow the security database maintenance subjects to read and write the database without having to worry about them accidentally destroying the objects containing the security database. There are numerous other examples of how type enforcement supports least privilege[13].

The second way that DTMach supports least privilege is in the separation of port rights and permissions. Since the only port rights that can be used to name objects in Mach are *receive*, *send*, and *send-once*, only one task can hold a *receive* right at any given time, and tasks can only name objects through port rights, it is usually necessary to provide a task a *send* or *send-once* right to a port in order for the task to have a name for the object the port represents. This violates least privilege since there are many cases in which a task does not actually need to send messages to a port even though it does need to have a name for the object the port represents. For example, consider the process by which tasks map files into their address space. The task requests the file from the file server and is given a *send* right to a port representing the object. The receiver for the port is the object's memory manager (pager). Messages that the object's pager receives through the port are typically expected to be coming from the kernel. Thus, allowing a client task to have *send* permission to the port requires the pager to protect itself against the possibility that a client masquerades as the kernel by sending messages to the port. In DTMach, we can provide the client task with a *send* right, but only give it permission to use the port for mapping purposes. Then, the kernel prevents client tasks from sending messages to the port.

As another example, consider a name server. In order for a name server to provide a port to client tasks, it must hold a right for the port. If there is no distinction between holding a right and being able to use the right, then the name server is free to access any object in the name space it manages. In DTMach, we can give the name server's domain permission to pass the right to client tasks while preventing the name server's domain from using the right itself. Then, there is no longer any concern that the name server might maliciously or accidentally access objects in its name space. When the client task receives the right, the client's use of the port is restricted based on the permissions the client's domain has to the port.

5.2 Execute Access

Now, we consider the binding between the source code and the executable code for a trusted subject. As discussed earlier, common practice is to analyze the source code for trusted subjects. If there is no evidence that the executable code is consistent with the source code, then no assurance is gained by analyzing the source code. Thus, maintaining the proper binding between a trusted subject, the source code it is trusted to execute, and the code object actually executed by the subject is a critical piece of the argument for the correct operation of the subject.

One threat to this binding is a faulty or subverted compiler. There is nothing the system can do to prevent a compiler developer from inserting malicious code in the compiler or to prevent bugs from being present in the compiler. However, the system can provide an assured means of installing the compiler and can protect the compiler from inappropriate modification after installation. This can be accomplished using type enforcement by:

1. Defining a type T_c to represent executable code for a compiler.
2. Defining a domain D_i to represent a subject trusted to perform the installation of a compiler.
3. Allowing only subjects in domain D_i to modify objects of type T_c.

Subjects permitted to operate in D_i must be assured to properly install the compiler and to not subsequently modify the compiler. Type enforcement prevents any other subjects from subverting the compiler.

A second way in which the binding between the source code and executable code can be broken is if the executable object associated with a trusted subject is incorrect. This could happen if the wrong object were bound to the subject at create time or if a flaw in the design of the source code resulted in a jump occurring to a new executable object. In the worst case, a user might be able to cause a trusted subject to execute a program designed by the user instead of the program that the trusted subject is supposed to run. This would allow the user to make use of the trusted subject's privileges. In 1988 the Internet worm [12] used these techniques to propagate itself through a significant portion of the Internet.

DTMach protects against this threat by requiring execute access to be a distinct access mode in the permision vector. The DTMach type enforcement policy prevents a subject from executing an object unless the object is of a type that is executable from the subject's domain. The typical security database configuration for a trusted subject is to define a domain to represent the trusted subject and a type to represent the executable object type for the domain. The domain is only given execute access to its executable object type, and the correct code object for the trusted subject is the only object having the executable object type as its type. Thus, type enforcement provides assurance that the trusted subject will only execute the intended code, even if there is a flaw in the design of the trusted subject.

Another way in which the binding between the source code and executable code can be broken is if modifications are made to the executable object. In the worst case, an untrusted subject such as a virus might be able to completely rewrite the executable object and cause the trusted subject to execute arbitrary programs.

The DTMach type enforcement policy can be used to control such modifications. As a simple example, the security database might be configured so that no domains have permission to modify executable objects for trusted subjects. Another possibility would be to allow the objects to be modified only by a maintenance domain. By controlling which users can have subjects active in the maintenance domain and by placing restrictions on the programs that these subjects can run, the system can control the maintenance of trusted programs.

