Providing Non-hierarchical Security through Interface Mechanisms

Deborah Hamilton

Hewlett-Packard Labs

Abstract

Common security models provide protection in an hierarchical fashion (i.e. there is a trusted core with outer circles of less secure code and data). There is only one method of providing protection. This model makes it difficult to protect code and data with multiple types of non-hierarchical policies. It implies complete trust in the core requiring thorough evaluation each time modifications are made. This paper first describes a paradigm shift to non-hierarchical security. It then describes an interface mechanism with the potential for providing an efficient, configurable and non-hierarchical security mechanism more suitable for commercial requirements.

1 Introduction

The increasing complexity of current systems and the growing range of security requirements is forcing a paradigm shift. This paper first describes some problems with current security models and then discusses a paradigm shift that could provide a more suitable model. It then provides some background information, assumptions and an overview on a procedural communication and protection mechanism which we claim can aid in the paradigm shift. Next, the paper describes how this interface mechanism allows for a separation between the policy and mechanism. A comparison is made between security enforced through an interface mechanism and common models. The paper then discusses how interfaces can provide an efficient enforcement for multiple and configurable policies. Functionality requirements and technical challenges are also discussed. Finally, a prototype is described along with ideas for future work.

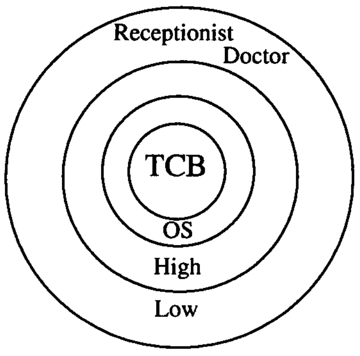
2 Limitations of current systems

Two major limitations in current systems are the integration of the security policy with the enforcement mechanism and the size and complexity of the of the TCB. These limitations cause problems because:

1. One must trust the operating system. Although the operating system has access to everything, one must trust it will not modify anything directly or indirectly in an incorrect way. There are many cases, for example, where users have been able to get the operating system to indirectly modify code that should not be accessible to the user. Due to the size and complexity of common operating systems it is almost impossible to verify all actions for all inputs without restricting the functionality and impacting the performance significantly. Furthermore, it is cumbersome to reevaluate the code when changes are made.
2. It is difficult to support a variety of policies as well as support multiple policies at one time because the security policy and enforcement mechanism are entangled and not designed with this in mind. If they could be separated, it would allow solution providers to concentrate on the security requirements instead of the mechanism. It would also allow configurable policies and make production of and experimentation with a variety of policies possible.
3. Most operating systems provide no support for a variation in the type and granularity of protection. Often applications are built over the operating system to overcome some of these limitations. For example, databases can be built to give more variety on the type and granularity of protection. However, one must believe that the database protection can not be subverted and one must accept the increased overhead.

These are serious limitations which are unacceptable in the majority of commercial systems.

The traditional orange book model conceptually has the TCB surrounded by layers which give less and less accessibility as one goes away from the core. This approach is hierarchical. Most if not all the code and data is available while in the operating system core while less code and data is accessible as one goes further out in the circle. Figure one gives a conceptual picture of the traditional security model with both categories (receptionist/doctor) and security labels (high/low).



Hierarchical Security

Figure one

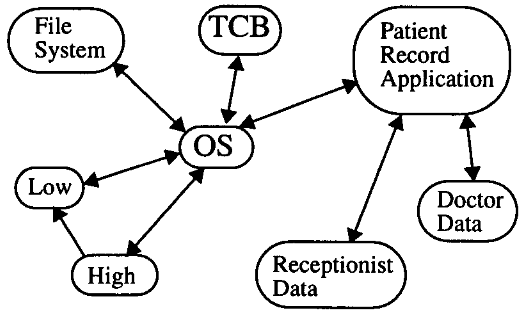
This model has one method of protecting code/data and one method of gaining access to it. Code/data in the outer layers are accessible to those operating further in. Programs may execute in one layer but have access to areas further out. It is similar to a one-room log cabin with one door and several walls encircling the cabin each with one door. It is difficult to support multiple and non-hierarchical policies with one method of entry (i.e. one door). It is also difficult to provide control once a program has access to the operating system. Using a one-room house was sufficient before the necessity arouse to have many people using the room for many different purposes. However, the one room - one entry model no longer maps to the real word needs of commercial systems.

