Proceedings of the 2007 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY, 20-22 June 2007

Arachne: Integrated Enterprise Security Management

Matthew Burnside, Angelos D. Keromytis

Abstract— **Security policies are a key component in protecting enterprise networks. There are many defensive options available to these policies, but current mechanically-enforced security policies are limited to traditional admission-based access control. There are defensive capabilities available that include logging, firewalls, honeypots, rollback/recovery, and intrusion detection systems, but policy enforcement is essentially limited to allow/deny semantics. Furthermore, access-control mechanisms operate independently on each service, which often leads to inconsistent or incorrect application of the intended system-wide policy. To begin to solve these problems, we propose a new system for defense-in-depth using global security policies. Under a global security policy, every policy decision is made with near-global knowledge, and re-evaluated as global knowledge changes, given an initial configuration provided by the administrator. Using a variety of *actuators*, we make the full array of defensive capabilities available to the global policy. We outline our proposal for enterprise-wide security policies, explore the design space, and discuss Arachne, our prototype implementation.**

I. INTRODUCTION

Modern security policies are inflexible. They are limited by a mismatch between notions of policy enforcement and the defensive capabilities in the network. There are a variety of defensive options available to a large network, including but not limited to logging, firewalls, honeypots, and intrusion detection systems – mechanisms that allow for escalating the response to an attack – but policy enforcement is essentially limited to allow/deny semantics.

Furthermore, traditional access-control mechanisms used in enterprise networks operate independently on each service. When a user issues a request to a network service, the service's access-control mechanism independently uses its security policy to make a decision on how to handle the request, then goes inactive. There may be information relevant to the decision elsewhere in the network, but the decision is made without consulting any other network entities, so the component may arrive at a *locally correct, but globally wrong* decision.

The access-control mechanism then goes inactive after its initial use, so no future action by the user within the context of the security policy can cause the decision to be revisited. This presents a problem with long-lived sessions (*e.g.*, SSH), because there is traditionally no action by the user that can cause the access-control mechanism to re-examine its decision.

Consider a local network with a web server and database connected to the Internet through a firewall, as shown in Figure 1. The security policy at the firewall (the firewall rule set) is defined by hand by the system administrator, as are the policies at the web server and database (`htaccess` and grant tables, respectively). Each of these policies is evaluated by the corresponding application, with no input from the other network entities, even though that input might sometimes be highly relevant, as we shall see shortly. Additionally, there are many defensive capabilities available to this network, including using the firewall to redirect misbehaving traffic to a honeypot, logging mildly suspicious traffic, and asking for re-authentication at the database. However, there is no coherent mechanism for specifying or coordinating these responses.

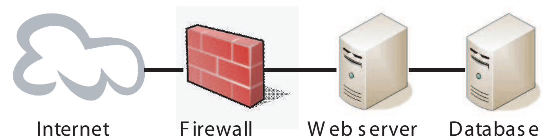

Fig. 1. Example network. A web server and database are connected to the Internet through a firewall.

The applications make policy decisions independently, so each application must make assumptions on the behavior of the others. In this case, the security policy at the web server must implicitly assume that all of its traffic has arrived through the firewall. An attacker can compromise the system by bypassing the firewall, through insider knowledge (gaining access to a local machine) or through an open wireless access point. He can then probe the web server's scripts for, say, SQL-injection vulnerabilities with impunity. The firewall verifies that each packet it sees conforms to its security policy, but the attacker's misbehavior goes unnoticed. There is no way for the web server to determine that an incoming request has been vetted by the firewall. We observe that although the network services work together in handling requests, there is no cooperation in determining the proper security context for authorizing these requests.

To resolve these conflicts, we propose a new scheme based on global security policies. Every policy decision is made with near-global knowledge, and then re-evaluated as the global knowledge changes. Furthermore, through the use of *actuators*, we make the complete array of defensive options available to the policy. In this paper, we will explore the design space of such a system, raising questions such as how to define and distribute policies, how to scale the system, and how to manage information flow, and then describe our prototype implementation called Arachne.

