Asynchronous Policy Evaluation and Enforcement

Matthew Burnside
Computer Science Department
Columbia University
New York, NY
mb@cs.columbia.edu

Angelos D. Keromytis
Computer Science Department
Columbia University
New York, NY
angelos@cs.columbia.edu

ABSTRACT

Evaluating and enforcing policies in large-scale networks is one of the most challenging and significant problems facing the network security community today. Current solutions are limited by an out-of-date allow/deny paradigm, and policies are evaluated synchronously and independently at each service. This makes it difficult to detect or defend against multi-stage attacks, or attacks which begin as innocent requests and then later exhibit malicious behavior in the same context. In this paper we describe Arachne, a prototype for asynchronous policy evaluation. We evaluate the system by testing it against pre-recorded traffic containing known and unknown attacks and show that it is capable of processing events at more than 10x the required rate for a deployed, heavily-used network.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*access controls, information flow controls*

General Terms

Management, Security

Keywords

Policy, Access control, Security, Scalabilty

1. INTRODUCTION

One of the most significant problems facing the network security community is that of evaluating and enforcing policies in large-scale networks. As networks increase in complexity, so must policy definition, evaluation, and enforcement. While substantial work has been done on policy definition in large-scale networks [14, 3, 10], the jobs of policy evaluation and enforcement are still mired in the independent allow/deny semantics first proposed by Lampson [15, 16].

A traditional allow/deny evaluation and enforcement mechanism performs a one-time evaluation of each request against the policy and enforces an allow or deny decision. Once made, that decision is not revisited for the duration of the request. This mechanism fails when, for an allowed request, subsequent actions in the context of that request make clear that the evaluation engine should be re-evaluating its decision.

Consider a web request containing an SQL injection attack arriving at the firewall of a large, well-defended network (for some definition of well-defended). The request is evaluated in the context of the firewall policy (the firewall rules) and allowed to continue. The attack is then (hypothetically) detected and rejected at the web server. However, this information is not transmitted to the firewall, and thus its initial decision stands; the attacker may continue to use the access through the firewall for subsequent attacks on other hosts within the network. We call this *synchronous* policy evaluation.

In synchronous policy evaluation, a request is evaluated against the policy exactly once, at the time of the request's arrival, and that evaluation is never revisited. Synchronous policy evaluation is flawed in two facets. Since a evaluation engine P_1 never revisits its policy decisions, so an attacker can perform innocent actions to gain access and subsequently introduce an attack. If that attack is against a downstream service governed by P_2, there is no back-channel to update P_1 on the information gleaned by P_2. Therefore, each policy engine makes its evaluation based solely on local information.

Meanwhile, limiting policy enforcement to allow/deny ignores the diverse assortment of responses available to the modern system administrator. These responses include raising the log levels for certain requests, or redirecting requests to honeypots or instrumented networks.

This paper describes Arachne, a system for asynchronous policy evaluation. In asynchronous policy evaluation, policy may be re-evaluated against a request at any node at any time. Arachne provides a back-channel – a database – for collecting and linking policy-related events and then asynchronously evaluates a global policy against that database. Arachne also provides a plug-in architecture for policy enforcement modules, allowing a system administrator to define arbitrary responses. In this paper, we describe the Arachne prototype and evaluate it using pre-recorded traffic from a heavily-used network, containing both known and unknown attacks. We show that Arachne is capable of handling events at a rate up to 10x the rate generated by this network.

The initial concepts of the Arachne architecture were first sketched out in [4]. This new paper reflects the substantial changes to the core architecture and lessons learned during the development pro-

CSAW'08, October 31, 2008, Fairfax, Virginia, USA.

cess, including radical changes to the event correlation and policy evaluation mechanisms. This paper also includes details on and evaluation of our prototype implementation. The work in [5] implements a proper subset of Arachne, allowing back-channel communication only between immediate neighbors.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we give an overview of the Arachne architecture. In Section 4 we discuss our evaluation of the system, and we conclude in Section 5.

2. RELATED WORK

In traditional policy enforcement mechanisms, as proposed by Lampson [15, 16] and refined by Graham and Denning [12], the access-control engine operates as a *gatekeeper*. An incoming request is evaluated against the policy and the request is either denied or allowed to continue. The decision is never revisited.

This philosophy is most clearly embodied in firewalls [7, 18]. They are one of the most common and most well-known mechanisms for policy enforcement. The Firmato system [2] is a firewall management toolkit for large-scale firewall deployments. It provides a portable, unified policy language, independent of firewall specifics. Firmato is limited to packet filtering for enforcing policy, and policy is evaluated synchronously and not revisited.