The current approach to making a system more secure is by re-enforcing the security boundaries (i.e. adding another wall around the house or reinforcing an existing one). This prevents *bad people* from entering the yard or house. However, this doesn't help us address the limitations previously listed. This is - it doesn't help us control and limit access to those allowed into the yard and/or house.

3 The paradigm shift: Moving from an hierarchical to a non-hierarchical model

It is time for a paradigm shift from a one-room/one entry model to a modern office complex. We must build rooms inside the operating system and allow multiple entries and methods of providing protection for the various rooms. In an office complex there are different means for entry depending upon the hour and person entering. For example, temporary employees wouldn't be given free reign of the building. They may be allowed entrance only when escorted after showing an ID during specified hours at predefined doors. The VIPs, however, may have keys which allow them in any door at any time. In addition, there may be a drive-through window staffed with people who service requests for information and maintain control over access.

Figure two gives a picture of a non-hierarchical model. This model allows parts of the operating system to operate in separate segments. Programs may also run in one or more separate segments. Data may be stored in the same segments as the program that accesses them or may divided



Non-hierarchical Security

Figure two

among a few segments. Several segments may be accessible at the same time.

The states in figure two represent accessibility to a group of data and/or code. Notice that there may be multiple methods of getting to a state. For example, the Low state can be reached either directly or via the high state. Information about how the state was reached is maintained in case it is needed to determine the next state.

Sometimes one may want the security model to be hierarchical as when using a low/high level security model. It is easy to see how a non-hierarchical model could imitate a low/high level by defining high security to give access to both high and low. However, it is difficult to imagine how a hierarchical model could effectively produce a non-hierarchical model such as categories.

The non-hierarchical model can be thought of as an object-oriented view. Each segment represents an object which may have code or data. These objects can maintain attributes and state information which may be based on users, programs, or arguments given when running the program.This information can be used when determining access or when restricting access to parts of the object from access/view. For example, in a patient record system, one may want to designate access based on roles and allow access to only portions of the record (i.e. only the doctor can write in diagnostics section). Although information is available about which user has run a program, there is no pretense that programs are always run on behalf of one and only one user as in the current models.

4 The interface mechanism

A successful transition to a non-hierarchical mechanism requires an efficient, secure, flexible and easy to use mechanism. Some people have attempted to move from a hierarchical model to a non-hierarchical model but were unsuccessful for various reasons.

Attempts to build a non-hierarchical protection mechanism into the software application layer present several problems. First, security depends on the mechanism being impossible to subvert. Second, the mechanism must give adequate performance in order for it to be used pervasively throughout the system. Third, if applications must be retrofitted, it stands less chance of being used and accepted. Fourth, the granularity of the protected . regions should be adaptable. Fifth, the mechanism should provide protection within the kernel so that giving access to a part of the kernel does not mean free reign. This makes verification easier and allows a true non-hierarchical model.

We have developed an interface mechanism that is efficient and secure. We believe it can be used pervasively in a system with little overhead. A user can run programs as they now do with the desired amount of protection.

5 Background

A paper design of an operating system, Brevix, was developed at Hewlett-Packard Labs [1]. Various aspects of the design showed significant potential for short-term and long-term applications and were further developed and prototyped. This paper describes one aspect, the interface mechanism, which provides a well-defined inter-space interface for isolating and controlling access to code. We also discuss potential extensions related to implementing security policies. The low-level detail of the interface mechanism is not required for a high-level discussion of the security aspects and was thus left out of this document.

6 Abstracting out the underlying hardware assumptions

The interface mechanism assumes a mechanism for protecting virtual address space. The prototype described in this paper was developed on the PA-RISC platform which meets the above requirement by using *protection IDs*. This virtual address protection mechanism will not be discussed here - refer to [3] for details. Instead we will assume this protection is possible and will abstract out the underlying assumptions made by the interface mechanism for the sake of simplicity and generality.

We will use the term *protection ID* to imply that there is something one must posses in order to gain access to a protected region. One is able to restrict what type of access the protection ID gives: read-only, read/write, read/execute, or read/write/execute. There can be many protected regions and many protection IDs. The protection IDs are not transferable - only trusted code controls them and can give out access. One protection ID may give access to multiple protected regions and there may be identical protection IDs giving access to the same region but two different protection IDs can not give access to the same protected region. There may be unprotected regions which do not require a protection ID for access and trusted regions whose access is controlled through a mechanism other than protection IDs. If access is requested without a valid protection ID or other type of permission, a trap is taken in the trusted code. If the protection ID can not be obtained an error is returned to the caller.