The remainder of this paper is organized as follows. In Sec-

CS Department, Columbia University, mb@cs.columbia.edu
CS Department, Columbia University, angelos@cs.columbia.edu

ISBN 9-9999-9999-9/99/ $20.00 ©2007 IEEE

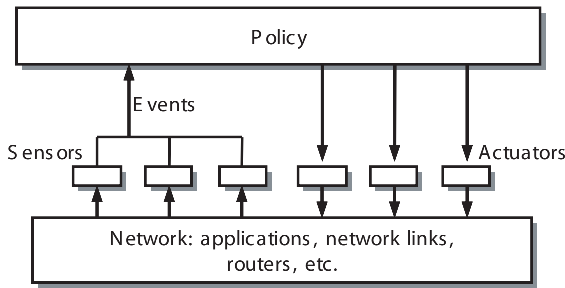

Fig. 2. System model. Sensors generate events, which are processed by the policy, which makes decisions and notifies actuators to modify network and application behavior.

tion II we describe the general model for systems to solve this problem. We discuss some of the design tradeoffs in the space in Section III. We give some details on our prototype, Arachne, in Sections IV and V. We discuss some related work in Section VI and conclude the paper in Section VII.

II. Architecture

Our goal is to create the tools necessary to make automated policy decisions in an enterprise network with near-global knowledge, and to re-evaluate those decisions in real time. For this system to be viable, it must improve the security of the network, be capable of scaling to enterprise network sizes, have a minimal impact on performance, and be straightforward for a system administrator to install and to define policies.

We model the system by dividing it into four major types of components (as shown in Figure 2):

Sensors Small programs scattered around the network that generate *events* corresponding to observed network and application behavior. Each sensor is customized for the particular application or network link it is observing. A sensor observing a web server is configured to parse web requests, server logs, *etc.* An intrusion detection system (IDS) is also considered a sensor.

Events An event is any action performed by an application that may be relevant to some policy decision. Examples of events include authenticating a user, initiating a network connection, and requesting a file. Events may be positive or negative; a firewall rejecting a connection is an event.

Policy A list of objectives, rules for behavior, requirements, and responses, whose goal is to ensure the security of the network. An example policy: *all incoming requests must connect on port 80 and may only request files named* `index.html`. The global policy may be distributed across multiple network nodes.

Actuators An actuator is a program which modifies application behavior after being triggered by a policy. An actuator might close a port on a firewall, turn on logging on a file server, redirect requests to a web page, or activate an intrusion detection system. The is the policy enforcement point.

A network, consisting of applications and network links, is observed by the *sensors*, each generating *events* in response to requests. Events are evaluated by the *policy*, which makes decisions and notifies the *actuators* to modify application behavior in response.

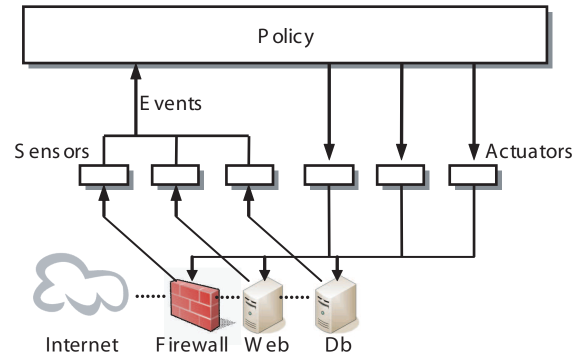

Fig. 3. The example network under our model.

Figure 3 shows the model as applied to the example from Figure 1. The firewall, web server, and database are observed by sensors which generate events which are evaluated by the policy; the policy then signals the actuators to modify each application's behavior in response to observed events. For example, consider the following policy:

1. Incoming requests are only allowed through the firewall if they are on port 80.
2. Requests are only processed by the web server if they pass through the firewall.
3. Requests are only processed by the database if they come from the web server, and are authenticated.
4. If a request fails authentication at the database, the firewall should drop the request, and not allow any further requests from that source.