Other large-scale policy-enforcement engines include RADIUS [22] and its successor DIAMETER [6]. These are authentication, authorization and accounting protocols. They require communication with a policy server to make policy-based decisions and are typically used for user administration in roaming and dial-up situations. Both enforce policy synchronously.

Recent work [8, 9] on policy-based management [23] and the NSA's RAdAC (Risk Adaptable Access Control) [17] model have demonstrated that evaluation of dynamic policies is feasible and desirable. This is a powerful mechanism, closely related to Arachne. However, though polices themselves are dynamic, they are still evaluated synchronously.

The usage control philosophy (UCON) [20] integrates mechanisms including authorizations, obligations, and mutability. UCON is based on *continuity*, which refers to the concept of ongoing controls for long-lived sessions or asynchronous revocation. It uses the continuity concept to allow for re-evaluating decisions when an attribute change occurs in an entity. Arachne can be viewed as an extension of UCON by re-evaluating decisions when *any* relevant event occurs.

Arachne builds graphs that are, conceptually, similar to attack graphs as proposed by Sheyner, *et al*. [24]. However, attack graphs statically represent attack pathways into a system, while the graphs built by Arachne are dynamic and represent all active requests – not just the attacks.

There is a substantial body of work on correlating alerts in intrusion detection systems [19]. As we shall see, Arachne applies techniques from this arena on the small scale, in order to correlate events, rather than alerts.

3. ARCHITECTURE

Figure 1 shows an overview of the Arachne architecture. The Arachne system is designed to protect large-scale service-oriented architecture (SOA) networks against multi-hop or multi-stage attacks by using asynchronous policy evaluation. It performs this function by using *sensors* to build and maintain an internal model describing the current state of each request made on the protected network. We call the state of each request a session, and the session models are stored in the *session database*. To determine the access level permitted to each session, each is evaluated by the *policy engine* against a *policy*. The nature of Arachne allows for more sophisticated responses than the allow/deny semantics used by firewalls. The Arachne response architecture is modular, and each module is called an *actuator*. Each rule in the policy has an associated actuator which is activated when the rule triggers. In the remainder of this section, we describe the details of those components in the Arachne architecture.

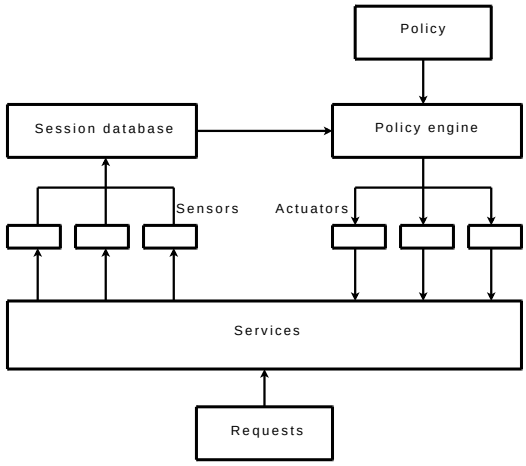

Figure 1: Arachne architecture.

3.1 Session database

The core of the Arachne system is the session database. It maintains the session models for each incoming request. Each session S is represented as a graph $S = (V, E)$. The vertices $v \in V$ represent session-related events generated by sensors. Each vertex has a unique 32-bit identifier and a set of attribute key-value pairs describing the event. Each edge $(u, v) \in E$ represents a causal link between events u and v. That is, event u *caused* event v.

The causality information is generated at the sensor level and it is associated with the corresponding events before delivery to the sensor database. When an event is generated by a sensor and sent to the sensor database, it contains its successor information (a list of the 32-bit identifiers $p_1 ... p_n$). The session database inserts the event into the appropriate location in the session graphs.

Intuitively, a session is a model of the path taken by a request in the network. It is directed and acyclic.[1] It is an embodiment of the the answer to the question, "Where did this request go when it entered the network, and what did it do?" We use the session database as a window into the network, to facilitate policy enforcement.

3.2 Sensors

[1]The edges in the graph represent causality so creating a cycle would require time travel.

```
{ 'app': "httpd",
  'src': 10.23.1.2:234,
  'dst': 10.1.2.3:80,
  'method': "GET",
  'path': "/index.html",
}
```

(a)

```
{ 'app': "pf",
  'src': 10.23.1.2:234,
  'dst': 10.1.2.3:80,
  'method': "RDR",
  'seqnum': 3851915398,
}
```

(b)

Figure 2: (a) An event representing an HTTP GET on the file `/index.html`. (b) An event representing a firewall redirect of an incoming TCP session.