7 Interface mechanism overview

The interface mechanism provides a method for controlling access through a well-defined procedural interface between address spaces. The granularity of address space is flexible. For example, the kernel or users may each operate in one address space or may be broken apart into multiple address spaces based on functionality.

Each process has its own collection of currently available protection IDs which we will refer to as a protection domain. These protection IDs give each process direct access to their own space (code/text/data) as well as possibly other protected regions. The protection domain may also include information on what type of access is allowed and other security relevant information. The protection domain can only be modified through the interface mechanism.

An interface definition must be given for each set of entry points that is provided to external callers. This information is used by the interface mechanism to determine how to modify the caller's protection domain as it crosses into the callee code through one of the predefined entry points. During the call, the caller will have access to the necessary protection IDs for execution in the callee code. Immediately before the execution returns to the caller code, the interface mechanism modifies the protection domain again deleting the protection IDs temporarily given for access in the callee space (code/text/data).

Protection domains can also be changed by privileged processes in a special way which will not be undone after crossing back out of the privilege process. This allows privileged processes to setup the protection domain during login, for example.

Regions of the virtual address space may also be defined as unprotected and thus accessible to all without the use of a protection ID. When a caller crosses an interface into an unprotected region, the protection domain need not be updated. The callee code will be executed under the same protection domain as before the interface was crossed.

The crossing from caller code into the callee code through the interface mechanism is referred to as the interface crossing. A thread represents a single flow of control within the system.When a thread crosses an interface by making a procedure call, three things are ensured by the interface mechanism.

- The thread may enter the new code only at a valid entry point advertised by that code.
- If necessary, the thread enters the protection domain of the new code. The system ensures that the regions of virtual memory available to the thread are those of the new protection domain.
- The thread may carry parameters across the interface.

When the thread returns across the interface from the called code, three things are ensured by the system:

- The thread may only return to the call site.
- The thread returns to the protection domain of the caller. The system ensures that the regions of virtual memory available to the thread are those of the original protection domain.
- Any return values from the procedure call are made available to the caller according to the return convention of the system.

There are three types of interfaces: open, unprotected and protected. Each class of interface provides a higher degree of protection than it's predecessor.

- Open interfaces enable a callee's subroutine to be linked into the caller's code. The code will run in the same protection domain as the caller. An open interface provides an efficient mechanism for library calls such as bzero where no data from the callee is accessed. This is accomplished by performing all interface related activity at initial binding time, rather than at call time.
- Unprotected interfaces cause a change in the protection domain at interface crossing time. Any caller may use an unprotected interface.
- Protected interfaces cause a change in the protection domain at interface crossing time. Access to protected interfaces is restricted to a set of callers through the use of access control qualifiers.

Interface crossing must be efficient in order to provide a viable solution. As much of the required processing as possible should be done before the actually crossing. For example, entry point addresses and protection IDs can be determined at boot time. In additional, the interface mechanism can be optimized to:

- Use hardware protection access checks for efficiency
- Eliminate unnecessary context-switches (one is required at most, none if an interface specifies that no protection is required)
- Minimize branches (entry point addresses are embedded in caller's a.out image)

Since all protection domain changes and thus all changes to security information and availability to protection IDs are done through the interface mechanism, access and modifications to security information can be controlled and monitored. The trusted code executed at interface crossing time can provide hooks for a security manager.

8 Implications of the interface mechanism on security

Security functionality can be separated into two parts:

- Enforcement mechanism - the actual access check
- Policy implementation - determining when to give access

The interface mechanism provides the enforcement through the hardware at access time. The policy implementation is done by means of temporary or permanent modifications to the protection domain during interface crossing. Notice that the enforcement mechanism and the policy implementation are separate and independent.

The interface mechanism must determine how to modify the protection domains - either by using some built in knowledge or by consulting a security manager. In either case, the protection domain modifications will be determined based on the security policy, the current protection domain (including the relevant security information), and the type of interface about to be crossed.

9 Comparisons with other models

Discussing the differences between the interface mechanism and security in other models helps to clarify the interface mechanism.

