

Seraphim: Dynamic Interoperable Security Architecture for Active Networks*

Roy H. Campbell, Zhaoyu Liu, M. Dennis Mickunas, Prasad Naldurg, Seung Yi
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{roy, zhaoyu, mickunas, naldurg, seungyi}@cs.uiuc.edu

Abstract

Security is an important concern in the active networking paradigm because a breach in security can quickly compromise many systems in the network. This paper describes an extensible, reconfigurable security architecture that is flexible and accommodates a wide variety of security policies and mechanisms. It provides applications and users the ability to create and enforce highly customized and situational policies dynamically, and is well-suited to the security issues in active networks.

Seraphim [7] implements this architecture and allows the creation of dynamic security policies. Innovative applications use these policies in an exploration of the nature and scope of “dynamic security”. The implementation facilitates research of interoperability and portability security issues. Based on the experience from this effort, we are investigating a unified model for security mechanisms that preserves security guarantees across domains.

Keywords: *active networks, security, policy, access control, active capability, reference monitor, interoperability, dynamic, reconfigurable*

1 Introduction

Active networks aim to provide a software framework that enables network applications to customize the processing of their communications. Applications encapsulate the methods that manipulate the data, with or without the data itself, and inject these capsules into the network. Active routers install and execute these capsules on the data dynamically, thereby facilitating fast protocol and service deployment. Securing this infrastructure against threats and exposures remains a major challenge in this paradigm.

The traditional definition of security includes authentication, access control, and encryption. Active network applications and routers can establish a basis for trust through mutual authentication. Encryption and digital signatures can protect the privacy and integrity of the active network capsules that contain code and data. Access control mechanisms and security policies can provide

controlled access to router resources and routed code and data. Much security research for active networks focuses on providing a secure environment for the routers that primarily

- prevents malicious behavior of arbitrary user code and
- protects the user code and data from malicious routers [22].

Research at MIT identifies the use of active networks to locate and neutralize the source of unwanted network traffic like ping and ack packets [25]. This kind of application of active networks suggests the possibility of developing “active security”, programmable security policies and mechanisms that respond dynamically to the activities on the network.

Our research complements these efforts and emphasizes the possible applications of active security in an active network. In the course of our research, we have studied applications of active security to the issues of interoperability and dynamic security policies. Most existing security provisions, especially regarding policies and access control, are static in nature. Once a security system is deployed within a network, the provisions are difficult to change and modify dynamically. For example, most security systems cannot change security policy implementations in response to a successful security attack. Though a wide range of security policy types have been proposed, most systems implement a common subset of these policies and mechanisms. Applications that require sophisticated security requirements and customized security policies must use lesser or weaker security guarantees provided by the deployed static security system. In the spirit of the motivation for developing active networks, our approach provides a unified security framework that allows users or applications to create and enforce their own security provisions and policies, similar to customizing their own communication protocols. We identify two example application scenarios that benefit from such flexible security support:

- **Dynamic Firewall Formation:** The creation of dynamic protection domains or enclaves is very useful

*This research is supported by DARPA F30602-98-1-0192

in many situations. In this way, a security system can build agile and dynamic firewalls in reaction to the detection of a security attack, thereby isolating the target of the attack from its attacker. When an active security administrator or trusted authority detects intrusion, it can send out an active capsule carrying a security agent with the appropriate vaccine. This can be used to build a dynamic line of defense against outside attacks and to raise the level of security within the domain. When the threat disappears the administrator or trusted authority can transmit another active capsule in order to resume normal operation. This can be used as a very powerful security tool in conjunction with intrusion detection and countermeasure systems.

- **Secure Emergency Multicasts:** Emergency notification of events like a storm warning must be secure to be effective. We describe a “geo-casting” scheme to alert or warn subscribers about natural disasters like tornadoes passing through a geographic region. A multicast group is formed using dynamic join and leave, based on geographic information. As the tornado moves, the multicast group can move along with it by adding and removing appropriate receivers in real time, using a fast join and leave. Active capsules can be sent to meteorological “subscribers” in a larger area, or to mobile users to warn that they are entering a danger area. Active security is used to reduce false alarms and to prevent unauthorized use of the system.

In this paper, we present a dynamic, fully extensible, interoperable security architecture based on and built into the underlying active network architecture. This architecture allows the configuration of existing active network routers with only a minimal set of security functions. These functions are used to recursively install and support the secure deployment of new security mechanisms. For instance, sophisticated and application-specific or user-specific security functions may be installed at run time using a secure recursive reconfigurable bootstrapping process. Currently, our framework provides mechanisms to specify, separate and enforce a number of different and often mutually exclusive access control policies dynamically. In addition we allow applications to encapsulate credentials and to encode situational policies that are authenticated by a trusted policy server to add, alter or revoke existing access control rules and mechanisms dynamically. Applications write active network code which uses these credentials and policies to inject customized security into the routers. Much of the architectural framework has been built and tested in Seraphim [7, 16]. Thus, applications may choose an access control policy and enforce this policy on their active network code. The framework can provide consistent security policy guarantees and platform independent enforcement of security policies across all active routers.

Section 2 of the paper gives a brief overview of the im-

portant terms and presents a self contained tutorial on some of the background material. Section 3 describes our architecture and talks about its place in the general active network architecture. Section 4 focuses on the implementation of the reference monitor and its flexible policy framework. This allows us to create and enforce dynamic policies. Section 5 talks about our testbed implementation and some of the experiments that we have performed using our testbed. Section 6 talks about related work and the final section talks about our conclusions.