In summary, DTMach uses type enforcement to ensure that each trusted subject is bound with the correct code object and to support the binding between the code object and the source code for the trusted subject.

5.3 Process Control

Now, we consider the manner in which the progress of a trusted subject can be controlled by other subjects. Ways in which a subject can directly control the progress of a second subject include:

- Creating the subject.

  As an example of a threat related to creating subjects, consider a subject that is responsible for shutting the system down. If a user that is not authorized to shut the system down can start such a subject, then the system might be shut down at inappropriate times.

- Destroying the subject.

  As an example of a threat related to destroying subjects, consider a subject that is responsible for updating the security database. If the subject is destroyed while it is in the process of updating a record in the database, the database might become inconsistent due to the partial update.

- Suspending or resuming the subject.

  The threats related to suspending subjects are similar to those for destroying subjects. There is a great deal of similarity between a subject that is suspended indefinitely and a subject that is destroyed.

  There is also a great deal of similarity between the threats related to creating subjects and the threats related to resuming subjects. The primary concern is that a subject might perform an action at the wrong time as the result of being resumed prematurely.

- Invoking requests in the name of another subject.

  The ability in Mach of tasks to invoke requests in the name of other tasks makes it difficult to analyze trusted subjects. For example, suppose that the source code for a trusted task has been demonstrated to be correct, but some other task has permission to send requests to the trusted task's kernel port. By sending requests to the kernel port, the second task can cause the trusted task to perform actions different from those called for by its executable object. This would invalidate the analysis of the source code. As a specific example, suppose that an untrusted task can invoke a **vm_write** request through a trusted task's kernel port. This would allow the untrusted task to arbitrarily modify the trusted task's virtual memory space. Consequently, the untrusted task could cause the trusted task to behave improperly by corrupting the trusted task's virtual memory space.

To destroy, suspend, or resume a task or to invoke a request in the name of a task, a second task must have *send* permission to a kernel port associated with the first task. Thus, destroying, suspending, resuming, and invoking can be addressed by controlling access to task and thread kernel ports. The DTMach service vectors allow very fine control to be placed upon the services that can be requested through a port. In particular, the requests that a task in a given domain can make through a kernel port of a given type are controlled. As a specific example, kernel ports for trusted tasks have types that do not allow untrusted tasks to invoke any services. This is not a particularly interesting case, though, since untrusted tasks should not have permission to send any kind of message to kernel ports for trusted tasks. As a more interesting example, suppose that trusted task t_1 spawns trusted task t_2 in a different domain and does not need to control t_2. Then the security database can be configured so that t_1's domain has permission to create a task in t_2's domain, but subjects in t_1's domain are not permitted to invoke any services through a kernel port for a task in t_2's domain.

As with maintaining the binding between a trusted subject and its code object, we see that process control threats are addressed by including a type enforcement component in the security policy.

5.4 IPC

Some trusted subjects interact directly with untrusted subjects. Due to the client-server nature of the Mach paradigm, this may be more common in the Mach paradigm than in typical secure systems. For example, file servers and name servers must accept requests directly from untrusted subjects. The following issues must be addressed:

- Identifying the sender of a message.
- Protecting messages from modification while in transit.
- Preventing message interception.
- Ensuring message delivery.
- Misdirection.

5.4.1 Identifying the Sender of a Message

For a trusted subject to implement a policy extension, it must identify the security context of the sender of requests it receives. Since many different security contexts might be permitted to send messages to a given port, it is not possible to identify the security context of the sender from the port through which the message is received. To address this concern, DTMach binds the sender's security context to the message at send time. Then, the receiver of a message can identify the security context of the sender by retrieving the security context bound to the message.

Additional requirements are needed to address this requirement in a distributed system. While the kernel can prevent the context bound to a message from being modified while the message is in transit within a node, the network servers must ensure label integrity when messages are passed across the network. If the communication links are physically protected, it suffices to use some form of reliable broadcast protocol[9]. If links are not physically protected, it is necessary to use cryptography in conjunction with a reliable broadcast protocol to protect against malicious agents who have access to the communication links.