An external user issues a request on port 80 for file `index.html`, which contains dynamic content. The request arrives at the firewall, generating an event. The request is allowed under Rule 1, so the request passes through the firewall, generating a "passed through the firewall" event. The request then arrives at the web server, generating another event. The policy determines that this request is allowed under Rule 2, because it can verify that the request has passed through the firewall (having seen the previously generated events). The file `index.html` has dynamic content, so the request must authenticate at the database, and then make the appropriate query. Each step generates a new event and is evaluated by the policy along the way. If authentication fails at the web server, the policy uses an actuator on the firewall to cut off the request and prevent it any further access to the network. Under a traditional security policy, the database might be able to lock out the request, but the attacker would be allowed to continue probing the rest of the network.

III. Design

To attain the goals of this system, we must address several design-related questions. Every design choice has trade-offs in performance, scalability, and so on. In this section, we discuss

some of the most significant of those questions and give some thoughts on answering them.

Which events are noteworthy? Any action taken by an application (or even a non-action) could be relevant to some policy decision. It requires intimate knowledge of the application to determine which actions are truly relevant. Thus, each sensor may require customization for the specific application it is observing.

How do we organize events? Event broadcasting and retrieval can be viewed as a publish/subscribe system [1]. However, with the potential for thousands of sensors, each generating thousands of events per minute, it is imperative that only noteworthy events are delivered to specific policy components. One possibility, which we will explore in Section IV is to aggregate events through groups of like interest, based on the idea of a session. We informally define a session as the collection of events generated by all services in the process of handling a specific request.

Beyond filtering on sessions, it may be desirable to filter on users, target services, or other types of events, thus creating channels for specific users, specific services, or specific service types (*e.g.*, any event related to any file server). Filters like this make it possible for the system to detect, for example, extremely slow port scans, or misbehavior by a particular user.

Where is the policy enacted? The policy engine may be a single centralized entity, or it may be distributed across multiple components. A centralized policy server may not scale, but with fully distributed policy decision-making and enforcement, it may be more difficult to administer and to define new policies. The latter case may be alleviated by centralizing the policy management while maintaining a distributed policy engine.

How does the policy interact with local policies? Most services have built-in policy mechanisms. Apache has `.htaccess`, MySQL has grant tables, and so on. This system might interact with the existing policy mechanisms by: overriding the existing mechanism, making all decisions locally, or finding a compromise where most decisions are made locally the the global mechanism steps in to make corrections.

How is the policy abstracted? That is, what abstraction do we use as the basis for enacting the policy? Modern access control models include access control lists (ACLs) and, more recently, role-based access control (RBAC). We believe neither is appropriate in this architecture. Consider a scenario where each incoming request is assigned a role from a pre-defined list. Each role has a set of services it is allowed to access. If the request misbehaves on the web server, we may wish to remove the web server from the set of services to which the request is allowed access, essentially changing the request's role to a new role with more limited access. Since a new role is required for every possible subset of services we may wish to allow for a request, the list of roles will quickly become unwieldy. Thus, some other mechanism is required.

As we will explore in more detail in Section IV, we propose a simple metric. For each session, we maintain a value estimating the trustworthiness of that session – a simple weighted count of misbehaviors the session has performed. If the session misbehaves, we deduct from the trustworthiness. More serious misbehavior leads to larger deductions.

IV. Arachne

We will now describe Arachne, our prototype implementation. Arachne is written in C and Python, and it consists of five major components: sensors, principals, an event database, a behavior-based policy engine, and actuators. We will discuss each in turn.

A. Sensors

Arachne sensors are small programs scattered around the network that generate *events* corresponding to observed network and application behavior. Each sensor is customized for the particular application or network link it is observing. As we will see in Section IV-C, a sensor must be configured with some knowledge of the local network topology.

B. Principals

The term *principal* has different meanings for different components in a network. At the network layer, it may be an IP address. For one application, a principal may be a $\langle username, password\rangle$ pair. For another application, a principal may be a public/private key pair. That is, the definition of principal is application-specific.

However, Arachne must be principal-agnostic, which is achieved through sensor customization. Sensors must already be customized on a per-application basis. When a sensor is customized, we teach it what a principal is for the application or network link it observes. A sensor for a firewall recognizes IP address:port pairs as principals, while a sensor for a database recognizes keys as principals.

C. Event database

Recall that one of the design questions for this system is how to manage the volume of events that will be generated by the sensors. Regardless of whether all events are stored in a central database or distributed, the volume requires some method for organization. In Arachne, we organize the events using two components: a primary MySQL database for storing events, and a custom database for linking events into *sessions*. The primary database allows for queries such as "list all events with a destination port of 22."