2 Background

Although our research investigates dynamic security provisions for all aspects of security including authentication, access control, and encryption, in this paper we emphasize access control. Access control is the mechanism by which a security system exercises control over the access and utilization of shared resources. Historically access control has been defined in terms of $\langle \textit{subject}, \textit{object} \rangle$ tuples and access control matrices. Typically the matrix is indexed by the name of the user (the subject) and by the resource that needs to be protected (the object). The intersection of this pair contains a Boolean value that indicates whether the access is allowed or denied. (The method is usually encoded implicitly in the Boolean value.) For example, Unix file systems use 3 bits to encode various combinations of read, write, and execute permissions for files. However, this matrix method of implementation does not scale to systems that serve a large number of users with a large number of resources.

The security policy associated with an access control mechanism refers to the characteristics of the security that it enforces. A variety of types of access control policies have been defined in the literature including Mandatory Access Control (MAC), Discretionary Access Control (DAC), Double Discretionary Access Control (DDAC) and Role Based Access Control (RBAC).

The simplest form of access control is DAC which may be represented directly by the matrix model. Typically a DAC policy implementation maintains an indexed list of allowed $\langle \textit{subject}, \textit{object}, \textit{operation} \rangle$ triples. Unix file system permission is a simplified example of a DAC policy. DDAC maintains two lists, an “allowed list” similar to DAC and a “denied list”. MAC policies use the concept of labeling. A military example of labeling has labels “Top Secret”, “Secret”, “Confidential”, “Classified” and “Unclassified”. MAC is used in trusted operating systems. Every entity in the MAC system is assigned an immutable label. A hierarchy is defined in terms of these labels, and access control is enforced by comparing the labels. Subjects with higher labels have access permissions that allow them to write to objects with equal or greater labels only. Subjects with lower labels cannot read from objects with higher labels. This is often called the “no read up, no write down” rule [11]. This hierarchy strictly controls the flow of information.

Among the policies listed, RBAC is the most flexible

type of access control policy [23]. All RBAC subjects are assigned roles. Each role represents a particular set of objects and the allowed operations on each object. The major benefits of this aggregation are the considerable saving in terms of space and simplification in terms of management and enforcement. RBAC allows users to create policies with more sophisticated specifications than simple DAC, DDAC or MAC. A single user may have many different roles, and different permissions depending on the current role. Different constraints related to role and privilege may be enforced in RBAC.

Traditional systems provide a static implementation of any one of these access control mechanisms. For example, system security cannot be dynamically changed from a DAC policy to a MAC policy. Different applications with different access control policies cannot co-exist. Typically, applications cannot be ported across different systems without compromising the security guarantees offered by their access control mechanisms.

3 The Architecture

This section gives a brief overview of the basic active network architecture [6] as proposed by the architecture working group to provide a context for our security architecture. The software on an active router consists of three distinct, functionally separate layers: the application, the EE (Execution Environment), and the NodeOS. The NodeOS is similar to the kernel of traditional operating system. On an active network router, this component also performs resource allocation and management. Typical resources include shared memory, communication channels, and routing tables. The EE runs on a NodeOS and provides an interpreter for capsule code. An EE behaves like a user shell that has access to and can manipulate routing tables and packets. The EE provides an interface for accessing the NodeOS resources. Applications create capsules that include both the code to run on the EEs and communication data. The active network installs and executes the capsule code dynamically on remote routers.

Following is a brief and self-contained overview of our security architecture which is called Seraphim. The major components of our architecture and their interactions in the context of the active network architecture are shown in Figure 1.

The key component of Seraphim is a reference monitor. The reference monitor is implemented as a co-located extension to the Node OS. Every node has a reference monitor through which all accesses to the node resources occur. The policy framework is a component of the reference monitor. The policy framework itself is reconfigurable and it can be downloaded dynamically when required. Applications or administrators use the interface provided by the policy framework to create a customized piece of code that encodes the type of access control policy and other constraints used in the access control decision making process. This code fragment is called the *active*

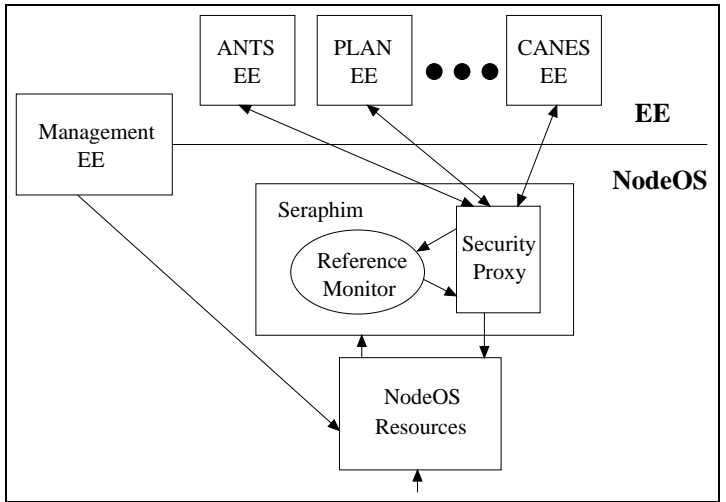


Figure 1: Secure Active Network Node

capability (AC) [16, 9, 8].

