Escort: A Path-Based OS Security Architecture

Oliver Spatscheck and Larry L. Peterson

TR 97-17

Abstract

Escort is the security architecture for Scout, a configurable operating system designed for network appliances. Scout is unique in that it is designed around *paths*—a communication-centric abstraction that encapsulates information flows through the system—rather than the more traditional processes and servers. Scout uses paths to make end-to-end resource allocation decisions. Escort extends this idea to isolate these information flows, as well as to provide end-to-end accountability. This paper introduces the Escort security architecture, shows how it can be used to enforce common security policies, and evaluates its design according to several well-established criteria.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

The ability to secure a computing system depends greatly on the design of its operating system. Saltzer and Schroeder [14] introduced a set of criteria by which one can evaluate the secure design of a computer system. These criteria have recently been summarized by Wallach, et. al. [17], and are listed below.

Economy of Mechanism: Designs which are smaller and simpler are easier to inspect and trust.

Fail-safe Defaults: By default, access to every object should be denied unless it is explicitly granted.

Complete Mediation: Every access to every object should be checked.

Least Privilege: Every program should operate with the minimum set of privileges necessary to do the job. This prevents accidental mistakes from becoming security problems.

Least Common Mechanism: Any resource that is shared among different programs can be used as a communication path, and thus, a potential security hole. Therefore, as little data as possible should be shared.

Accountability: The system should be able to accurately record who is responsible for using a particular privilege.

Psychological Acceptability: The system should not place an undue burden on its users.

Monolithic operating systems like Unix violate several of these design principles. First, a monolithic kernel is large, making it difficult to inspect and trust. Second, monolithic systems violate the principle of least privilege since they offer a single broad interface, and they allow a process to use all privileges given to the user that spawned it. Third, they do not follow the principle of least common mechanism since many kernel data structures are unnecessarily shared by all processes. Fourth, monolithic systems do not support full accountability since there are kernel tasks not associated with any user process; e.g., time spent processing incoming network packets is not associated with the receiving process. Finally, the principle of complete mediation is only enforced on a very coarse-grain level.

Microkernel-based systems like Mach [1] attempt to address some of these limitations. Instead of a single large kernel, they consist of a small (micro) kernel and a collection of servers. Each server consists of less code and defines a narrower interface, making it easier to verify each component and improving the system's ability to restrict programs to the minimum set of privileges they need to do the job. Unfortunately, the servers are still fairly coarse-grain—they are on the order of a UNIX server or a network message server—meaning that many of the limitations outlined above are only marginally improved: servers are still difficult to inspect, access control is still fairly coarse-grained, processes that enter a server still have access to shared state, and all the processes that enter a server have to trust that server will not leak information from one process to another. In addition, server-based designs actually make accountability harder since a cascading of servers—i.e., one server invoking another—makes end-to-end accountability more difficult.

The DTOS operating system [10] addresses some of these problems by adding finer grain control over the individual ports in Mach. For example, a user that has access to the port of a UNIX server in regular Mach can issue all UNIX system calls; it is up to the UNIX server to make finer grain access control decisions. DTOS makes it possible to restrict this access in accordance with a global policy; e.g., limit the user to certain file accesses. However, this approach also falls short. If a server uses another server to fulfill its task, the access control of the user is based on the rights available to the intermediate server rather than the user that originally issued the request. Again, all servers have to be trusted.

Based on this experience, it is obvious that a secure OS design depends not only on fine-grain access control and a global policy, as DTOS provides, but also on the ability to protect the flow of information through the system; i.e., from server to server. As early as 1985, Boebert and Kain [5] introduced the idea of protecting information flows in *assured pipelines* to enforce certain policies. T-Mach [15] and DTOS also allow policies that define an implicit flow of information, but no operating system has centered its design around the idea of a flow rather than processes and servers.

This paper describes an OS security architecture built around the idea of an information flow. The architecture, called Escort, is designed for the Scout operating system [11], which defines a first class *path* object to encapsulate I/O data as it moves through the OS. Scout paths were originally designed to enable code optimizations and to support quality of service guarantees, but as we demonstrate in this paper, they can be extended in a natural way to build secure systems. While this path-based approach can be applied to Unix-like systems, Scout is designed for network appliances, such as network-attached devices (cameras, disks, displays), application-level gateways (firewalls, web caches, proxies), hand-held and portable devices, and specialized servers (file and web servers). This paper describes the Escort security architecture in terms of network appliances.

Section 2 gives a brief overview of Scout and Section 3 presents the Escort security architecture. Section 4 then describes an example system—a web server—to illustrate how the architecture works, and Section 5 evaluates Escort in terms of the criteria outlined earlier in this section.

2 Scout OS

Scout is a configurable OS that includes primitive abstractions to support communication. It is written in C and runs stand-alone on Intel Pentium and Digital Alpha processors. This section gives a brief overview of Scout.

2.1 Configurability

Modules are the unit of program development and configurability in Scout. Each Scout module provides a well-defined and independent functionality. Well-defined means that there is usually either a standard interface specification, or some existing practice that defines the exact functionality of a module. Independent means that each single module provides a useful, self-contained service. That is, the module should not depend on there being other specific modules connected to it. Typical examples are modules that implement networking protocols, such as HTTP, IP, UDP, or TCP; modules that implement storage system components, such as VFS, UFS, or SCSI; and modules that implement drivers for the various device types in the system.