It is also important to note that certain DTMach tasks are permitted to specify a sending context to be attached to messages they send. Currently, the only such tasks are network servers. In order for messages to be transparently forwarded across the network, the sending context for the forwarded message must be that of the original sender rather than being the context of the network server. The ability to explicitly set a sending context is controlled using permissions; the DTMach type enforcement policy restricts the domains that are permitted to specify contexts for messages sent to a port of a given type. Currently, the network server domain has permission to specify contexts on all of the types of ports that can be shared across nodes. No other domains are permitted to specify sending contexts.

5.4.2 Protecting Messages from Modification While in Transit

If a message is sent between two trusted subjects and an untrusted subject can modify the contents of the message, then the untrusted subject might be able to trick the trusted subjects into misusing their privileges. For example, consider a subject trusted to sanitize files and a subject trusted to downgrade sanitized files. The sanitization/downgrade process consists of the sanitizer removing sensitive data from the file and then sending a message to the downgrader indicating that the file has been sanitized. Since the security context bound to the message is that of the trusted sanitizer subject, the receiving downgrader subject might assume the file has been sanitized and perform the downgrade. However, if an untrusted subject modified the message so that it requested the downgrade of a file different from the one sanitized, then sensitive information might be downgraded by the downgrader subject.

The Mach kernel protects the integrity of messages that are in transit within a node. The only potential concern is messages containing out-of-line data. The contents of such messages are dependent on memory objects referenced by the messages. This introduces the possibility that a subject might modify the contents of a message by modifying the contents of a memory object referenced by the message. For the most part, the copy-on-write semantics used for out-of-line data address this concern. If a referenced memory object is modified, then a physical copy is performed and the message references the copy rather than the modified object.

However, the Mach user pager concept introduces a small hole. If the pager for an object referenced as out-of-line data invalidates the current contents of the object and informs the kernel of new contents for the object, no copy is made of the original object. Informing the kernel of new contents for an object is viewed as being different from writing the object. Since copy-on-write only requires that a copy be made when a write occurs, the object's pager can actually modify the contents of the message.

To address this "back-door," DTMach restricts the types of memory objects that each task can access and which tasks can act as a pager for each type of memory object and also makes a physical copy whenever a task receives out-of-line data contained in a memory object of an inappropriate type. For example, suppose that $task_s$ sends a message to $task_r$, the message references memory object obj as out-of-line data, and $task_p$ is a user pager for obj. To protect $task_r$ from back-door modifications made by $task_p$ it suffices to configure the security database so that whenever typ is a memory object type that is appropriate for access by tasks in $task_r$'s domain, then no untrusted tasks are permitted to page that type of memory object. Then, the receipt of the message by $task_r$ causes the contents of obj to be physically copied to a new object obj_c that is inaccessible to $task_p$ [4].

As with label integrity, the network servers are responsible for ensuring the integrity of the contents of a message while it is in transit between nodes.

5.4.3 Preventing Message Interception

Trusted subjects often rely on the system to prevent messages they send from being received by subjects other than the subject "intended" to receive the message. In some cases, this is a confidentiality issue; the trusted subject has determined that the intended receiver is permitted to see information and relies on the system to prevent the message from being intercepted by other subjects. In other cases, there are integrity issues to consider. For example, suppose that a trusted subject in DTMach determines that a certain task should be allowed to have a port right and passes the port right in a message to the task. If another task intercepts the message, then it might receive the right even though the trusted subject's policy prohibits it.

In DTMach, this is addressed by allowing a receiving context to be bound to each message. Any message that does not have such a context bound to it can be received by any task that has permission to receive from the port to which it is sent. When a receiving context is bound to the message, then the intent is that only a task with that context can receive the message. As with specifying a sending context, the DTMach type enforcement policy is used to restrict the tasks that are permitted to receive messages when they are not the specified intended receivers of the messages. It is also the case that the kernel is responsible for protecting the context associated with messages in transit within a node while the network servers are responsible for protecting the context associated with messages in transit between nodes.

5.4.4 Ensuring Message Delivery

If a subject is trusted to provide service to other subjects, then it is necessary to ensure that clients of the service can always communicate their requests to the subject providing the service. For example, if an untrusted task can prevent other tasks from accessing a file server, then the file server cannot provide the service that it is supposed to provide. Similarly, if a subject is trusted to provide information to a client subject, then it is necessary to ensure that the subject providing the information can send the information to the client.