Unlike a traditional capability, which is merely a static authorization credential that encodes the principal and the permissions associated with the principal, an active capability is actually an executable Java bytecode in our implementation. In addition, an active capability is protected by digital signatures, resides in user space, and can be freely passed around. Conceptually, an active capability is a piece of unforgeable code that encodes a critical, application-specific part of the decision making code used in access control.

By using an active capability we can encode various situational policies that depend on system attributes. For instance, by writing a piece of code that checks the current system time and compares it with a value stored in the active capability we can introduce a policy that expires after a certain time deadline. Similarly, various enforcement and revocation schemes based on other attributes like quota, history, and information content can be implemented. These schemes are very useful in an open internetworking environment with diverse application requirements. An application can use quota-based revocation to limit the amount of system resources a client can consume. This is useful to counter denial of service attacks.

An active capability relies on a policy framework for context. An application presents an active capability along with its regular data or protocol capsules to the active router's reference monitor at execution time. The access control policy type and user credentials are extracted from the capability. The remote router's reference monitor recreates the context of the policy type within its policy framework. If at any point during this process, the policy framework discovers that it does not have an implementation for the type of the policy, it downloads the code dynamically into the framework, using the underlying active network. It then instantiates the run-time parameters associated with the application in its sandbox-like environment and executes the active capability in this

environment. Based on the result of the evaluation of this active capability, the access control decision is enforced.

The principal of the active capability, which is typically an application user, must be authenticated by a trusted authority. The trusted authority also acts as the policy server in our system. This entity is responsible for generating and keeping track of the active capabilities. Usually, we associate one or more policy servers with each protection domain. Application programs contact their nearest or least-loaded server and obtain the active capability dynamically.

A security proxy component was added as a temporary module in our design. Presently, the active network community is still working on the specifications of a standardized NodeOS interface [20]. In order to provide interoperability between an application written for any EE and our reference monitor, we need an entity that intercepts the requests to NodeOS resources and redirects them to the reference monitor. At this point the EEs direct their requests for NodeOS resources to the security proxy which sits on top of the NodeOS. The proxy acts as a wrapper to the NodeOS API and redirects access requests to the reference monitor. The reference monitor evaluates the request and passes the result on to the proxy. Depending on the result the proxy either forwards the request to the NodeOS or returns it to the EE with a denial notification.

The next section describes the reference monitor, the policy framework and active capabilities in detail.

4 Active Capabilities, Policy Framework, and Reference Monitor

In our approach, active capabilities distribute access control information including security policies within an active network. Security provisions are componentized so that complex policies and controls may be dynamically downloaded component by component. Active capabilities (ACs) are capsules (or part of capsules) that encapsulate security code like a security policy or access control decision. Policy servers operate as a communication front-end for distributing executable security policies in the form of ACs. An AC may either provide all the code for a security policy or access control, or it may specify a policy server from which to retrieve code. A reference monitor is used to intercept application and active network capsule code resource requests. The reference monitor applies the appropriate security access controls to resource requests. As the access controls are applied, the security code in an AC may request further policies that must be downloaded from a policy server.

Typically, traditional security systems are designed to enforce one particular type of security policy like MAC or DAC. Security policies are usually static and are not easy to change once deployed. In many cases, the security policies are specified in a policy language and com-

piled to an implementation that provides access control. In our approach, security policies are mobile agents or downloadable executable code in the form of ACs. In order to help users with policy specification, we provide an object-oriented policy representation framework in Java. The policy representation framework consists of a hierarchy of classes as shown in Figure 2.

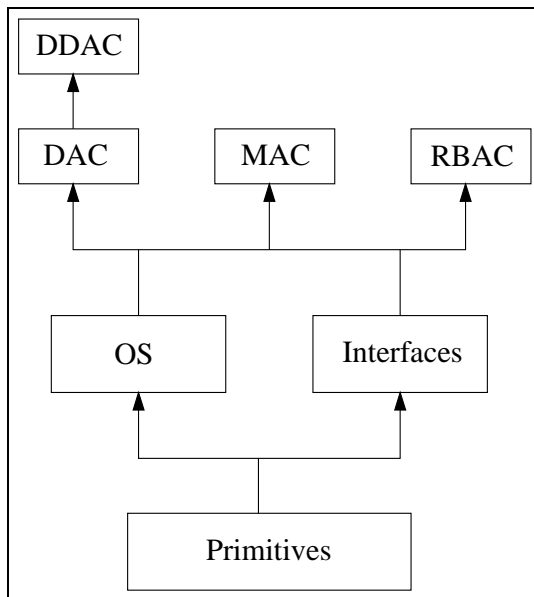


Figure 2: Component-level Map of the Policy Framework

The classes at the bottom of the framework are mostly abstract and are mainly used to represent mathematical concepts such as sets and mappings. These classes form the basis for a hierarchy of successively more specialized classes representing concepts such as labels and access control lists. At the top of the framework are classes which can be used to represent a variety of generic policy forms.

A policy framework that places a heavy burden on its users will not be popular. With this in mind, we provide a policy framework GUI which makes the process of creating new policies or specializing existing policies as painless as possible. Typically, to use a security policies, most users will just select one of a list of predefined policies or use the default settings chosen by a system administrator. However, our approach also allows system administrators and expert users to create and modify policies that respond to specific application needs or security threats. The goals of our policy framework are to allow predefined policies to be enforced efficiently and effortlessly and also to provide a convenient interface for policy authors to create more sophisticated policies.

