Integrated Access Control and Intrusion Detection for Web Servers*

Tatyana Ryutov, Clifford Neuman, Dongho Kim and Li Zhou
Information Sciences Institute
University of Southern California
{tryutov, bcn, dongho, zhou}@isi.edu

Abstract

Current intrusion detection systems work in isolation from access control for the application the systems aim to protect. The lack of coordination and inter-operation between these components prevents detecting and responding to ongoing attacks in real time, before they cause damage. To address this, we apply dynamic authorization techniques to support fine-grained access control and application level intrusion detection and response capabilities. This paper describes our experience with integration of the Generic Authorization and Access Control API (GAA-API) to provide dynamic intrusion detection and response for the Apache Web Server. The GAA-API is a generic interface which may be used to enable such dynamic authorization and intrusion response capabilities for many applications.

1 Introduction and Motivation

Web servers continue to be attractive targets for attackers seeking to steal or destroy data, deny user access or embarrass organizations by changing web site contents. Furthermore, because web servers must be publicly available around the clock, the server is an easy target for outside intruders. In order to penetrate their targets, attackers may exploit well-known service vulnerabilities. A web server can be subverted through vulnerable CGI scripts, which may be exploited by meta characters or buffer overflow attacks. These vulnerabilities may be related to the default installation of the server or may be introduced by careless writing of custom scripts.

Web servers are also popular targets for Denial of Service (DoS) attacks. An attacker sends a stream of connection requests to a server in an attempt to crash or slow down the service. Launching a DoS attack against a web server can be accomplished in many ways, including ill-formed HTTP requests (e.g., a large number of HTTP headers). As the server tries to process such requests it slows down and becomes unable to process other requests. In addition, web servers exhibit susceptibility to password guessing attacks.

To address these risks, web servers require increased security protection. Effective system security starts with security policies that are supported by an access control mechanism. Access control policy to be enforced should depend on the current state of the system, e.g., time of day, system load or system threat level. More restrictive organizational policies may be enforced after hours, when the system is busy or if suspicious activity has been detected. Unfortunately, many web servers (e.g., Apache and IIS) support only limited identity- and host-based policies that deny/allow access to protected resources. The policies are checked only when an access request is received to determine whether the request should be permitted or forbidden. These policies do not support observing and reporting suspicious activity (e.g., embedding hexadecimal characters in a query) and modifying system protection as a result.

Thus, the security policies must not only specify legitimate user privileges but also aid in the detection of threats and adapt their behavior based on perceived system threat conditions. Even a single instance of a request for a vulnerable CGI script or malformed request should be reported immediately and countermeasures should be applied. Such countermeasures may include:
- generating audit records;
- notifying network servers that are monitoring security rel-

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: (1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (DEFCN); (2) DABT63-94-C-0034, Security Infrastructure for Large Distributed Systems Project (SILDS); (3) J-FBI-95-204, Global Operating System Technologies Project (GOST); and (4) DE-FC03-99ER25397, Diplomat project. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, U.S. Department of Energy or the U.S. Government. Figures and descriptions were provided by the authors and are used with permission.

evant events in the system;
- tightening local policies (e.g., restricting access to local users only or requesting extra credentials);
- modifying overall system protection. Examples include terminating the session, logging the user off the system, disabling local account or blocking connections from particular parts of the network or stopping selected services (e.g., disable ssh connections).

These actions would be followed by an alert to the security administrator, who can then assess the situation and take the appropriate corrective actions. This step is important, since an automated response to attacks can be used by an intruder in order to stage a DoS (the intruder could have impersonated a host or a user).

Traditional access control mechanisms were not designed to aid the detection of threats or to adjust their behavior based on perceived threat conditions. Common countermeasures to web server threats depend on separate components like firewalls, Intrusion Detection Systems (IDSs), and code integrity checkers. While these components are useful in detecting some kinds of attacks, they do not fully address a web server's security needs. For example, firewalls can deny access to unauthorized network connections, but they can not stop attacks coming in via authorized ports. In the general case, IDSs provide only incomplete coverage, leaving sophisticated attacks undetected. Other disadvantages include: large number of false positives and inability to preemptively respond to attacks. Integrity checkers can detect unauthorized changes to files on a web site, but only after the damage has been done.