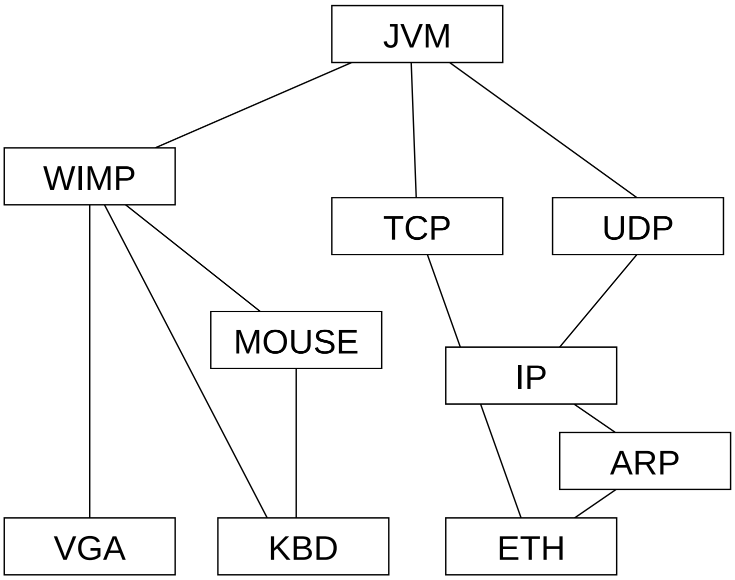

Figure 1: Example Scout Module Graph

To form a complete system, individual modules are connected into a *module graph*: the nodes of the graph correspond to the modules included in the system, and the edges denote the dependencies between these modules. Two modules can be connected by an edge if they support a common *service interface*. These interfaces are typed and enforced by Scout. By configuring Scout with different collections of modules, we can configure kernels for different purposes, including network-attached devices, web and file servers, firewalls and routers, and multimedia displays.

For example, Figure 1 shows an extract of the module graph for a Scout kernel representing a Java appliance. The configuration includes a device driver for the network card (ETH), four conventional network protocols (ARP, IP, UDP and TCP), a Java virtual machine (JVM), and a window manager (WIMP) with a mouse (MOUSE), keyboard (KBD) and graphics card (VGA) device drivers.[1] Such a configuration is specified at build time, and a set of configuration tools assemble the corresponding modules into an executable kernel. A Java system built around this configuration is described more fully in [8].

2.2 Path Abstraction

Scout adds a communication-oriented abstraction—the *path*—to the configurable system just described. Intuitively, a path can be viewed as a logical channel through a modular system over which I/O data flows. In this way, a path is analogous to a virtual circuit that cuts through the nodes of a packet-switched network; in fact, one can think of a path as a continuation of such a circuit through the host OS. In other words, the path abstraction encapsulates data as it moves through the system, for example, from input device to output device. Each path is an object that encapsulates two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

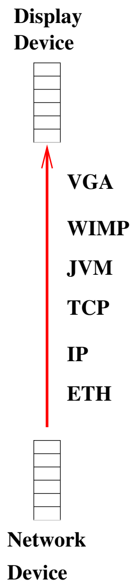

Figure 2: Example Path

Although the module graph is defined at system build time, paths are created and destroyed at run time as I/O connections are opened and closed. Figure 2 schematically depicts a path that traverses the module graph shown in Figure 1; it has a source queue and a sink queue, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries. This particular path processes incoming ethernet packets and displays them on the VGA display.

The path-specific local state of each module is stored in a data structure called a *stage*. Stages from a sequence of modules are combined to form the path data structure. In addition to this path-specific state, when executing code within a certain module, paths also have access to the state of the module. For example, a path executing code of the IP module has access to the routing tables stored in the IP module.

Each path goes through three phases during its lifetime. The first phase is path creation, during which the topology of the path—i.e., the sequence of modules it traverses—is determined, and the state of the path is initialized. Path

[1]The configuration can also include access to the file system, but we have not included these modules to simplify the example.

creation is triggered by a pathCreate call to the kernel. This operation takes a set of attributes and a starting module as arguments. The attributes define invariants for the path; e.g., the port number and IP address for the peer. The kernel then establishes the path incrementally: it invokes an open function on the specified module, which determines the next module to visit. The kernel then calls the open function for this module, and so on.

The open function for each module visited during path creation inspects the attributes to determine which module to visit next. It may also modify, add, or delete attributes. The path creation process terminates for two reasons: (1) the attributes violate some module constraint, or (2) the attributes are not strong enough to determine the next module. In the first case, path creation is denied. In the second case, the maximum length path has been discovered. Note that in the case of a failed path creation, no module state is modified. This is for two reasons: the open functions are side-effect-free, and the modules along a path are not allowed to initialize their path-specific state until the full path is known.

At this point, the path enters its operational phase and data is sent and received over it. Both send and receive work in the obvious way: data is enqueued at one end of the path and a thread is scheduled to execute the path. There is one complication, however. When data arrives on a device—e.g., a network packet arrives on the ethernet—the kernel must determine which path it belongs to. This is done in a way that is analogous to path creation: the kernel identifies the path incrementally by invoking a demux operation on a sequence of modules. Each module's demux function has three choices: (1) it can determine that a unique path has not yet been identified and call the demux function of some adjacent module; (2) it can reject the request and drop the data; or (3) it can return a unique path. As was the case with open, each module's demux function is side-effect free.