The current policy framework supports the following common types of access control policies: Mandatory Access Control (MAC), Discretionary Access Control (DAC), Double Discretionary Access Control (DDAC), and Role-based Access Control (RBAC) [21]. More application specific access control policy systems can be easily extended from this object-oriented framework ([14

provides several good examples). In our model, we can specify not only the $\langle \text{subject}, \text{object}, \text{operation} \rangle$ access control triple, but also include a resource limit on usage, situational decision rules, constraints and dependences, e.g., based on current time of the day or current role of the principal.

Our framework also lets users specify pre-conditions and post-conditions. Pre-conditions allow necessary security checks to be performed before the actions take place, and post-conditions can be used to maintain state and perform additional checks after the action has been completed and when more information becomes available.

The policy framework is used in both the domain policy administrator and the reference monitor. Figure 3 shows the interaction of the various components of the policy administration mechanism.

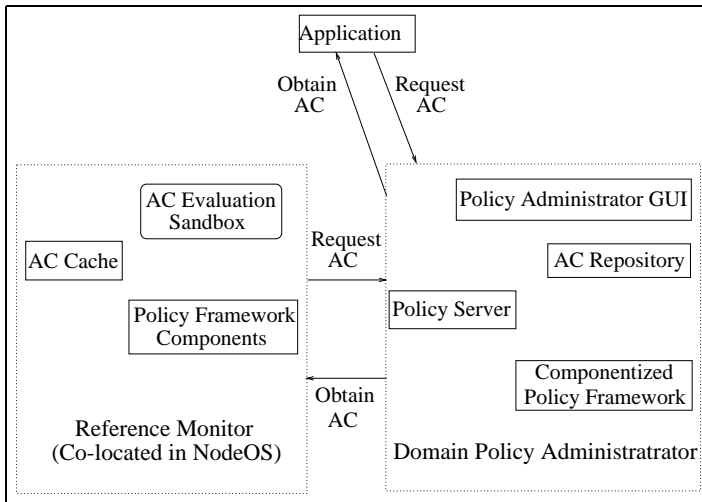


Figure 3: Policy Administration

There are three ways to pass an AC to the reference monitor:

- The applications can create application specific ACs or obtain them from the policy server and then send the AC along with active capsules. The AC may be embedded into the active capsule, or it may be an active capsule itself. When a capsule arrives at a remote node, it is demultiplexed to the appropriate EE, which maintains the state concerning the capsule and recognizes protocols and flows. The EE presents the AC to the security proxy along with its request to a NodeOS resource.
- If the application capsule does not have an AC, upon receiving the resource request via the EE, the reference monitor contacts the domain policy server directly, and asks for the AC associated with the principal of the application capsule.
- For common applications or frequent users, the policy server may distribute the ACs in advance to the reference monitors during system initialization.

For frequently occurring operations like IP forwarding, dynamically changing capabilities are not necessary. Access control rules tend to be static and caching provides a short circuit or fast path processing alternative for such requests. On the other hand, our application section demonstrates some protocols that can benefit greatly by using the expressibility afforded by the power of dynamic capabilities.

In order to improve the AC evaluation efficiency, the reference monitor uses a cache to store the ACs, or even the result of AC evaluations. Depending on the freshness and type of the AC, a request may be satisfied by a simple cache lookup instead of an expensive AC evaluation. On the other hand, for some types of capabilities, the reference monitor can always download the latest capability from the policy server. Caches are purged periodically to maintain their freshness. We plan to improve the efficiency and optimize the cache consistency protocol used in our architecture.

Another important attribute of this architecture is the ability of the trusted authority to revoke a capability at any point in time. The trusted authority can send a “purge cache” message to the relevant reference monitors and install a new capability at run time. Alternately, the application can present a properly signed new capability during run-time with a newer version number which invalidates the existing capability.

4.1 Discussion

We are using the JDK1.2 security API to do simple key generation and management and AC authentication. We plan to leverage ongoing work in this area, and integrate it with our system. Of particular interest are the systems being developed by Network Associates, Inc. [17] and KeyNote [1]. Using the attributes of the two systems we plan to define an infrastructure that again allows users to pick and choose the best attributes from either system dynamically.

Another problem that needs to be addressed is low-level code safety in the reference monitor. The minimum requirements for low-level code safety are control flow safety, memory safety, and stack safety [15]. Currently we rely on the Java byte code verifier [28] to provide low-level code safety. Before loading a class, the verifier performs data-flow analysis on the class code to verify that it is type safe and that all control-flow instructions jump to valid locations.

There are several other approaches for low-level code safety. The PLAN project [1] uses programming language techniques to address the code safety problem. Capsules are written using a strongly typed, resource limited language and dynamic code extensions are secured by using type safety and other mechanisms. Another approach is Proof-Carrying Code (PCC) [19]. Besides regular program code, PCC carries a proof that the program satisfies certain properties. The proof is verified before the execution of the code. The generation of a proof may be complex and time consuming, while its verification should

be simple and efficient. Software fault isolation (SFI) [26] provides another alternative for low-level code safety. It uses special code transformations and bit masks to ensure that memory operations and jumps access only the correct memory ranges.

Here again, a variety of different mechanisms and protocols have been proposed. Each method has its own advantages and disadvantages. Ultimately the application must be given the choice to pick the mechanism that is most suitable for its purpose. We plan that in the future our framework will be generic enough to allow all these mechanisms to co-exist, using the same principles that guided the design of our experimental policy framework.