In Mach the only way for a task to prevent a message from being delivered to a port is by flooding the port with messages. Each message queue has a limit on the number of messages that it can contain. When this limit is reached, then subsequent messages cannot be enqueued. The DTMach type enforcement policy addresses the flooding issue by controlling which tasks can send messages to which ports. In particular, given a port that is intended for communication between two trusted subjects, the type enforcement policy can be used to prevent untrusted subjects from flooding the

[4] This is similar to the TMach mechanism for addressing this threat.

port by preventing the untrusted subjects from sending messages to the port.

5.4.5 Misdirection

In some cases, an untrusted subject can cause a trusted subject to send a message. In these cases, the trusted subject must prevent the untrusted subject from tricking it into misusing its privileges.

As a simple example, consider an untrusted client requesting a service from an MLS server. It is quite common for the client to include a reply port in the service request, with the intent that the server respond to the requested operation through the reply port. Suppose the MLS server may operate at UNCLASSIFIED and SECRET and a given client has level SECRET. Suppose the client specifies a reply port having level UNCLASSIFIED. Since the MLS server may operate at UNCLASSIFIED, it is permitted to send the reply to the specified port. Consequently, the client can signal information to subjects at level UNCLASSIFIED by sending requests to the MLS server. The client has tricked the MLS server into using its privileges to inappropriately downgrade information.

To address this problem, the MLS server must ensure that the level of the reply port is at least as high as the level of the client. The DTMach approach for addressing this problem is to require that the client have permission to send messages to the reply port. Then, there is no problem with the server sending the reply to the port since the client itself could forward the reply to that port after receiving it.

5.5 Summary

The type enforcement component of the basic DTMach security policy provides a uniform approach for addressing a wide variety of security threats. Many of these threats, such as the violation of least privilege, control of execute access, and process control are present in most secure systems. The threats related to IPC are more specific to DTMach in particular or to distributed systems in general.

The type enforcement policy can be enforced in a straightforward manner by separating the operational notion of Mach port rights from the access control decisions required by the security policy. The DTMach kernel utilizes permission vectors to record these access control decisions. In previous sections we have discussed the abstract notion of permission vectors and the threats addressed by incorporating type enforcement and permission vectors into DTMach. In the next section we discuss related implementation issues.

6 Implementation

In this section we sketch the plans for implementing the permission vector concept in DTMach. First, we describe the structure of the permission vector. Then, we discuss the computation of permission vectors. Finally, we describe the extensions to the Mach kernel interface that are required for DTMach.

6.1 Permission Vector Structure

Conceptually there is a DTMach permission vector for each pair of security contexts. We call these the subject security context and the object security context of the permission vector. The permission vector is divided into two distinct components, corresponding to the notions of permission vector and service vector that were introduced earlier in Section 2.2.2. See Figure 5.

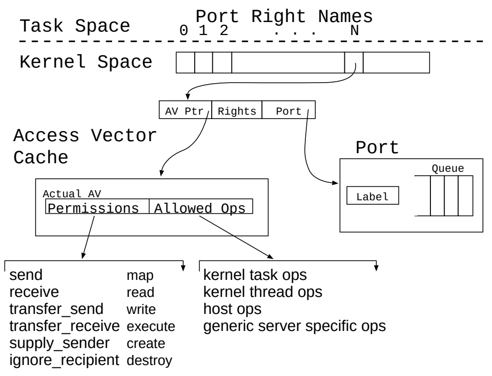

Figure 5: Permission Vector Structure

The first component of the permission vector controls the basic port accesses of tasks in the subject security context to ports with the object security context. The permission vector includes fields that provide:

- Control of the IPC delivery facility: send and receive.
- Control of transfer of port rights: transfer send and transfer receive.
- Control of memory access to objects represented by the port: read, write, execute, create, and destroy
- Control over message identification information: supply sending context and ignore specified receiving context

The second component defines the services that a task in the subject security context is allowed to request in messages to ports with the object security context. Thus, a security policy can specify on a per security context basis exactly which services the kernel should allow for each type of service port. The kernel's role in making this facility available to the TCB is to assure that the policy specific logic is executed using the correct security attribute information and that the data is correctly provided to the server task along with the service request message.

6.2 Permission Computation and Revoction

Since any major application built on an IPC based kernel such as Mach is likely to access ports frequently, performance considerations dictate that permission computations not be required on every operation involving a message exchange. Another performance motivated consideration is that the Mach IPC facilities encourage the frequent creation and deletion of port rights. These considerations lead to the following design decisions.