Motivated by the multitude of web server vulnerabilities and generally unsatisfactory server protection, we propose integrated approach to web server security - the Generic Authorization and Access-control API (GAA-API) that supports fine-grained access control and application level intrusion detection and response.

The GAA-API evaluates HTTP requests and determines whether the requests are allowed and if they represent a threat according to a policy. Our approach differs from other work done in this area by supporting access control policies extended with the capability to identify (and possibly classify) intrusions and respond to the intrusions in real time. The policy enforcement takes three phases:
1. Before requested operation (e.g., display an HTML file or run a CGI program) starts; to decide whether this operation is authorized.
2. During the execution of the authorized operation; to detect malicious behavior in real-time (e.g., a user process consumes excessive system resources).
3. When the operation is completed; to activate post execution actions, such as logging and notification whether the operation succeeds/fails. For example, alerting that a particular critical file (e.g., /etc/passwd) was modified can trigger a process to check the contents of the file (e.g., check for a null password).

By being integrated with the web server and having the ability to control the three processing steps of the requested operation, the GAA-API can respond to suspected intrusion in real-time before it causes damage, whether it is site defacement, data theft or a DoS attack.

The disadvantage of the proposed approach is that a web server has to be modified in order to utilize the GAA-API. However, once the relatively easy integration is completed, it becomes possible to handle access control decisions and application level intrusion detection simultaneously. Furthermore, since the GAA-API is a generic tool, it can be used by a number of different applications with no modifications to the API code. In this paper we focus on the web server. However, the API can provide enhanced security for applications with different security requirements. We have integrated the GAA-API with Apache web server, sshd and FreeS/WAN IPsec for Linux.

2 Policy Representation

The Extended Access Control List (EACL) is a simple language that we implemented to describe security policies that govern access to protected resources, identify threats that may occur within application and specify intrusion response actions. An EACL is associated with an object to be protected and specifies positive and negative access rights with optional set of associated conditions that describe the context in which each access right is granted or denied. An EACL describes more than one set of disjoint policies. The policy evaluation mechanism is extended with the ability to read and write system state. The implementation is based on conditions that provide support for monitoring and updating internal system structures and their runtime behaviors.

A condition may either explicitly list the value of a constraint or specify where the value can be obtained at run time. The latter allows for adaptive constraint specification, since alowble times, locations and thresholds can change in the event of possible security attacks. The value of condition can be supplied by other services, e.g., an IDS.

In our framework, all conditions are classified as:

1. **pre-conditions** specify what must be true in order to grant or deny the request, e.g., `access identity`, `time`, `location` and `system threat level`.
2. **request-result** conditions must be activated whether the authorization request is granted or whether the request is denied, e.g., `audit` and `notification`.
3. **mid-conditions** specify what must be true during the execution of the requested operation, e.g., a CPU usage `threshold` that must hold during the operation execution.

4. **post-conditions** are used to activate post execution actions, such as logging and notification whether the operation succeeds/fails.

Failure of some of these conditions may signal suspicious behavior,e.g., access is requested at unexpected times or unusual locations. Some conditions can trigger defensive measures in response to a perceived system threat level, e.g., impose a limit on resource consumption or increase auditing.

A **condition block** defines a conjunction of a totally ordered set of conditions. Conditions are evaluated in the order they appear within a condition block.

An **EACL entry** consists of a positive or negative access right and four optional condition blocks: a set of pre-conditions, a set of request-result conditions, a set of mid-conditions and a set of post-conditions.

An **EACL** consists of an ordered set of disjunctive EACL entries. An EACL representation supports disjunction and conjunction of conditions to activate different control modes. A transition between the disjoint EACL entries is regulated automatically by reading the system state (e.g., time of day or the system threat level). Detailed EACL syntax is given in the Appendix.

In the current framework, the evaluation of entries within an EACL and evaluation of conditions within an EACL entry is totally ordered. Evaluation of an EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorizations is based on ordering. The entries which already have been examined take precedence over new entries.

The order has to be assessed before EACL evaluation starts. Determining the evaluation order is currently done by a policy officer. We recognize that the function of defining the order of EACL entries and conditions within an entry can be best served by an automated tool to ensure policy correctness and consistency and to ease the policy specification burden on the policy officer. We plan to design and implement such tool in the future. For further details about the authorization model see [4].