5 Experimental Testbed

Our initial testbed implementation is based on the ANTS toolkit developed at MIT [27]. The original toolkit was written before the architecture group was formed, and did not reflect the layered architecture proposed by the architecture working group. Our first task was to split the design into layers and separate the functionality of the original Node class into distinct NodeOS and EE components. We added the Seraphim reference monitor between the EE and the NodeOS. We called the modified system SAINTS (Secure Active Inter-operable Network Toolkit System). Our SAINTS is backwards compatible with original ANTS and can run original ANTS applications¹.

5.1 Using the Testbed

In order to use our testbed, the policy server has to be initialized first. The policy server is the trusted third party for the testbed. It acts as a front-end to the policy framework classes and allows applications to create active capabilities. Currently we do not provide support for dynamic policy negotiation but allow multiple security domains to exist. When the policy server is started, it also starts the policy administrator. The policy administrator starts a GUI which allows users or system administrators to create and define policy-specific attributes and generate active capabilities. Users of system administrator can choose any policy type from DAC, DDAC, MAC, or RBAC. In this section, we are going to show the usage of the DAC policy, the most simple one, in the Gnipper application, and then the usage of the RBAC policy, the most flexible and complicated one, in the dynamic secure multicast application.

A screenshot of the GUI for DAC is shown in Figure 4. The user or administrator selects and sets the policy type to DAC. In order to create a new active capability, the user types in a file extension and clicks on the “New DAC File” button. To reuse existing capabilities, the “Load DAC File” button is used, after specifying the file extension. To add (remove) ACs into policy specification the user name, the object and the allowed operations on that object are all entered in the appropriate

¹Contact authors for the Seraphim software release

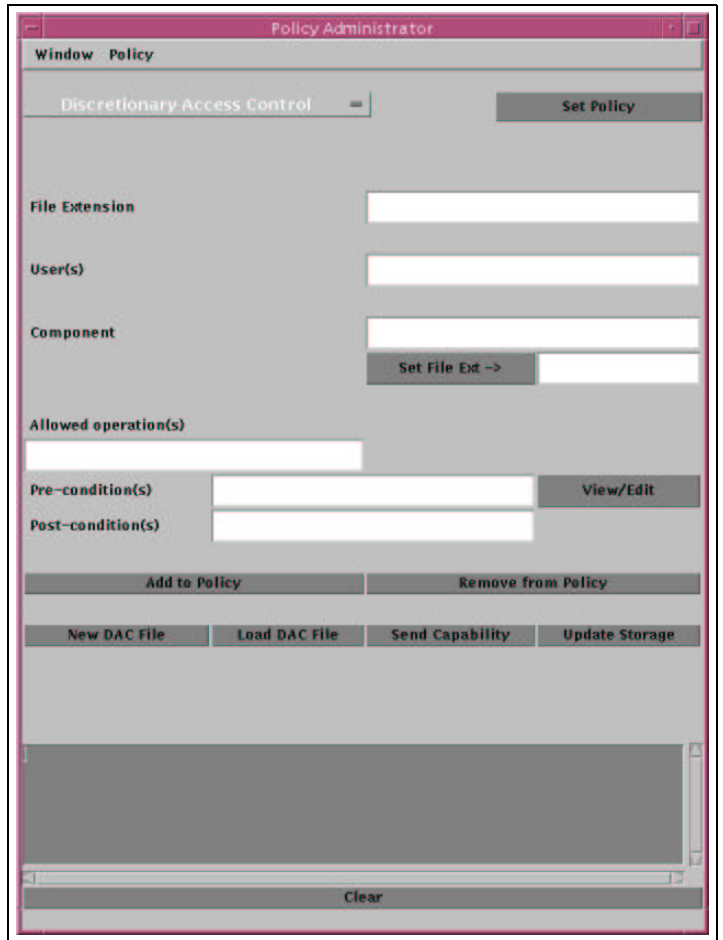


Figure 4: Policy Administrator GUI for DAC

fields and the “Add to Policy” (“Remove from Policy”) button is clicked. Post-conditions and pre-conditions are also added if necessary. The ACs can be stored using the “Update Storage” button, and be retrieved at any point in time, by using the “Load DAC File” command. The “Send Capability” button is used to send the ACs to a particular user or reference monitor. To test the policy specification the GUI also provides an “Evaluate” option in the “Policy” menu.

The GUIs for the other policy types are similar to the DAC GUI. A screenshot of the RBAC GUI is shown in Figure 5. The RBAC GUI supports more functionality and allows the administrator to create role definitions and to associate users and permissions with the role.

5.2 Applications

As a part of our testing and development phase, we developed several interesting, yet conceptually simple, applications on our testbed to demonstrate the significant advantages of our architecture. These include the Gnipper application and the secure multicast applications. The next two subsections give a brief overview of these experiments.