- Permission vectors are associated with a port right the first time the port right is accessed by a task.
- Permissions are cached by the kernel based on the security attributes of the relevant task–port pairs.
- Permissions are computed only the first time they are required.

The general view of security decision processing is then similar to page fault processing. When a task attempts to utilize a port right the kernel first determines that the port right allows the requested IPC operation. This is precisely what the Mach kernel does. Then the DTMach kernel checks to see if the requested access is allowed by a valid permission vector associated with the task and port. If there is none, the kernel will search its cache to see if it has a previously computed valid permission vector for the security attributes represented in the task–port pair. If no entry is found in the cache, the kernel initiates the required security check by providing the relevant security context information to a security server. The security server might not be necessary for a very simple security policy such as the conventional Mandatory Access Control Policy. The kernel might execute such simple decision logic more efficiently than it can maintain a cache. However, for more complicated and flexible security policies such as the DTMach type enforcement policy, it is more efficient to move the actual computation of permission vectors outside the kernel. Use of a security server task for computing the permission vectors places all the security critical logic in a single place in the TCB. This makes it very easy to analyze and evaluate the mapping of a specific security policy to the security critical decisions within the TCB.

The permission vector cache in the kernel makes resolution of the permission revocation problem relatively simple. The kernel provides a "revoke access" service which specific TCB tasks are allowed, by the service vector component, to request. Upon receipt of this request, the kernel invalidates all permission vectors in the cache. When a task makes a subsequent request to utilize a port right, the kernel reacts as if there were no permission vector present and initiates a recomputation based on the current state of the system. In most cases an application will be completely unaware of the new permissions. In some cases a request may be returned with an "insufficient permission" error. In this case the application will take whatever action is required.

6.3 Interface Extensions for DTMach

An explicit objective of the DTMach design effort has been to preserve the existing Mach kernel interface and to minimize the number of extensions that are necessary. If this objective is achieved, code written to operate on a Mach kernel will also operate on the DTMach kernel. The code will be able to make all the same requests and will find the same general semantics for all those requests. To address security issues, variants of a small set of kernel entry points have been proposed. In this paper we discuss the general issues associated with the proposed changes. The interested reader can consult [8] for detailed descriptions of the changes.

There are four general concerns that lead to extensions of the existing Mach kernel interface.

1. Definition and management of security contexts.
2. Association of security contexts with basic DTMach entities.
3. Access to security context information.
4. Required extensions to the semantics of task creation.

Since the kernel's role in the system has been carefully defined as the enforcer of decisions and not the maker of policy specific decisions, it is not required to interpret the content of the security attributes of entities. Its role is one of simply associating and storing the information with the relevant entity, passing it on to the security decision logic as required, and providing the information to other client tasks as requested. It is also desirable for the kernel to have the ability to determine when two security contexts are identical. There are points in the kernel processing logic that are used for both inter and intra security context operations. Allowing the kernel to quickly identify the usage as an intra context operation decreases the number of times that complicated security checks must be made. To allow applications to associate security contexts with ports, memory regions and tasks, variants of the **mach_port_allocate**, **vm_allocate** and **task_create** requests are specified. Each allows the caller to designate the security context to be associated with the specific entity that is created.

To implement a client server model in a TCB, the TCB servers must have access to the security identity of the client. The initial Mach IPC mechanism does not provide any kernel assured mechanism for receiving this information. This problem is solved by providing a variant of the standard Mach message service request. The variant allows a receiving task to tell the kernel to provide the security relevant information for the message sender along with the actual message. The information provided includes both the sender's security context and the permission vector defining the sender's actual permissions to the port used for the communication. This provides two distinct benefits to the TCB server. First, it has the identification information it requires to carry out any security relevant responsibilities that it might have. Second, it need not duplicate any previous security decision. This means that the most critical security decision logic can be centralized to a single system module, the security server. Such centralization is a distinct benefit to the assurance process for any secure system. Finally, as described in Section 5.4, DTMach provides a variant of the Mach message service request so that the security contexts of the original sender and/or the intended receiver can be bound to the message

The relationship between a parent task and its child is more complex in DTMach than it is in Mach 3.0 where there is an implicit assumption that the parent task always has full control over the child task. This is acceptable in a security environment where trust can be viewed as a non-increasing function when applied to the task creation processing, e.g. a Biba integrity model. However, in a more general case, where untrusted applications request services of trusted applications, e.g. the LOCK TCB [13], it is necessary to allow parent tasks to create higher integrity child tasks. This presents a more difficult problem because the transition from low integrity to higher integrity cannot rely on the parent to do anything for the child except specify its security attributes, tell it what to attempt to execute, and tell it when to start.