The last phase of a path is invoked by a pathDestroy call to the kernel. The kernel invokes a destroy function associated with each module along the path in the same order in which they were initialized.

3 Security Architecture

This section introduces the Escort security architecture that we have designed for Scout. We begin with the module graph configured for a particular network appliance. Figure 3 gives a simple module graph that represents a subset of the graph given in Section 2. It includes a module that implements the Java Virtual Machine (JVM) and the TCP/IP/ETH modules upon which JVM depends. This example is obviously simpler than a full-fledged system would require—for one thing, it is linear—but it does include enough complexity to use as a running example throughout this section.

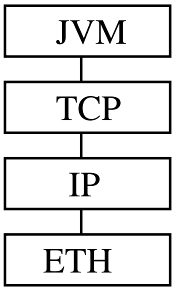

Figure 3: A Simple Module Graph for a Java System

The module graph shown in Figure 3 illustrates two important points about Scout. First, the OS includes only those modules needed by the network appliance; modules not configured into the graph pose no security risk. Second, each module implements a narrow, well-defined service interface; these services are on the order of "TCP" rather than "network subsystem". This means that a given module's service interface is much more restrictive than one would find in a server-based system, and that there is less code to inspect and trust.

3.1 Filters

The first security mechanism Escort adds to Scout are *filters*: automatically generated modules that are inserted in the module graph between the original modules. These filters are configured in the module graph just like the TCP or IP modules in Figure 3. The only difference is that they are generated from a global policy engine; this policy engine is described in Section 3.5. Figure 4 shows our example module graph augmented with filters modules.

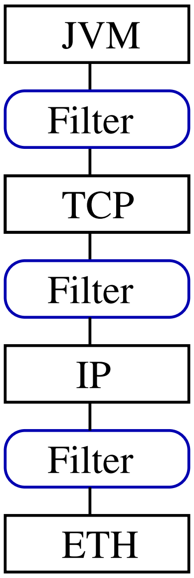

Figure 4: Module Graph Augmented with Filter Modules

The purpose of filter modules is to restrict the interface between the adjacent modules. For example, the filter between TCP and IP might restrict the TCP/IP interface from one that supports "send packets" to one that supports only "send packets to port 1180". The filter enforces this more restricted interface by filtering data that does not adhere to this restriction. Note that these filters can be used in conjunction with a vanilla TCP module, and conversely, the same TCP module can be flanked by different filters; none of the security policy is embedded in the TCP module.

Since filters are just like any other Scout module, they support both an open and a demux function. The former can restrict what paths can be created (a path might be rejected if it implements a connection to a disallowed port), while the latter can filter what packets are allowed to traverse the path at data transmission time. More on this below.

Some filters base their decisions on local knowledge; limiting a TCP module to just one port is an example of such a local decision. There are situations, however, when filters need to make policy decisions based on global runtime knowledge. For example, a policy might prohibit the coexistence of two paths to enforce the principle of least privilege. Escort provides a central *attribute manager* for the purpose of providing filters with access to global knowledge. The attribute manager is implemented in the kernel and is available only from within filter modules. In effect, the attribute manager implements a shared tuple-space that filters can use to communicate with each other. Filters can read and write attributes from and to the attribute manager. The attributes are typed, and the attribute manager uses these types to restrict what attributes a given filter can access. Obviously, the attribute manager introduces the possibility of data leakage between different paths, and therefore, the system designer should use this feature with extreme care.

3.2 Protection Domains

Filters serve to limit module interfaces, but they trust that the adjacent modules will use this interface. That is, filters include no mechanisms to ensure that one module doesn't bypass the interface and directly access the memory of another. (Scout also does not restrict such access since all modules exist in a single address space.) Filters are sufficient if two adjacent modules trust each other, but this is not always the case.

Escort's second mechanism—hardware enforced *protection domains*—addresses this issue. Protection domain boundaries can be drawn between any pair of modules to ensure that access from one module to the next is restricted to that module's well-defined interface. Protection domain switches are implemented as memory faults on function calls. Therefore, the presence or absence of protection domain crossings is transparent to the modules involved.

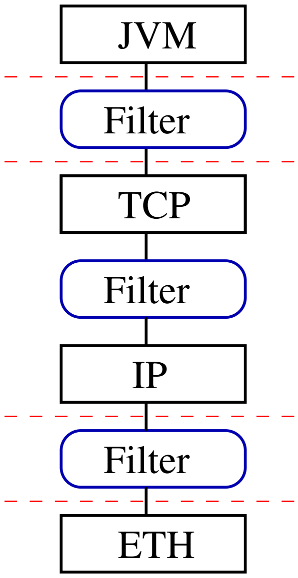

Figure 5: Module Graph Augmented with Protection Domains

A collection of modules, and the filter modules between them, can be placed in a single protection domain as long as they trust each other. A global policy (see Section 3.5) defines which modules are contained in which protection domains. When two modules do not trust each other, a domain boundary is drawn on either side of the filter module that was inserted between them. Figure 5 illustrates how protection domain boundaries might be drawn on our example module graph. In this case, we assume TCP and IP trust each other, but that there is no trust between this collection of modules and JVM. Thus, the filter between JVM and TCP resides in its own protection domain, which allows it to enforce a narrow interface between these two modules.