9.1 Reference monitor

There are several major differences between the reference monitor model [2] and the interface mechanism:

- *Degree of separation between security enforcement and policy implementation:* A reference monitor integrates the two - both are done at the same time by the same process. There have been some attempts to separate the two. For example, in the IX Unix model developed by AT&T everything is labeled and checked before every data transfer as in the reference monitor model. However, the actual access check is optimized by setting a flag and calling the policy implementation code when necessary. Even though the code has been optimized, the two are still dependent on one another. The interface mechanism separates the security enforcement and policy implementation completely. The security enforcement has no dependencies on the policy implementation - it simply expects to enforce whatever has been set up by the policy implementation.

- *Location of interception:* A reference monitor intercepts each data access where as the interface mechanism only intercepts when making intra-space calls in order to update the caller's protection domain. This creates a more efficient mechanism. It creates extra work, however, if trying to track all actual data accesses since the interface mechanism can only know when one has a protection ID for access and not when and how it is used (see the section on auditing below).
- *Ability to trade-off protection and efficiency:* The interface mechanism allows the callee to either have unprotected regions, give direct access to protected regions or force callers to go through the interface crossing mechanism in order to gain access to protected regions. The overhead associated with modifying the protection domain is only taken when crossing into a protected interface. One may choose not to trade security off for efficiency by defining unprotected interfaces which require no overhead. In addition, much of the work for interface crossing can be preprocessed making the crossing more efficient at the expense of disallowing dynamic changes to the security information. The reference monitor model takes the overhead hit for calling the monitor each time even if nothing needs to be done.
- *Efficiency of access check:* Access checks are integrated into the hardware when using interface mechanisms on PA-RISC -- there is no overhead associated with it.
- *Size of trusted code:* Since the interface mechanism uses hardware access checks, the amount of trusted code is minimized.
- *Data sharing methods:* Interface mechanisms allow four types of data sharing: no protection, direct access, parameter passing, and restricted use while in the callee code. The strict reference monitor model provides only one outside of the trusted code - data passed through the reference monitor.
- *Control over granularity of protection:* The reference monitor does not allow control over the granularity of protection The interface mechanism has a great degree of flexibility in determining the appropriate granularity for protection. The data to be shared can be divided into small chunks of which each requires a distinct protection ID. This option may create significant overhead but is possible. One can also put all shared data into one region and have an optimal solution at the expense of granularity of protection.

9.2 Messaged-based communication and protected ports

Several systems such as MACH use a protected message port scheme to control access to data. All communication and data sharing is done through messages which often results in a large overhead. There is no method of efficiently sharing some region of data unless the protection mechanism is by-passed. A port must be defined for each type of access desired. Once again, the separation of enforcement and policy implementation, location of interception, the ability to make trade-offs, the efficiency of access checks, size of trusted code, data sharing methods, and control of protection granularity vary between this scheme and interface mechanisms.

9.3 Token-based protection

Token-based protection does allow separation between enforcement and policy implementation but they are not independent - only certain types of security policies may be implemented. Once tokens are granted, it is difficult to control or monitor their use (i.e. what data is accessed) since tokens are not integrated into and controlled by the operating system. In the interface mechanism, it is possible to enforce more control by allowing protection IDs to be used only during a call through a specified interface crossing and deleted immediately upon return. Thus the protection ID is only available only within the callee code. The efficiency of access checks, size of trusted code, and data sharing methods vary between this scheme and interface mechanisms.

10 Security policies

The implications of the separation between the security enforcement and the policy implementation are important. The interface mechanism method of controlling access is not linked to classifications and labels nor any other type of policy. A variety of security policies could be implemented. For example, the policy could stipulate that all interfaces must be protected and that no direct global data sharing is allowed. This is done by restricting the sharing of protection IDs. One could simulate delegation and tokens by allowing interfaces which designate permanent protection ID access to particular callers.

The protection domains as well as the interfaces can contain information such as classification labels for use with MAC and DAC policies. The ability to cross an interface (and which interface to use) could be determined based on these labels. A high security interface could give out different protection IDs but use the same entry point and therefore the same code as a low security interface. In this situation, two processes using the same code would have different access.

Roles can be implemented by operating within a specific callee code. All associated protection domain changes can be done by crossing the interface into that callee. Information about the role can be stored in the protection domain as well. An auditing mechanism can store the role information along with any protection domain changes during interface crossing in order to capture the full environment.

10.1 Configurable policy support

Since the security enforcement and policy implementation are independent and separate, it is possible to allow configurable security policies defined at system generation or boot time (if a secure boot is available). The security enforcement mechanism needs to know nothing about the policy. The interface mechanism need only know where to find the security manager.