In DTMach this problem is solved by incorporating several existing task operations into a single new variant of the **task_create** request. In Mach there are three distinct operations that a creating task must perform to start a new task. It must indicate what part of its address space is to be made visible to the child task. It must then enter the **task_create** request. Upon completion of the **task_create** request the parent receives a *send* right to the child's task port, which allows the parent to enter any and all kernel requests in the name of the child. To complete the process of making a useful child task, the parent must use this right to enter requests in the name of the child to create a thread, set the context for the thread, and tell the newly created thread to resume processing. The DTMach variant of the **task_create** request incorporates the multiple kernel requests into a single request. Thus the DTMach **kernel_cross_security_context_task_create** is a two step process for the parent task. The parent task still identifies the parts of its address space that it would like to share.[5] Then the parent task enters the variant task create request which provides for not only the creation of the task, but the creation and initiation of an initial thread within the task.

7 Conclusion

The preceding discussion has described threats that must be addressed in secure systems and provided an overview of how the DTMach security policy addresses each of the threats. By addressing the identified threats rather than simply addressing MLS security, a wider range of policies can be supported. High assurance is obtained by incorporating the type enforcement policy in the TCB. Rather than repeatedly assuring complex application level policies, assurance is provided for the DTMach type enforcement policy and simple arguments are provided for how the DTMach type enforcement policy supports the application level policy. By using type enforcement to construct protected subsystems, the analysis can be modularized and consequently made much more feasible.

As described throughout this paper, type enforcement is the key component of the DTMach security policy. As described in [11], type enforcement was also found to be invaluable in the design of the LOCK TCB. The ease with which type enforcement was "ported" from LOCK to DTMach provides support for the claim that it is a generally applicable security policy. The fine granularity of control it provides is essential to the development of high assurance systems.

Acknowledgments

Thanks to Tom Haigh, George Jelatis, and Carol Muehrcke for providing many helpful suggestions. This paper was derived from work performed under contract number F30602-91-C-0136 under the sponsorship of Rome Laboratory, C3AB.

[5]The actual sharing of information via memory regions is restricted by the kernel enforced security policy decisions.

References

[1] "Trusted Mach Kernel Primer," Trusted Information Systems, Inc., Draft 0.1, November 27, 1991.

[2] Martha Branstad, Homayoon Tajalli, Frank Mayer, and David Dalva, "Access Mediation in a Message Passing Kernel," *IEEE Symposium on Security and Privacy*, pages 66-72, May 1989.

[3] Li Gong, "A Secure Identity-Based Capability System," *IEEE Symposium on Security and Privacy*, pages 55-63, May, 1989.

[4] P.A. Karger and A.J. Herbert, "An Augmented Capability Architecture to Support Lattice Security and Traceability of Access", *IEEE Symposium on Security and Privacy*, pages 2-12, April, 1984.

[5] P.A. Karger, "Improving Security and Performance for Capability Systems", University of Cambridge Computer Laboratory, October, 1988.

[6] Keith Loepere, "Mach 3 Kernel Principles," Open Software Foundation and Carnegie Mellon University, March 15, 1991.

[7] "Software Requirements Specification for Distributed Trusted Mach. DTMach CDRL A005," Secure Computing Corporation, June 22, 1992 .

[8] "System/Segment Design Document. DTMach CDRL A006-V1,2," Secure Computing Corporation, July, 1992.

[9] Mamoru Maekawa, Arthur Oldehoeft, and Rodney Oldehoeft, *Operating Systems Advanced Concepts*, The Benjamin/Cummings Publishing Company, Inc., 1987.

[10] MACH Networking Group, "Network Server Design," Carnegie Mellon University, August 1989.

[11] O.Saydjari, J. Beckman, and J. Leaman. "LOCK Trek: Navigating Uncharted Space," *IEEE Symposium on Security and Privacy*, pages 167-175, May 1989.

[12] Andrew S. Tannenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

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