A session is a graph $G = (V, E)$ where the vertices V are a set of events and the edges E are causality links between them. In general, there is no method for identifying causality, but sensors distributed through the network can be used to build a causality graph that closely approximates the graph that would be built with total knowledge. We use sensors with local knowledge to generate special linkage events that indicate causal links between normal events.

For example, a TCP connection arrives at a firewall from some IP:port (so the firewall sensor generates an *incoming con-*

nection event) and then leaves to IP:otherport (the firewall sensor generates a *departing connection event*), and the firewall sensor *also* generates an event indicating that the previous two events are causally linked. These three events are processed by the session database to build a graph, the first two events are the initial two vertices and the third event represents the link between them.

There are two types of links: links across a single layer (as in the firewall example) and links that move up or down the network/application stack, and there is a sensor that covers each type. A linkage of the latter type is, for example, a TCP connection arriving at a web server that is causally linked with the application-layer behavior of responding to the request. Again, this does require that each sensor be customized for the specific application or network location it is observing.

When a request disconnects, a sensor generates an *unlink* event, so the session graph stored in the database always represents the current state of the session.

D. Policy engine

The policy engine is the core of Arachne's behavior-based access control mechanism. It maintains a score for each principal, modifies that score based on events linked to the principal through its sessions, and changes the network's responses to the principal by triggering actuators.

Each principal ID is retained in long-term storage in the event database, along with a behavioral score. Every principal starts with a score of 100, and that is also the maximum score. When events with non-zero penalty values are associated with the principal (by being linked through sessions) that value is deducted from the principal's score.

The penalty value for a given event is application-specific. For one application, a failed password is relatively unimportant, resulting in a small penalty, while for another it is significant should result in a large penalty. Sensors are already customized on a per-application basis, and it is a simple matter to store the application-specific penalty values for each class of event at the sensor as well. When a sensor generates an event, it attaches the penalty value before sending it to the event database. Similarly, actuators are locally customized to trigger for a specific range of a session's score.

Over the lifetime of a session, the scores for each principal linked to that session are modified based on the penalty values of the events generated. We treat each session as having the score of the lowest scoring principal linked to it, but we are researching more sophisticated heuristics.

E. Actuators

Each actuator is associated with a scoring range for which that actuator should trigger. When a session enters that scoring range, the actuator triggers. Consider an actuator which increases the log level for sessions in the scoring range of 0-95. If a session with a score of 100 fails a password auth, generating an event with a penalty value of 10, the session score will be reduced to 90, immediately triggering the increase-log-level actuator.

V. Discussion

In this section we discuss some subtleties in Arachne, including shared principals, how we use a forgiveness server to prevent the system from only becoming more restrictive, and finally, how we prevent the system from entering sensor/actuator cycles.

A. Shared principals

There are some cases where a principal may be shared. Consider a web server that multiplexes requests over a single connection to a database. All incoming requests to the database share the same key, and hence, the same principal. This opens the possibility for an adversary to mount a denial-of-service attack by misbehaving and driving down the score of that key. To resolve this issue, we require that shared principals be registered with the policy engine so that it can reduce the weight of the score of the shared principal (or ignore it all together) when calculating the behavioral score for a session.

Unfortunately, some principals may be shared outside our network so we have no direct knowledge of the sharing. For example, in NAT and DHCP situations, one IP may represent multiple individuals over time. A malicious user behind a NAT can deny service all the other users behind the NAT by misbehaving and driving down the score. This problem is handled by the forgiveness server, described in Section V-B.

A related situation is the case where multiple principals are controlled by a single hidden individual, such as the case of a botnet. In this case, we make use of IDS sensors which are able to detect, for example, DDoS attacks and multi-source port scans, and generate a "meta-principal" with linkages to the principals performing the attack. Thus, each small transgression taken by the individual principals is also deducted from the score of the meta-principal and the group as a whole is punished.