Device driver modules, such as ETH, have a special status since they are allowed to access hardware devices directly. They do not necessarily have to be separated from other modules, but all modules contained in the device's protection domain will be able to access the address ranges occupied by the hardware device. In our example, we isolate ETH in it's own domain; the filter between ETH and IP is therefore also in its own domain.

Another issue is the containment of misbehaving modules. If the module along some path misbehaves by trying to violate its protection domain boundaries, the protection domain containing this path is removed and pathDestroy is called on all paths that traverse this protection domain. This rather radical measure guarantees that no state that was altered in violation of the policy remains within the system after a module has been identified as misbehaving.

3.3 Paths

As described up to this point, Escort is roughly equivalent to DTOS in the support it provides to build a secure system: it provides mechanisms for fine-grain access control as defined by a global policy. Escort's main advantage is that its modules are finer grain, which means each contains less code to trust and their interfaces are more restrictive. The module graph, filters, and protection domains do nothing to improve end-to-end accountability or isolate flows from each other. That's where paths come into play. Figure 6 schematically depicts two paths traversing the example module graph we've been considering.

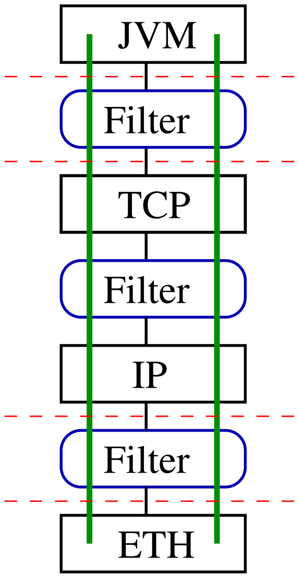

Figure 6: Module Graph with Two Paths

As described in Section 2, paths represent information flows through a modular system, and each path goes through three phases in its lifetime. Escort controls all three phases. Both the creation and destruction phases are controlled by a *path manager*. The path manager, which is implemented in the kernel, restricts the origin of pathCreate and pathDestroy calls to certain modules. In case of pathCreate, it also validates—and possibly adds or removes—the attributes provided by the caller. Keep in mind that the filters included between modules can also check and adjust the attributes passed between modules, as well as deny path creation. The filters are not sufficient, however, since a path is not rooted at a filter module. This is the main reason for separating out the path manager rather than folding it into the filter mechanism.

After a path is created and enters its operational phase, the filters configured into the module graph are used to restrict the data that might traverse the path. It does this by restricting the demultiplexing, as well as the data exchanged between path modules. A filter can also destroy a path or remove data if the data passed between modules indicates that the path is used against its policy.

Filters restrict the data that flows through operational paths, but paths themselves play a central role in resource management. This is because resources are allocated on a per-path basis: CPU utilization is enforced by the scheduler; memory utilization is enforced by the heap; and device access is limited by the module graph, the filter modules, the size of the output queue for a device, and the frequency with which a path is scheduled. The path manager defines the resource limits placed on each path during path creation.

Resource control is made easier by the ability to determine the destination of data at demultiplexing time. No permanent state is changed and only a few cycles are spent to determine if data is destined for a particular path. This prevents priority inversion in network-based systems as long as the demultiplexing decision is fast enough to keep up with incoming traffic.

Escort also encourages the use of appliance-specific schedulers, which use detailed knowledge of the use of the path to make scheduling decisions. In a WWW server, for example, the priority of a path accepting new connections and closing paths might decrease under high load, compared to the paths representing existing connections. In many cases, application specific knowledge can be used to make denial of service attacks substantially harder. The mediation between different application specific schedulers and their automatic generation is a topic of current research.

3.4 Multiple Instantiation

Paths support end-to-end accountability, and provide the machinery needed to separate information flows, but it is still possible for two mutually untrusting paths to flow through a common module and access shared state. To address this problem, Escort allows modules to be multiply instantiated. Figure 7 extends our example to multiply instantiate the JVM module, with each path running through a private copy of this module.

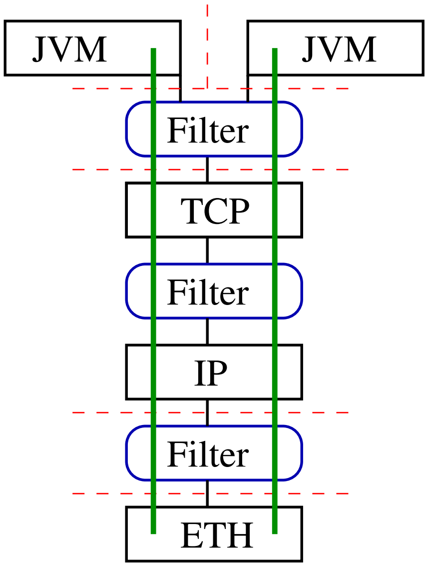

Figure 7: Multiple Instantiation of Modules

The decision as to what modules should be multiply instantiated is up to the appliance designer, and there is a tradeoff. On the one hand, the more modules are multiply instantiated, the less state is shared between paths. On the other hand, there are two drawbacks to multiply instantiating a module. First, since multiply instantiated modules cannot share state, state that must be shared has to either be generated by each module independently, or isolated into a new module that is then secured by an additional filter. Examples of such state include the IP routing table or a TCP port manager. Second, each multiply instantiated module consumes some amount of memory for its local state, although the code segment of each module is shared between all instances, and therefore, does not impose additional overhead.