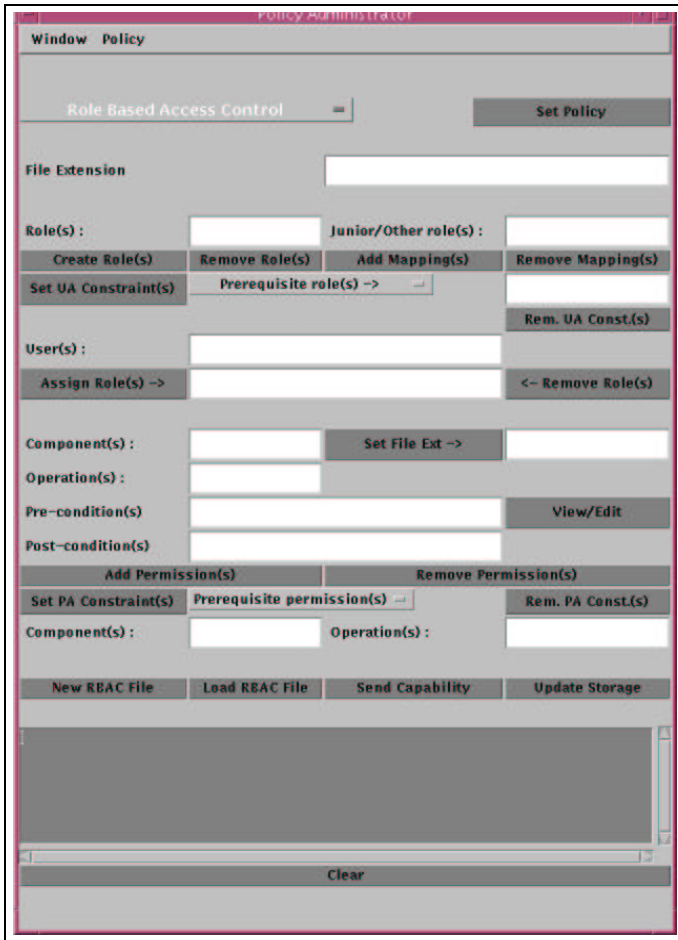


Figure 5: Policy Administrator GUI for RBAC

5.2.1 Gnipper

The Gnipper experiment demonstrates the creation of dynamic protection domains or enclaves. This is best explained with the help of an example presented below (Figure 6).

In our example, User U at Node A is trying to discover the network topology and sends out a Ping capsule with destination address of Node D. Ping packets may be unwelcome because they may be used in a denial of service attack or because of privacy. If we decide that Node D should be secure from Ping requests from User U at Node A, then we create a Gnipper vaccine and install it at the

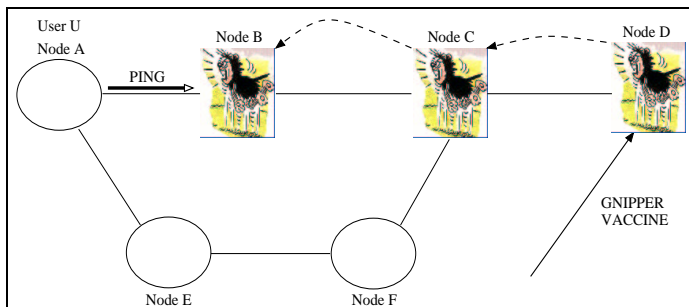


Figure 6: Gnipper Application

reference monitor of Node D. The vaccine, an anti-Ping AC, will disable the ability of User U at Node A to ping Node D. So when the Ping capsule from User U arrives Node D, Node D drops the capsule and propagates the vaccine to the previous node, Node C. Now Node C is vaccinated. When user U sends another Ping capsule to Node D, the capsule will be dropped at Node C and the vaccine will be installed at Node B. So the vaccine dynamically moves one-hop at a time toward the source of the Ping, in response to Ping requests for original sender. It is important to note here that the vaccine is reversible and the system administrator or trusted authority can send another active capability and install it on Node D dynamically, purge the caches on Node B and Node C and allow Ping from User U at Node A to reach Node D. Then the normal execution of the Ping is resumed.

The exact node which drops the Ping request changes dynamically depending on the number of Pings generated by the source and the pinging routes. For example, although Node E and Node F can also be used for Ping, the vaccine is not installed on those nodes because they were not used in the previous Ping attempt from Node A. By selectively broadcasting the vaccine on the frequently used routes between Node A and Node D, we have succeeded in building a dynamically growing firewall around Node D, and also reduced the traffic and moved the denial of service attack away from the intended victim.

Although our prototype implementation was not built for performance, we did make preliminary performance measurement (Refer to [16] for more performance measurements). The average overhead for an application running on a Sun SparcStation 10 machine to create a vaccine and install it at another Sun SparcStation 10 machine on the same 100Mbps local Ethernet LAN was measured as 77ms. Without creation, the average time to send a vaccine and install it under the same setup took 34ms.

This experiment can be extended to build agile and dynamic firewalls that can react to attacks at runtime. When an active node or trusted agent detects attempted attacks, it can send out an active capability carrying a “warning” message with the appropriate vaccine, to build a dynamic line of defense against outside attacks or to raise the level of security within the domain. Similarly, a firewall can be built dynamically around a compromised node to isolate the victim. When the threat is gone the active node or the trusted agent can send out another active capability to resume normal operation. This can be used as a very powerful security tool in conjunction with intrusion detection and countermeasure systems.

5.2.2 Dynamic Secure Multicast

The dynamic multicast application was intended to showcase the benefits of using the RBAC policy implementation and to demonstrate the creation of dynamic multicast groups. In addition it demonstrates a range of different situational specifications using active capabilities.

Most of the existing secure multicast schemes are based on sharing a secret session key among the subscriber

nodes to ensure privacy of data. The main problem with this approach is the prohibitive overhead associated with the need to change the session keys whenever a person leaves the group. This is necessary, in order to make sure malicious members do not continue to listen to multicast data. In order to facilitate dynamic joining and leaving, we use active capabilities to grant and revoke users access to sensitive multicast data in our experiment. When a user joins, the trusted authority installs an active capability that gives the user privileges to receive the multicast data, at the reference monitor of the user's local node. When the user leaves the group, the active capability for the user is simply revoked by the trusted authority. Our experimental scheme has a much lower overhead compared to the traditional schemes.