B. Forgiveness server

As described, our system can only get more restrictive over time. To allow for relaxation of a principal's score, we introduce the notion of a forgiveness server. Any time a principal is blocked by an actuator from taking some action, they are instructed (or redirected automatically, depending on context) to visit the forgiveness server. The forgiveness server is a web server that examines the score of the incoming principal and, based on its value, makes the principal perform whatever actions are necessary to bring the score back to 100. For example:

- To be forgiven a score of 95, the principal must reply to an automated email.
- To be forgiven a score of 75, the principal must submit his/her SSN.
- To be forgiven a score of 50, the principal must fill out a form explaining his/her behavior and submit his/her credit card number.

ISBN 9-9999-9999-9/99/ $20.00 ©2007 IEEE

• To be forgiven a score of 10, the principal must talk to a system administrator on the phone.

The specific actions required can be customized on a per-site, application, and principal basis.

C. Correctness

There are two fundamental ways the scoring system can fail: wrongly decrementing a score and wrongly forgiving a score.

The first is prevented on an ongoing basis by maintenance on the part of the system administrator, examining action reports and fine-tuning scores as necessary. The second is prevented by the forgiveness server which directly links the forgiveness values with the actions required to receive the forgiven value.

There is still a problem, however, and that is the initial assignment of the scores, penalty values, and actuator triggering ranges. The system administrator starts out with some initial values and then fine-tunes them to correctness. This is only feasible if the session-and-actuator-response package is isolated. That is, an actuator response can only affect a single session; if it affects multiple sessions, then it is possible to cause a cascade where a session generates an event which triggers an actuator which generates an event which triggers another actuator, *etc*.

To prevent these cascades, we disable events generated as a result of an actuator and allow actuators only to affect a single session. This reduces the broadness of our system (by limiting the abilities of the actuators) but it prevents catastrophic errors. The session-and-actuator-response is isolated so that a misconfiguration or other error in a single session will not affect other sessions. As long as the configuration is correct for a given session, that session will continue to be handled, even if some other session is configured incorrectly.

The key is that misconfigurations and errors are always possible, but by isolating the effects of those mistakes, the system itself continues to operate.

VI. Related work

In existing services, the access-control mechanism operates as a *gatekeeper*. When a principal makes a request, the access-control mechanism consults a security policy, makes a decision, and goes inactive. The access-control mechanism (and hence, the security policy) is not consulted again, regardless of any future actions taken by that principal. This style of access control was first described by Lampson [2], [3], and refined by Graham and Denning [4]. Their work provided the basis for specifying security policies in the form of the access control matrix, from which the widely used access control list (ACL) is derived. ACLs consist of a list of tuples:

$$< \text{subject}, \text{object}, \text{access rights} >$$

that define the security policy for the system – which subjects have which access rights on which objects.

ACLs do not scale well in all cases, so in large-scale networks they are often replaced by role-based access control (RBAC) [5], [6], [7]. RBAC is now the predominant model for advanced access control. Each principal is assigned one or more roles, and each role has an associated list of privileges that are permitted members in the role. Both ACLs and RBAC are useful tools, and may play some part in the system proposed in this paper, but alone they do not solve the problems we have described. They both fall into the gatekeeper category, where the security policy is consulted only once, and there is no mechanism for escalating response.

Recent work [8], [9] on policy-based management [10] and the NSA's RAdAC (Risk Adaptable Access Control) [11] model have demonstrated that synchronous repeated policy evaluation is feasible and desirable. Arachne takes it one step further and allows for *asynchronous* reevaluation.

Most prior work in the policy field can be divided into three major categories: policy specification [12], [13], resolving policy conflicts [14], [15], and distributed enforcement [16], [17].

In their work in the field of trust management, Blaze, *et al*., [18], [19], [20] built PolicyMaker, a tool that takes a unified approach to describing policies and trust relationships in enterprise-scale networks by defining policies based on credentials. It is based on a policy engine that identifies whether some request r with credentials c complies with policy p. In PolicyMaker, policies are defined by programs evaluated at runtime. SPKI [21], [22], [23] is a similar mechanism that uses a formal language for expressing policies. In both cases, the focus is on trust management rather than policy correctness.

Bonatti, *et al*., [24] propose an algebra for composing heterogeneous security policies. This is useful in networks with multiple policies defined in multiple languages (*i.e.*, most networks today). However, this system requires that *all* policies and supporting information and credentials be available at a single decision point.