Sensors are the eyes of the Arachne system. They detect session-related events and report them to the session database. In most cases, a sensor is targeted at and associated with a particular instance of an application. There is a sensor for each web server, a sensor for each firewall, *etc.* Every sensor has been pre-configured to observe and report on the policy-related activity of its specific application. That is, a sensor reports each local policy decision. There is no proscribed means by which a sensor must operate, but the typical case is one of processing the application log file in real time.

For example, the sensor for an Apache web server parses the log file `access_log`. Each entry in the log file represents a step taken by the web server in response to a request. For each entry, the Apache sensor generates an event to report to the session database. For example, an event representing an HTTP GET request takes the form shown in Figure 2a. The fields comprising the event are arbitrary and application dependent. An event representing a redirected TCP connection on an OpenBSD PF firewall takes the form shown in Figure 2b.

Sensors may also be more general. One of the strengths of the Arachne system is that any event-reporting system may be incorporated as a sensor. The alerts generated by IDS like Snort are easily translated into events understood by Arachne, and included in the session database. Another class of sensors used by Arachne are link sensors. These sensors are on the wire; they consist of network taps at both ends of every link in the protected network. Each link sensor maintains a TCP connection with its sibling and uses that link to update the sibling regarding the creation and destruction of TCP sessions, and to transmit linkage information as described below.

The Arachne infrastructure depends on the sanctity of the sensor mechanisms. As such, sensors must be well protected. They may be virtual-machine based [11] or remote, through *e.g.*, remote logging. Regardless, if a sensor is compromised an adversary may be able to use that position to generate events such that his session appears innocent. This attack may be alleviated through attestation, similar to that of SBGP [13] – the events generated by a compromised sensor will not correlate with those generated by neighboring uncompromised sensors. However, we do not address this issue further in this paper.

3.3 Event correlation

Sensors are grouped into sensor groups, and events are multiplexed through a group handler and processed there by a correlation engine before being reported to the session database. Typically, a sensor group consists of the sensors associated with all services on a single host. The correlation engine generates linkages between events by detecting correlations between their attributes. For example, if the TCP sequence number of an incoming connection on an internal web server is reported by a link sensor (ID `dc1903c8`) and corresponds to the TCP sequence number reported by the firewall redirect (ID `b24838c4`), the correlation engine infers that events `dc1903c8` and `b24838c4` are causally related. It assigns a new attribute to the firewall redirect, indicating that the link sensor event is a successor:

`{ 'succ': [dc1903c8] }`.

The correlation engine will attempt correlations between events based on arbitrary fields, including source or destination IP:port, TCP sequence number, timing, *etc*. However, each sensor facilitates this operation by providing hints to the correlation engine on which fields are appropriate. For example, when generating events relating to TCP sessions, the firewall sensor indicates that correlation on the TCP sequence number should be attempted first.

When, in the process of handling a request, a service forwards that request on to a downstream service that is situated on another host, the creation of the TCP session for that forward is detected by the link sensor on the originating service, using its network tap. This link sensor forwards to its sibling link sensor on the target service the successor information for the incoming session. In this fashion, successor information is passed from sensor group to sensor group as a request traverses a network.

3.4 Policy

A policy is a wildcard-based set of pre-defined session graphs. Each entry in the set is associated with an actuator to be triggered if the rule matches.

As sessions for incoming requests are built, they are evaluated against the set of rules in the policy. If an incoming request graph is determined to be isomorphic with one of the policy graphs, the policy engine performs a wildcard match of the attributes of the corresponding vertices. If the wildcard match succeeds, the rule's associated actuator is triggered.

Figure 3 demonstrates a set of requests (Figure 3.1-3.3) being made against a policy set (Figure 3.a-3.d). The requests and ruleset are simplified for this example and, of course, would be substantially more complex in a real-world deployment. Additionally, each vertex has an associated set of attributes (as in Figure 2) which are not shown. The policy engine in this example has been configured to use the ruleset as a simple whitelist: each rule shown is associated with a null actuator and there is a fifth rule (not shown) which is a catch-all rule. It is associated with an actuator configured to reject the incoming request.