It is worth noting that multiple instantiation is useful even within one protection domain. If, for example, mutually untrusted Java applets are executing on a Java system, both applets could be run on multi-instantiated JVM modules. Java guarantees the type safety of the individual applets—hence, there is no need to put them in separate protection domains—but the JVM module might not be trusted to separate the runtime state of the two applets.

3.5 Policy Engine

In a network appliance, both the available resources and the applications to be supported, are known at system build time. Escort takes advantage of this constraint to provide a central configuration file that defines the policy for the entire network appliance. Compared with more general systems, such as Unix, the advantage is that there is no separation between operating system and application policy, and therefore, no translation between these two policies is necessary. However, the granularity of the policy is limited by the information available outside the modules since all Escort mechanisms restrict only the external interfaces to these modules.

A *security editor* generates all the security mechanisms described in previous sections from the policy description file (`config.policy`) and the module graph (`config.graph`). Filters, the attribute manager, and the path manager are

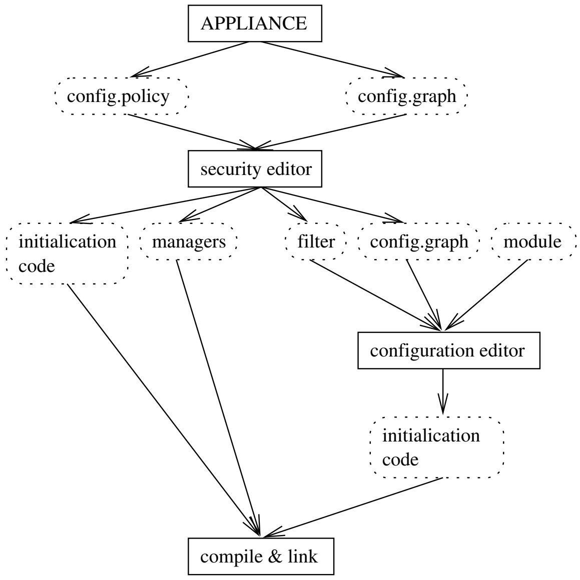

Figure 8: Build process of an Escort secured Scout kernel.

first translated into C code, compiled, and then linked into the Scout kernel. The steps involved in building an Escort secured Scout kernel is shown in Figure 8.

The policy configuration language used in config.policy is limited to particular module interfaces, and is still being developed. The current implementation supports only the netInterface, netListenInterface, and resolverInterface interfaces. These interfaces are sufficient to build firewalls, network routers, DNS servers, and similar low level network oriented devices. The language has to be extended if new interfaces are introduced.

The policy configuration language is low-level, and therefore, can easily be translated into the Escort mechanisms. This makes the definition of higher-level policies like Bell LaPadula [2] rather complicated to configure. We are currently investigating how higher level policies can be translated automatically into our policy configuration language.

It is important to understand that different policy decisions are enforced at different times. First, since Escort exploits the fact that it is used to build specialized network appliances, it is able to restrict communication between modules in the module graph at configuration time. The advantage of this early access control decision is that it does not affect the runtime performance of the system.

Second, at runtime, paths are explicitly created using attributes that define invariants of the path. This pre-allocation of communication paths is again utilized by Scout to optimize the performance of the path, and by Escort to make access control decisions as early as possible. The access control decisions that made at path creation time are of a finer granularity than those made during configuration time since invariants of the path are not available when the system is configured. For example, the IP addresses and TCP port numbers for a Telnet connection are known during path creation, but not during configuration time.

Third, if even finer access control mechanisms are required, Escort provides the mechanism to filter the data between each processing step in a path. This mechanism obviously imposes the highest performance penalty, depending on the complexity of the filter.

4 Example

This section illustrates how Escort works by describing the implementation of a simple web server containing both private documents that have to be protected and public documents that are globally accessible. We first describe the elements shown in the Scout module graph, and then configure an example policy using Escort.

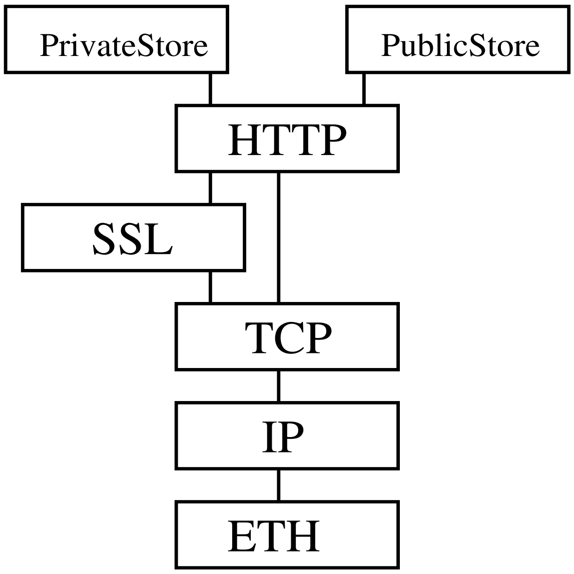

Figure 9: The simple WWW server module graph.