When there are multiple policies or multiple users defining policy there is always the possibility of conflict [15]. The problem is exacerbated in large-scale networks.

The STRONGMAN trust management system [25] focuses on the problem of scaling the enforcement of security policies and resolving policy conflicts. In STRONGMAN, high-level, abstract security policies are automatically translated into smaller components for each service in the network. However, in STRONGMAN, if some base assumption in the high-level policy is violated, there is no way for the individual components to detect or recover.

Ponder [26] and SPL [27] are policy languages that avoid policy conflicts by depending on a unique namespace for all resources in the system. A rule operating on an object is always guaranteed to be operating on the same object, regardless of where in the system the rule is interpreted.

Firewalls [28], [29] are one of the most common and most well-known mechanisms for policy enforcement. However, nearly all firewall research has focused on isolated firewall nodes and the specifics of the enforcement mechanisms, rather than policy coordination.

The Oasis architecture [30] takes a wider view and uses a role-

based system where principals are issued names by services. A principal can only use a new service on the condition that it has already been issued a name from a specific other service. Oasis recognizes the need to coordinate the dependencies between services, but since credentials are limited to verifying membership in a group or role, it is necessary to tie policies closely to the groups to which they apply.

The Firmato system [31] is a firewall management toolkit. It provides a portable, unified policy language, independent of the firewall specifics. Firewall configuration files are generated automatically from the unified global policy. Firmato is limited to packet filtering, and the complete policy must be available at the policy-enforcement point so Firmato may not scale well in large networks.

RADIUS [32] and its successor DIAMETER [33] are authentication, authorization and accounting protocols. They require communication with a policy server to make policy-based decisions. These protocols are typically used for user administration in roaming and dial-up situations.

Park and Sandhu [34] use the notion of usage control (UCON) to integrate many mechanisms, including authorizations, obligations, continuity and mutability. *Continuity* refers to the concept of ongoing controls for long-lived sessions or asynchronous revocation. UCON uses the continuity concept to allow for re-evaluating decisions when an attribute change occurs in an entity. Our work extends UCON by re-evaluating decisions when *any* relevant event occurs.

Hale, *et al.*, [35] propose a ticket-based authorization model to manage distributed policies. In this architecture, each network is managed by a controlling mediator communicating with a central policy repository. The mediator serves as a middleware layer, facilitating communication between disparate objects and principals.

Vandenwauver, *et al.*, [36] use a combination of mail, web and script-based attacks to show that any intranet protected solely by firewalls and intrusion detections systems cannot be made completely secure.

VII. Conclusion

We have argued that current mechanically-enforced security policies are incapable of addressing the complexities of modern enterprise networks. They are unable to take full advantage of the defensive capabilities available to them, while the independent nature of access-control decision-making leaves the door open to adversaries. We propose a system to address these problems through global security policies, where every policy decision is made with near-global knowledge, and re-evaluated as global knowledge changes. Through the use of actuators, we make the full array of defensive capabilities available to the global policy. We describe Arachne, a prototype implementation of this system. The goal is a coherent, enterprise-wide response to any network threat.

Acknowledgments

This research was sponsored by the NSF through grants CNS-06-27473 and CNS-04-26623. We authorize the U.S. Government to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the U.S Government.

References

[1] D. Powell (Guest Ed.), "Group communication," *Communications ACM*, vol. 39, pp. 50–97, Apr. 1996.

[2] B. Lampson, "Protection," in *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pp. 473–443, March 1971.

[3] B. Lampson, "Protection," *Operating Systems Review*, vol. 8, pp. 18–24, January 1974.

[4] G. S. Graham and P. J. Denning, "Protection: Principles and Practices," in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 417–429, 1972.

[5] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[6] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, pp. 224–274, August 2001.

[7] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role Based Access Control*. Artech House, 2003.

[8] R. Choudhary, "A Policy Based Architecture for NSA RAdAC Model," in *Proceedings of 6th IEEE Workshop on Information Assurance and Security*, (United States Military Academy, West Point, NY), June 2005.