Rule 3.a describes the allowed pathway for requests on a particular web server. In this case, the rule allows requests which pass through the firewall `fw` to the web server `web` and on to the database `db` and

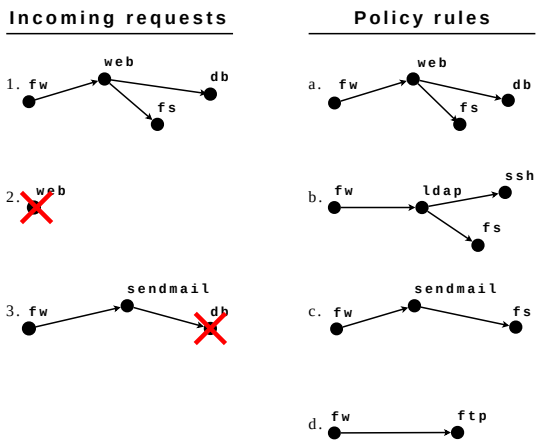

Figure 3: Example requests and a policy set. Each incoming request is evaluated against the policy set.

file server `fs`. The attributes associated with each vertex may further restrict allowed requests to, *e.g.*, particular file sets on the web server or tables on the database. Rules 3.b-3.d describe, respectively, pathways for logging on to an SSH server using LDAP authentication, sending mail, and obtaining files from an FTP server.

As each incoming request is processed by the network, it is modeled by the session database. Those models are shown in 3.1-3.3 and any request which deviates from an allowed rule is rejected. Request 3.1 is allowed, as it matches rule 3.a. Request 3.2 is rejected immediately as it does not match any rule. Request 3.3 matches rule 3.c, initially, and is allowed access. When it deviates from 3.c, however, by attempting to connect to the database, it is rejected.

The ability of the Arachne system to defend against attacks represented by 3.2 and 3.3 is one of the key features of asynchronous policy evaluation. Figure 3.2 represents a scenario where, for example, an adversary has obtained access to the internal network through an unauthorized wireless access point and is probing the web server. In a traditional network, there is no means for the access control mechanism on the web server to consult with the firewall to determine whether the incoming request has been vetted. Arachne, on the other hand, detects and rejects the request immediately.

Figure 3.3 represents a scenario where an attacker has delivered an exploit against the mail daemon and is using it as a platform as a launching point to probe the database. As soon as the attacker deviates from the rule, Arachne triggers the reject actuator. Note that, even though the misbehavior happened only at the database, since the session database stores the complete details of this connection, the actuator assigned to reject the connection is able to reject it simultaneously from the database, mail daemon, and the firewall.

3.5 Actuators

Through the actuator infrastructure, Arachne allows a system administrator to define arbitrary responses when policy rules are triggered, simply by implementing an actuator to embody the desired response. Additionally, by providing the session data from the matched session to the actuator, the response may be carefully targeted. Example actuators range from cutting off the session at the firewall to raising the log level across all services involved in the session to redirecting the session to a honeypot.

4. DISCUSSION

We have designed and built a prototype Arachne system and evaluated that prototype on a number of testbed networks. The prototype consists of approximately 10,000 lines of Python code, including sensors for 10-15 of the most common open source server applications, and 10-15 actuators.

Deployment of the Arachne system requires an installation of the session database and policy engine, deployment of the sensors and actuators, and creation of the policy set. Note that each sensor must be customized for its target application, so the creation cost is relatively high. However, that cost is amortized over all future uses. Only one user has to write the sensor for a given application and all future users may utilize and benefit from it.

Once the session database, sensors, and actuators are deployed, the system administrator may begin to create and install policies. Policies are created by hand using a graphical editor, called the Policy Editor, also written in Python using the Tkinter GUI package. The Policy Editor allows the system administrator to draw the session graphs for each policy and define the accepted attribute values for each vertex in the graphs. The Arachne prototype provides a null actuator which may be associated with session graphs during the testing process. In this fashion, the system administrator may fine-tune a session graph before deployment.

We evaluate the Arachne infrastructure by deploying it on testbeds which replicate networks with known attack histories, and replay traffic on the testbed. We have been involved in the forensic analysis of a number of sophisticated multi-stage attacks against various portions of our department network. As such, we have recordings of traffic and detailed logs during the time period of those attacks. By replicating the attacked network segments, instrumenting them with the Arachne prototype, and replaying the traffic against them, we are able to evaluate the Arachne prototype.

4.1 Sample testbed

In one instance of a real-world attack, an attacker used a well-known exploit in a popular open source web-based photo gallery to obtain a shell on a departmental web server. The attacker then used a local privilege escalation exploit to obtain root on that same machine. The photo gallery in question was Coppermine Photo Gallery [1] 1.4.14, which is vulnerable to an exploit which allows remote command execution [25]. The web server was running Apache 2.0.55, PHP 5.05, and the Coppermine Photo Gallery backend was MySQL 4.0.17. All applications were hosted on Ubuntu 7.04 running Linux kernel 2.6.20. That version of the Linux kernel is vulnerable to the so-called vmsplice exploit [21], which is a privilege escalation exploit. Given an account with user-level permissions on that host, the vmsplice exploit provides a shell with root privileges.