4.1 Module Graph

The module graph of our WWW server is shown in Figure 9. It contains the network device module (ETH), along with modules for the IP, TCP, SSL [6], and HTTP [3] protocols. The configuration includes a PublicStore module and a PrivateStore module that serve the private and public documents, respectively. In a real system, these latter two modules would be implemented by a set of modules representing a local or network file system.

Using only Scout, without Escort policy control, the WWW server would operate as follows. At boot time, the HTTP module opens two paths: one spans the HTTP, SSL, TCP, IP, and ETH modules; the other skips the SSL module and includes TCP, IP, and ETH. Both paths use the netListenInterface to listen for incoming TCP connections to different TCP ports. These ports are set at configuration time. We call these paths "passive" because they are used to listen for incoming connections; they are not used to exchange data.

If a TCP connection is established by a remote site, the TCP or SSL module informs the HTTP module on the corresponding path. The HTTP module then opens an "active" path—one that corresponds to an active TCP connection—all the way from itself to the ETH module, using the same topology as the passive path on which the connection request was received.

After HTTP has received the GetDocument request, it extends the path all the way to either the PublicStore or the PrivateStore module, depending on where the document can be found. The path then starts serving the data. The path eventually destroys itself when the transfer is complete.

4.2 The Policy

There are obviously many possible security policies, depending on the environment in which this WWW server is being used. One of Escort's advantages is that the security policy is defined separately from the appliance's functionality. Thus, the same modules and module graph can be used for many different policies. To illustrate the mechanism introduced by Escort, consider the following simple policy.

Keep Secrets: All documents stored in the PrivateStore should only be transfered via an encrypted channel.

Limit User: Only certain users are allowed access to documents stored in directory /secret/limited/, which is maintained by the PrivateStore module.

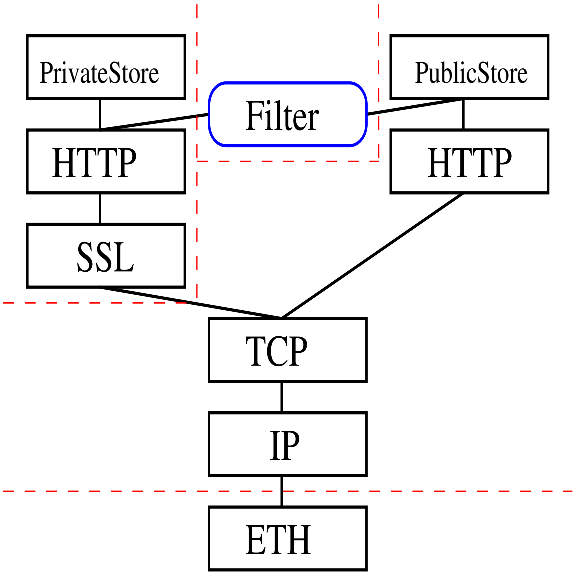

Figure 10: The simple WWW server with multiple protection domains

Prioritize: Secure connections have higher priority than insecure connections. Existing connections have a higher priority than new connections.

This policy does not include all aspects of a real world WWW server, but is small enough to describe. A more complete policy would also deal with other properties, such as restricting the WWW server to GetDocument requests, and restricting IP and TCP to only provide services necessary for a WWW server. It would also deal with possible denial of service attacks in a much more thorough way.

4.3 Enforcing the Policy

The first policy restriction to keep information secret can be enforced in multiple ways with different levels of assurance. One way to do this is to simply include a filter in front of the PrivateStore module that checks during path creation that the SSL module is contained in the path. This method has the drawback that state is shared in the HTTP module between the encrypted and non-encrypted paths.

This sharing can be removed by multi-instantiating the HTTP module and connecting the PrivateStore module only to the HTTP module connected to SSL, and the PublicStore module to both HTTP modules. It is also necessary to include a filter between the secure HTTP and the PublicStore module to disallow the access from the PublicStore module to the secure HTTP module.

This approach makes the assumption that our modules do not try to bypass the established interfaces; i.e., they do not look in the other modules' memory. If this is an unreasonable assumption, we can separate the higher part of the SSL path into its own protection domain. Figure 10 shows the final module graph divided into multiple protection domains. To further increase assurance, we also separate the device from the remaining modules since a device module has limited hardware access.

Now turning our attention to the second policy constraint, we can limit the user by inserting the filter shown in Figure 11 into the encrypting path, just below the HTTP module. This filter destroys the path if any request for a document in directory /secret/limited is received and the attribute manager does not contain an attribute of type ALLOWED_USER that matches the current user. The ALLOWED_USER entries in the attribute manager are either set during configuration time by the system builder, or at runtime by another filter.