10.2 Multiple policy support

A security manager may know about one or more policies to implement. The security manager would have overall rules as to which policy to use when. For example, a process with a protection domain stating MAC could only use interfaces labeled as MAC and all data regions it allocates would be identified by the correct label. The security manager would also have rules as to how to resolve conflicts between the policies using perhaps a predefined precedence. This paper does not intend to address all of the interesting issues associated with trying to implement multiple policies - only to point out that interface mechanisms might be a foundation for work in this area.

10.3 Auditing

The modifications of the protection domains are done through one process and therefore all changes to access can be monitored. All security information is contained in the protection domain (such as user ID and history of interfaces crossed) so it too is accessible to the audit mechanism. Since only the interface crossing mechanism is able to modify the protection domain, it can be securely maintained.

One drawback to the interface mechanism is that in order to track all data assesses and modifications at a fine granularity, one must force all accesses through a callee which will securely audit the transactions. Alternatively, one could have the security manager keep track of the parameters passed in and out and define the interfaces such that the resulting data access and modifications are completely predictable.

11 Functional requirements and technical challenges

There are many interesting research topics associated to interface mechanisms. This paper does not address the problems associated with ensuring trusted code for the interface mechanism or security manager. Neither does it address how a policy is securely configured or implemented. Revocation and information management are two other areas which require more investigation.

During interface crossing, certain information must be made available to the security manager. For example, the interface mechanism must know which policy applies as well as security information relevant to the caller and interface such as user ID and classification. It is important to determine how this information can be securely maintained and accessed by the security manager.

In order to implement some security policies, it must be possible to revoke protection IDs. The brute force method may be possible but inefficient. It would require either keeping track of who is given which protection ID or changing all of the required protection IDs. A more efficient method might be forcing a trap for all processes which would result in a re-evaluation of each protection domain. In addition, any preprocessed interface crossing information might need to be re-evaluated.

12 Interface mechanism prototype

We are in the process of prototyping an interface mechanism on PA-RISC using HP-UX. The motivation for this effort stems mostly from the difficulties of enhancing, debugging and maintaining kernel code as well as introducing third party code. In addition, this prototype allows us to investigate possibilities such as the potential for security hooks aimed at commercial applications.

Our prototype implements a simplified version of the Brevix interface mechanism design. In order to fit the mechanism into an existing operating system for prototyping in a timely fashion, we eliminated some of the complexity and thus some functionality. We focused on first providing an efficient mechanism to protect against non-malicious errors. It is our intention to incrementally work towards a complete Brevix interface mechanism implementation. The major security relevant distinctions between our prototype and what is described in the Brevix design document (as well as in this paper) are:

- Our prototype does not address malicious attacks. Stacks will always be shared between the caller and callee. Access to the interfaces will not be controlled in the prototype - any process that knows about the interface may use it.
- Each protected region can have only one set of defined access rights.

- Support for reference parameters will be limited to two cases: contiguous data and a minimal set of non-contiguous data types. Others must be identified and dealt with manually. Further work must be done in order to determine how to track down and protect chains of reference parameters.

Because of the above restrictions, only kernel-level interfaces are currently practical.

13 Future work

There are many interesting possible extensions to this work. Some possibilities include:

- Designing and implementing a security manager. David Bell and Ruth Nelson presented work at this workshop that would be interesting extensions for implementing a security manager on top of this mechanism.
- Implementation of both hierarchical and non-hierarchical security policies. It would be interesting to implement common polices and some of the new commercial policies within the interface mechanism.
- Dynamic interface support. The ability to modify the interface information on the fly would allow dynamic modification to the associated protection IDs or entry-points.

14 Summary

The current security models aren't working. We must find a better model. This is especially true for commercial systems. A non-hierarchical object-oriented model seems to provide a better mapping to the commercial requirements. The Brevix interface mechanism is a plausible mechanism for exploring this new paradigm.

15 Acknowledgments

The interface prototype was done by the Brevix team: Bart Sears, Marty Fouts, Tim Connors and myself. I also want to thank Marty Fouts for reviewing this paper.

16 References

1. Marty Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes. *Brevix design 1.00*. Technical Report HPL–OSR–93–22. Operating System Research Dept., HPL, 1 Apr. 1993.
2. Department of Defense: Trusted Computer System Evaluation Criteria. December, 1985.
3. PA-RISC 1.1 Architecture and Instruction Set Reference Manual. Hewlett-Packard. November 1990.