We recreated this environment in testbed form and deployed Arachne on the firewall, web server and photo gallery applications. We then replayed traffic derived from the logs from the two-week time period surrounding the original attack. A simple whitelist ruleset describing the path of valid requests on the system detected and prevent the known attack, along with a number of failed attempts from other attackers, both before and after the compromise.

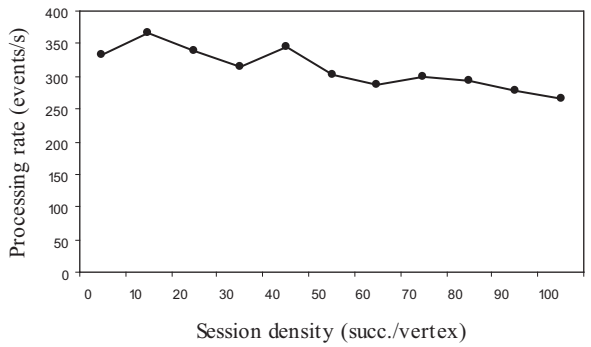

Figure 4: Processing rate at the session database and policy engine as a function of session density.

4.2 Evaluation

The scalability of the Arachne architecture is limited by the volume handled at the session database. Furthermore, growth of the session database and the density of the individual session graphs impacts performance of the policy evaluation engine. Figure 4 is a micro-benchmark demonstrating the impact on the session database and policy engine as the degree of connectivity of each vertex in the graph increases. This figure shows the processing rate (events/s) for 1000 events, as the degree of connectivity of each vertex increases. Each session is evaluated against a 30-rule policy. The benchmark shows that Arachne is capable of handling approximately 300 events/s for reasonable density values. The sensors in the sample testbed described above collectively generated an average of 8.3 events/sec, with peak rates up to 30.4 events/sec. Thus, Arachne is capable of processing events at 10x the peak rate.

The potential impact from high session count is lessened by leveraging the sensor infrastructure. Just as the sensors report each step in the construction of an incoming session, they also report the closing of each segment of the session. As each step in the request completes (the database request returns, the web request finishes, *etc*.) the sensors report that information to the sensor database. Each session, as it closes completely, is removed from the database.

5. CONCLUSION

The current paradigm of synchronous policy evaluation is flawed. A policy evaluation engine never revisits its decisions, and downstream events are not propagated back to upstream decision makers. Each policy engine makes its evaluation based only on local information, leading to security flaws. Additionally, policy enforcement is traditionally limited to allow/deny, ignoring the assortment of responses available to the modern system administrator. In this paper we have described the motivating philosophy behind Arachne. It provides asynchronous policy evaluation, a rich language for policy definitions, and a flexible policy enforcement engine. We have shown that the Arachne engine can handle more than 10x the events generated by a heavily-used real-world network.

Acknowledgements

This research was sponsored by the NSF through grants CNS-07-14647 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.

6. REFERENCES

[1] Coppermine Photo Gallery. http://www.coppermine-gallery.net.

[2] 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*, pages 17–31, May 1999.

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

[4] Matthew Burnside and Angelos Keromytis. Arachne: Integrated enterprise security management. In *8th Annual IEEE SMC Information Assurance Workshop*, pages 214–220, 2007.

[5] Matthew Burnside and Angelos Keromytis. Path-based access control for enterprise networks. In *11th Information Security Conference (ISC2008)*, September 2008. To appear.

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

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

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

[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, New York, NY, USA, 2002. ACM.

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

[13] Stephen Kent, Charles Lynn, and Kareo Seo. Secure border gateway protocol (secure-bgp). 18(4):582–592, April 2000.

[14] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith. The STRONGMAN Architecture. In *Proceedings of the 3^{rd} DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 178–188, April 2003.

[15] B.W. Lampson. Protection. In *Proceedings of the 5^{th} Princeton Symposium on Information Sciences and Systems*, pages 473–443, March 1971.

[16] B.W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974.

[17] Robert 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.

[18] 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*, pages 39–51, November 1987.

[19] Peng Ning, Yun Cui, and Douglas S. Reeves. Analyzing intensive intrusion alerts via correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October 2002.

[20] Jaehong Park and Ravi Sandhu. The UCON$_{ABC}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, February 2004.

[21] qaaz. Linux vmsplice Local Root Exploit. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=464953, February 2008.

[22] 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, April 1997.

[23] 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, November 2001.

[24] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[25] Janek Vind. Remote Shell Command Execution in Coppermine 1.4.14. http://www.waraxe.us/advisory-65.html, January 2008.