The multicast program we used is a modified version of the sample multicast application in the original ANTS toolkit. Using our modified multicast application, we devised two different scenarios. The first one is a cable-TV style "Pay-Per-View". A user who wishes to receive a sequence of special multicast packets contacts the trusted authority and obtains an active capability that has a resource limit built into it. This capability is then installed in the reference monitor of the user's node. Every time a special data packet is delivered to the node, the resource limit is decremented by one. When the resource limit reaches zero, the active capability expires and user can no longer receive the special multicast data traffic. If the user wishes to receive more, then the user has to pay again and get another active capability with the appropriate resource limit.

The actual implementation was done using role based access control (RBAC) policy. Users were assigned a default role, that did not let them receive any of the special multicast data packets. Once they "paid", their role was replaced by a "special" role that gave them the access rights for a predetermined number of special multicast data packets. When the resource limit reached zero, the "special" role was expired and the original default role was resumed.

The second scenario demonstrates the use of time-stamped active capabilities for control access. This experiment is a cable-TV style "Sneak-preview". Any user in the multicast group obtains, say, two minutes worth of free multicast data. Active capabilities are obtained and installed in the similar way as in the first scenario. Once installed, the active capability keeps track of the local time and expires after two minutes have elapsed. We are assuming here that applications cannot alter the local time, as it is a protected resource.

Both the "Pay-Per-View" and "Sneak-preview" experiments dramatically reduce the amount of state maintained by the server, compared to the state maintained by existing traditional secure multicast solutions. By eliminating the need for secret session keys, these experiments distribute the server processing load among all the participating routers and allow asynchronous, client-side initiated joining and leaving.

Based on our current efficient, dynamic, and secure multicast, we plan to conduct a "geo-casting" experiment that alerts or warns subscribers about natural disasters like tornadoes passing through a geographic region. A multicast group is formed using our dynamic join and leave, based on geographic information. As the tornado moves, the multicast group also moves by adding and removing appropriate receivers in real time, using our efficient join and leave. The proper active capabilities can be supplied to meteorologist "subscribers" in a larger area, or to mobile users to warn that they are entering a danger area.

5.2.3 Other Applications

Currently we are integrating our Seraphim security system into the CANEs[2] congestion control and error recovery multicast application. We use our RBAC policy to control the signaling procedure of CANEs, and to control the installations of different protocols dynamically. We also use our framework to control access to data packets dynamically.

We also provide Seraphim security services to an NS2 simulation [3] of a new secure multicast routing protocol. In the NS2 simulation, some multicast groups need higher security routing. The MAC policy of Seraphim assigns different security clearance levels to routers. The system then obtains the security information of the routers in the network, for a particular multicast group, based on their security clearance level. The simulation uses this information to set up proper secure multicast routing trees. The Seraphim reference monitor functions as the enforcement engine and ensures that the multicast joins follow the established security level hierarchy.

6 Related Work and Discussion

Little research has been done in security policy management and domain interoperability. In traditional systems, security policy defines access control which is enforced by enforcement engines such as reference monitors in operating systems and firewalls in networks. Individual policies can be defined at each enforcement point and managed separately or centrally. For example, each firewall in a company can be either configured individually to establish a set of rules defined by policy, or managed by a centralized policy administrator such as the Cisco Secure Policy Manager for firewalls [10]. The policy changes are expected to occur infrequently in traditional systems. More recently, Bhatt et al. [4] used self-managed and self-organized mechanisms for automating network management. Naccio of MIT [13] provided a high-level approach for safety policy expression and enforcement, which is implemented for enforcing policies on JavaVM classes. Schneider characterized a class of enforceable security policies [24] and there was an automata implementation [12] to enforce such policies.

The security working group [18] of the active networks

research community has been instrumental in publicizing and highlighting the importance of security in active networks. The security draft emphasizes the importance of incorporating security into the initial design stage of the active network architecture itself. As mentioned earlier, we believe that we can classify security related research in this field into three general categories. The first one deals with the more traditional notion of security. It includes authentication, access control, policies and enforcement. Some examples are protection of valuable information using encryption, providing data integrity using signatures. Public key infrastructure (PKI) and key distribution and management problems fall into this category. The security working group [18] has launched some important exploratory research in this direction.

The second category is related to security associated with the mobile nature of the environment. Protection of nodes from mobile code originating in foreign domains and protection of active packets or code from malicious hosts fall in this category. The PLANet effort [1] raises some of the issues associated with these protections. In addition they also provide a bootstrapping module that ensures that the system configures itself correctly at startup or reboot time. The protection from mobile code is provided by using a type-safe, resource limited, functional programming language with dynamic type verification. Mobile code can install protocols at nodes securely by using the extensibility features provided by the language.

Our research focuses on the third category: dynamic security. We believe that our work is complementary to the other research and attempts to enhance the flexibility and to improve usability of their techniques. By componentizing the security policy framework we provide an infrastructure to enforce any kind of expressible security policy. Using our infrastructure, applications can specify, implement, and enforce fine grained access control policies. These policies can be created, changed or revoked on the fly and enforced at run-time. Traditional mechanisms can be configured as components in our systems and their context can be instantiated and enforced on demand. The safety features provided by the bootstrapping and language features can be incorporated as an integral component of our framework.