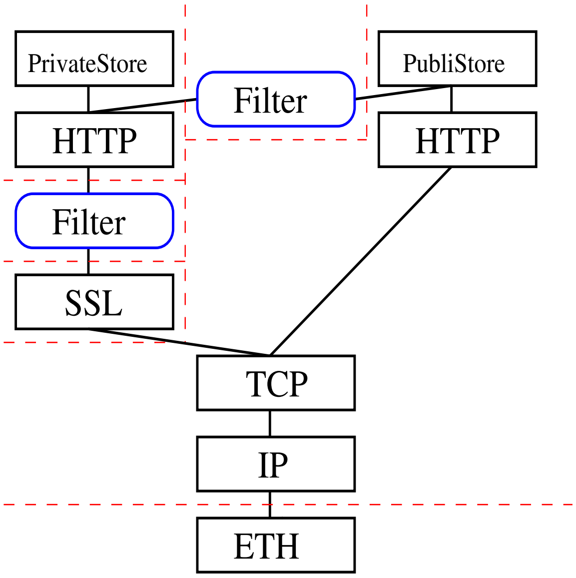

Figure 11: The complete module graph including not empty filters and protection domains.

Figure 12 shows the part of the config.policy file that is used to configure this filter. The first part of the extract describes the functionality of the filter itself. The checkTopPart command checks the user name passed to SSL during path creation and adds the path attribute ALLOWED_PATH to the path if the user matches an ALLOWED_USER entry in the attribute manager. If data is received on any path going through this filter, the checkData command checks if it is not a request to the /secret/limited directory, or if the path is marked ALLOWED_PATH. If the predicate is true, the execution continues. If it is not true, the default action on incoming data is invoked and the path is destroyed. The second default command allows unrestricted demultiplexing and sending of data. In the second part, the restrictAttribute command is used to restrict the access to the ALLOWED_PATH path attribute to the filter between SSL and HTTP.

Finally, prioritization is achieved by using a fixed priority round-robin scheduler with four priority levels. The highest priority is assigned to active secret paths, followed by active non-secret paths, followed by passive secret paths, and finally followed by passive non-secret paths. The priorities are assigned by the path manager at path creation time. Note that it would be possible to extend this scheme to limit the number of total non-secret paths, or to adjust priorities dynamically using filters or a specialized scheduler.

5 Evaluation

Comparing Escort to Unix-like operating systems is like comparing apples to oranges. Scout addresses the need of a network appliance, whereas Unix is designed as a general purpose operating system. Escort can prevent many attacks very early by restricting the system's functionality. This is impossible in a general-purpose operating system. Since Escort is not directly comparable to Unix-like systems, we evaluate its design in terms of the criteria introduced in Section 1. We also discuss the performance impact of Escort's design.

5.1 Economy of Mechanism

Escort addresses the economy of mechanism on different levels. From a global perspective, this property is addressed by the fact that a network appliance is designed for a certain task, and therefore, it provides restricted functionality.

```
SSL/HTTP{
  /* remember if path was created for an ALLOWED_USER */
  checkTopPart(CREATE, REMOTE, INDB(ALLOWED_USER,,),
    addAttribute(PATH,ALLOWED_PATH,TRUE));
  /* allow restricted request */
  checkData(POP,
    NOT(LINEMATCH(*/secret/limited*))
    OR getAttribute(PATH,ALLOWED_PATH),CONTINUE);
  /* if on incoming data no filter expression matches destroy
   * the path */
  default(POP, DESTROY_PATH);
  /* data can be send and demultiplexed */
  default(DEMUX|PUSH, CONTINUE);
}

/* ALLOWED_PATH path attribute is only accessible from SSL/HTTP
 * filter */
restrictAttribute(PATH, ALLOWED_USER, SSL/HTTP, ALL);
```

Figure 12: An extract of config.policy responsible for the filter between HTTPD and SSL.

On a lower level, the kernel that is trusted by all paths within a network appliance is minimal. It currently consists of less than 20,000 lines of C and assembler code. Also, the Scout kernel consists of only a few objects: modules, paths and interfaces. Escort uses the existing module abstraction to introduce filters; it adds only two managers to the main system.

The central config.policy file addresses the economy of mechanisms issue in yet another way. It combines the system and application policy in a central file. A disadvantage is that the policy description language is very low level, which requires complicated config.policy files for complex appliances.

5.2 Fail-Safe defaults

Escort supports fail-safe defaults by the fact that it limits the possible communication flows to the module graph at configuration time. The filters, path manager, and attribute manager also support this property since all operations by a path have to be explicitly allowed in config.policy.

5.3 Complete Mediation

As described in Section 3.4, Escort provides mediation of all resources, ranging from coarse-grain mediation during configuration time, to fine-grain mediation in filters at runtime. This distribution of the mediation mechanism allows good performance despite the fine-grain control.

5.4 Least Privilege

In Scout, an application is implemented by a set of paths. Using the central configuration file, the privileges of these paths is easily made minimal. Escort also supports the principle of least privilege within a single application since each path represents one step in the application's life time. For example, an application that first needs access to the network, and later needs access to the file system has to explicitly request access to each by creating a path, and can destroy the former before creating the latter. Moreover, the filters, by exchanging global information through the shared attribute manger, can enforce this non-concurrent access policy.