[9] R. Choudhary, "Compound Identity Measure: A New Concept in Information Assurance," in *Proceedings of 7th IEEE Workshop on Information Assurance and Security*, (United States Military Academy, West Point, NY), June 2006.

[10] J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for Policy-Based Management," Request for Comments (Proposed Standard) 3198, Internet Engineering Task Force, Nov. 2001.

[11] R. W. McGraw, "Securing Content in the Department of Defense's Global Information Grid," in *Secure Knowledge Management Workshop*, (State University of New York, Buffalo, NY), September 2004.

[12] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The KeyNote Trust Management System Version 2." Internet RFC 2704, September 1999.

[13] M. Damianou, *A Policy Framework for Management of Distributed Systems*. PhD thesis, 2002.

[14] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A logical language for expressing authorizations," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 31–42, May 1997.

[15] L. Cholvy and F. Cuppens, "Analyzing consistency of security policies," in *RSP: 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

[16] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari, "Certificate-based access control for widely distributed resources," in *Proceedings of the USENIX Security Symposium*, pp. 215–228, August 1999.

[17] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith, "Managing access control in large scale heterogeneous networks," in *Proceedings of the NATO NC3A Symposium on Interoperable Networks for Secure Communications (INSC)*, November 2003.

[18] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," in *Proc. of the 17th Symposium on Security and Privacy*, pp. 164–173, IEEE Computer Society Press, Los Alamitos, 1996.

[19] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance Checking in the PolicyMaker Trust-Management System," in *Proc. of the Financial Cryptography '98, Lecture Notes = in Computer Science, vol. 1465*, pp. 254–274, Springer, Berlin, 1998.

[20] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The role of trust management in distributed systems security," in *Secure Internet Programming*, pp. 185–210.

[21] C. Ellison, "SPKI requirements," Request for Comments 2692, Internet Engineering Task Force, Sept. 1999.

[22] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI certificate theory," Request for Comments 2693, Internet Engineering Task Force, Sept. 1999.

[23] C. M. Ellison, "SDSI/SPKI BNF." Private Email, July 1997.

[24] P. Bonatti, S. D. C. di Vimercati, and P. Samarati, "A Modular Approach to Composing Access Policies," in *Proceedings of Computer and Communications Security (CCS) 2000*, pp. 164–173, November 2000.

[25] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith, "The STRONGMAN Architecture," in *Proceedings of the* 3rd *DARPA Information Survivability Conference and Exposition (DISCEX III)*, pp. 178–188, April 2003.

[26] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," *Lecture Notes in Computer Science*, vol. 1995, pp. 18–38, 2001.

[27] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, "Security policy consistency," 2000.

[28] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[29] J. Mogul, R. Rashid, and M. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 39–51, November 1987.

[30] R. Hayton, J. Bacon, and K. Moody, "Access Control in an Open Distributed Environment," in *IEEE Symposium on Security and Privacy*, May 1998.

[31] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: a novel firewall management toolkit," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 17–31, May 1999.

[32] C. Rigney, A. Rubens, W. Simpson, and S. Willens, "Remote Authentication Dial In User Service (RADIUS)," Request for Comments (Proposed Standard) 2138, Internet Engineering Task Force, Apr. 1997.

[33] P. Calhoun, A. Rubens, H. Akhtar, and E. Guttman, "DIAMETER Base Protocol," Internet Draft, Internet Engineering Task Force, Dec. 1999. Work in progress.

[34] J. Park and R. Sandhu, "The UCON_{ABC} usage control model," *ACM Transactions on Information and System Security*, vol. 7, pp. 128–174, Feb. 2004.

[35] J. Hale, P. Galiasso, M. Papa, and S. Shenoi, "Security Policy Coordination for Heterogeneous Information Systems," in *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC)*, December 1999.

[36] M. Vandenwauver, J. Claessens, W. Moreau, C. Vaduva, and R. Maier, "Why enterprises need more than firewalls and intrusion detection systems," in *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99), 16-18 June 1999, Stanford, CA, USA*, pp. p.152–7, Los Alamitos, CA, USA : IEEE Comput. Soc, 1999.

ISBN 9-9999-9999-9/99/ $20.00 ©2007 IEEE