However, there is more to dynamic security than simply dynamically deploying and enforcing security policies and mechanisms. We believe that we are barely touching the surface when it comes to exploring the potential applications and the limits of dynamic or active security. The combination of the active nature of the underlying architecture and the flexibility and dynamic nature of our framework has thrown open a new frontier for exploration and discovery. Examples like the tornado-watch and dynamic multicast demonstrate a fresh, alternative and functional approach to existing problems.

We are in the process of integrating our work within our active network working group and will demonstrate the flexibility and portability of our framework by incor-

porating it into CANEs [2]. We already have a implementation version that runs on the Abone [5]. In the future we plan to refine our techniques and define the protocols for interactions between heterogeneous security domains. In addition we also plan to explore applications of our framework to non-traditional ubiquitous computing environments and to integrate the applications into the active networking architecture.

Any node that uses our security has simply to add our reference monitor as an extension. The reference monitor will provide mechanisms for accessing advanced and composable services. The reference monitor can be used in an EE or Java environment, as well as in a NodeOS. We will also support the concept of domains and will provide domain-level policy conflict resolution and negotiation in the future.

In order to make the downloading of policy framework secure and to simplify the adding of extensions, we are in the process of developing a prototype of the Management EE. The Management EE will aid in managing the NodeOS, will initialize meta-level policies and will provide a framework for secure bootstrapping. We are also working on the design of a generic framework for key management.

7 Conclusions

In this paper, we describe a prototype security architecture that complements the basic active network architecture and augments its functionality. The flexibility and expressibility afforded by this implementation framework enables us to implement a multitude of diverse, innovative and exciting applications. These applications exploit the active networking paradigm without compromising the security of the infrastructure. In addition, our architecture lays the ground rules for seamless integration with parallel and ongoing efforts in the active networks community.

With our prototype implementation, we developed applications that demonstrate the benefits of our infrastructure. In particular, the Gnipper application demonstrates the creation of dynamically growing protection domains using vaccines. The multicast experiments showcase the use of our framework as an alternate approach to tackling the key-distribution and revocation problems associated with secure multicast applications.

In summary, we believe that our approach is a step in the direction of designing a comprehensive and flexible framework to integrate various security mechanisms and services into the active network architecture. It also provides a foundation for discussing issues related to co-existence, inter-operation and portability of these mechanisms. At the same time, our architecture imposes minimum overhead on the existing infrastructure and allows applications to specify and enforce customized security mechanisms conveniently.

References

- [1] The SwitchWare Project Homepage <http://www.cis.upenn.edu/~switchware/>.
- [2] CANEs Project Homepage <http://www.cc.gatech.edu/projects/canes>.
- [3] UCSC Multicast Research Homepage <http://www.cse.ucsc.edu/research/ccrg/>.
- [4] S. Bhatt, A. V. Konstantinou, S. R. Rajagopalan, and Yechiam Yemini. Managing security in dynamic networks. In *13th USENIX Systems Administration Conference (LISA'99)*.
- [5] Bob Braden and Livio Ricciulli. A plan for a scalable Abone - a modest proposal, January 1999.
- [6] K. Calvert et al. Architectural framework for active networks. AN Architecture Working Group, Draft, 1998.
- [7] Roy H. Campbell and M. Dennis Mickunas. Building dynamic interoperable security architecture for active networks. an accepted proposal to DARPA BAA9803, 1998. Also see the web site at <http://choices.cs.uiuc.edu/Security/seraphim/>.
- [8] Roy H. Campbell, M. Dennis Mickunas, Tin Qian, and Zhaoyu Liu. An agent-based architecture for supporting application aware security. In *the Workshop on Research Directions for the Next Generation Internet*, May 1997.
- [9] Roy H. Campbell and Tin Qian. Dynamic agent-based security architecture for mobile computers. In *the Second International Conference on Parallel and Distributed Computing and Networks*, Brisbane, Australia, December 1998.
- [10] Cisco Systems, San Jose, CA. *Cisco security manager tutorial, DOC-786905*, 1999. Available at <http://www.cisco.com/warp/public/cc/cisco/mkt/security/csm>.
- [11] D. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1982.
- [12] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [13] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 9-12, 1999.
- [14] Tim Fraser. An object-oriented framework for security policy representation. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1996.
- [15] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, January 1998.
- [16] Zhaoyu Liu, Prasad Naldurg, Seung Yi, Tin Qian, Roy H. Campbell, and M. Dennis Mickunas. An agent based architecture for supporting application level security. In *DARPA Information Survivability Conference and Exposition*, Hilton Head Island, SC, January 25-27, 2000.
- [17] Sandra Murphy. Active Networks Mailing List.
- [18] Sandra Murphy et al. Security architecture for active nets. AN Security Working Group, July 15, 1998.
- [19] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL '97)*, pages 106-119, January 1997.
- [20] L. Paterson et al. NodeOS interface specifications. AN NodeOS Working Group, Draft, 1999.
- [21] Vijay Raghavan. On the design and implementation of a security policy administration for a dynamic security system. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.
- [22] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security, LNCS 1419*. 1998.
- [23] R. S. Sandhu and E. J. Coyne. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
- [24] F. B. Schneider. Enforceable security policies. Technical Report 98-1664, Department of Computer Science, Cornell University, January 1998.
- [25] Van C. Van. A defense against address spoofing using active networks. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, May 1997.
- [26] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93*.
- [27] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *IEEE OPE-NARCH'98*, San Francisco, CA, April 1998.
- [28] Frank Yelin. Low-level security in Java. In *WWW4 Conference*, December 1995.