5.5 Least Common Mechanism

The principle of least common mechanism is supported by Escort in three different ways. First, the path abstraction reduces the shared state between different threads. In contrast to Unix-like operating systems, where all data in the kernel is shared, Escort identifies subsets of these data structures by defining paths. Second, multiply instantiated modules increase this separations even further by separating the module state of different paths into different data structures. The third level of support is provided by protection domains, which are used to enforce the separation by providing guarantees at the operating system-level that data structures are separated. Shared state is present only in the small kernel, in the form of the path manager and the attribute manager, both of which are under the control of the central policy.

5.6 Accountability

Accountability in Escort is provided on a per-path basis. The attribute manager can be used to collect different log information provided by the filters. To allow persistent and possibly tamper-proof storage of the logs, Escort also provides a specialized log filter that accesses the attribute manager and writes the logs into the path to which it is attached. This can be used to write logs to a disk or a secure site over the network.

5.7 Psychological Acceptability

Paths are a natural choice to express data flows. Therefore, programming with paths and making path-based access control decisions are easy to understand by the network appliance developer. However, the fine-grain access control introduced by filters might become complex. The ongoing research in automatically translating high-level policies into low-level Escort mechanisms is therefore very important.

5.8 Performance

Filtering data flows is a rather expensive operation. Escort addresses this problem by applying the filter at the appropriate time: configuration time, path creation time, demultiplexing time, and path execution time. This allows us to keep the filters imposed in each path during execution time minimal.

After a path is created, it is also possible to use the invariants defining the path to optimize the code fragments contained in this path. Filters are especially useful during this process, as they define additional invariants that can be used to specialize the path. These optimization opportunities are still topic of current research.

6 Related Work

On the surface, Escort and Scout have similarities with Unix pipes [13], assured pipes [5], Corps [16], DaCapo [7] and Nemesis [12]. Of these, Corps is the only one that has explicit paths, and none of them uses explicit paths to limit and separate data flows.

Capability-based systems like KeyKOS [9], TMACH [15] and DTOS [10] are also similar to Escort since a path represents a capability to access data and execute code at each module. However, paths have the advantage of supporting global access control and resource allocation decisions before any state is changed.

Our path creation process, which requires a specification of a set of attributes (invariants) for the path is similar to the query mechanism in PolicyMaker [4]. Specifically Scout's path attributes are analogous to PolicyMaker's action string and its annotations. However, Escort generates and checks the action string and annotations in one phase instead of two phases, and the processing order of the filter is explicitly limited by the module graph and determined by the module not by keys and assertions.

7 Concluding Remarks

Escort is a security architecture based on the idea of securing data flows instead of processes or users. It uses Scout's explicit path abstraction to achieve that goal. Escort also provides the low-level mechanisms necessary to give the appliance programmer fine-grain control over all aspects of the system, and addresses the implementation of high-level security policies by providing translation tools to these low-level mechanisms.

Currently, the policy translation tools are only functional for small application domains. All of the Escort mechanisms have been implemented on a Digital Alpha workstation, but they have not yet been integrated. What also remains to be done is to demonstrate a fully integrated implementation of Escort on a wide set of domains. For example, work on an Escort-based firewall is currently under way.

Acknowledgments

We would like to thank Brady Montz, Inge Pudell-Spatscheck and the members of the Scout group for there feedback and support. This work supported in part by DARPA contracts DABT63-95-C-0075 and MailBox, and NSF grant NCR-9204393.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.

[3] T. Berners-Lee, R. Fielding, and H. Nielsen. RFC 1945: Hypertext transfer protocol — HTTP/1.0, May 1996.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *SympSecPr*, Research in Security and Privacy, Oakland, CA, May 1996. IEEECSP.

[5] W. E. Boebert and R. Y. Kain. A practical alternative to hierachical inetgrity policies. In *Proceedings 8th National Computing Security Conference*, October 1985.

[6] A. Freier, P. Karlton, and P. Koch. The SSL Protocol. *IETF Internet Draft*, pages 1–63, Nov. 1996.

[7] A. Gotti. The da capo communication system. Technical report, Swiss Federal Institute of Technology, Zuerich, Switzerland, June 1994.

[8] J. Hartman, L. Peterson, A. Bavier, P. Bridges, B. Montz, R. Pilz, T. Proebsting, and O. Spatscheck. Joust: A platform for communication-oriented liquid software. Technical Report TR97, The Department of Computer Science, University of Arizona, Nov. 1997.

[9] C. Landau. Security in a Secure Capability-Based System. *Operating Systems Review*, 23(4):2–4, Oct. 1989.

[10] S. E. Minear. Providing policy control over object operations in a Mach-Based system. In USENIX Association, editor, *Proceedings of the fifth USENIX UNIX Security Symposium: June 5–7, 1995, Salt Lake City, Utah, USA*, pages 141–156, Berkeley, CA, USA, June 1995. USENIX.

[11] D. Mosberger and L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of OSDI '96*, October 1996.

[12] D. Reed, A. Donnelly, and R. Fairbairns. Nemesis the kernel, Sept. 1997.

[13] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[14] Saltzer and Schroeder. The protection of information in computer systems. *Proc.of the IEEE*, 63(9), Sept. 1975.

[15] Trusted mach philosophy of protection, May 1993. NIST Document No: TMACH 93-014.

[16] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 36–45, Laguna Beach, Ca, Feb. 1996.

[17] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 116–128, Saint Malo, France, Oct. 1997.