The GAA-API provides a general-purpose execution environment in which EACLs are evaluated.

2.1 Policy Composition

Policy Composition is a process of relating separately specified policies. Our framework supports system-wide and local policies. This separation is useful for efficient policy management. Instead of repeating policies that apply to all applications in individual application policies, we define these policies as a separate *system-wide policy* that is applied globally and is consulted on all the accesses to all applications. *Local policies* allow users and applications to define their own policy in addition to the global one.

The composed policy is constructed by merging the system-wide and local policies. First, system-wide policies are retrieved and placed at the beginning of the list of policies. Then the local policies are retrieved and added to the list. Thus, system-wide policies implicitly have higher priority than the local policies.

A system-wide policy specifies a **composition mode** that describes how local policies are to be composed with the system-wide policy. The framework supports three composition modes:

expand

A system-wide policy broadens the access rights beyond those granted by local policies. It is the equivalent of a disjunction of the rights. The access is allowed if either the system-wide or the local policy allows the access. This is useful to ensure that a request permitted by the system-wide policy can not fail due to access rejection at the local level.

narrow

A system-wide policy narrows the access rights so that objects can not be accessed under particular conditions regardless of the local policies. The policy that controls access to an object may have mandatory and discretionary components. Generally, mandatory policy is set by the domain administrator, while discretionary policy is set by individuals or applications. The mandatory policies must always hold. The discretionary policies must be satisfied in addition to the mandatory policies. Thus, the resulting policy represents the conjunction of the mandatory and discretionary policies.

stop

If a system-wide policy exists, that policy is applied and local policies are ignored. An administrator may require complete overriding of the local policies with the system-wide policies. This is useful in order to react quickly to an attack. One might use the *stop* mode to shut down certain component systems. This is also useful when the administrator wants to, for example, allow access to a document (e.g., a system log file) only to himself. If he specifies a policy using the *expand* mode, then additional access can be granted at the local level. If he uses *narrow* mode, the local policies could add additional restrictions that can deny the access.

To evaluate several separately specified local (or system-wide) policies, we take a conjunction of the policies.

3 GAA-API and IDS interactions

The data extracted from an application at the access control time can be supplemented with data from a network- and host-based IDSs to detect attacks not visible at the application level and reduce false alarm rate.

The current GAA-API interaction with an IDS is limited to determining the current system threat profile and adapting the security policy to respond to changing security requirements. Our next task is to support closer interaction be-

tween the GAA-API and different IDSs. Here are the kinds of information[1] that the GAA-API can report to IDS:

1. Ill-formed access requests, which may signal an attack. Because the GAA-API processes access requests by applications, the API can apply application level knowledge to determine whether the request is properly formed.
2. Accesses requests with parameters that are abnormally large or violate site's policy.
3. Access denial to sensitive system objects.
4. Violating threshold conditions, e.g., the number of failed login attempts within a given period of time.
5. Detected application level attacks. The report may include threat characteristics, such as attack type and severity, confidence value and defensive recommendations.
6. Unusual or suspicious application behavior such as creating files.
7. Legitimate access request patterns. This information can be used to derive profiles that describe typical behavior of users working with different applications.

The GAA-API can request a network-based IDS to report, for example, indications of address spoofing. This information can be used in addition to the application level attack signatures to further reduce the false positive rate and avoid DoS attacks. This is particularly important for applying pro active countermeasures, such as updating firewall rules and dropping connections.

The API can request information for adjusting policies, such as values for thresholds, times and locations. The values may depend on many factors and can be determined by a host-based IDS and communicated to the GAA-API.

4 The Apache Access Control

Apache's access control system provides a method for web masters to allow or deny access to certain URL paths, files, or directories. Access can be controlled by requiring username and password information or by restricting the originating IP address of the client request. Access control is usually confined to specific directories of the document tree. When processing client's request to access a document Apache looks for an access control file called .htaccess in every directory of the path to the document. Here is a sample .htaccess file:

```
Order Deny, Allow
Deny from All
Allow from 10.0.0.0/255.0.0.0
AuthType Basic
AuthUserFile /usr/local/apache2/.htpasswd-isi-staff
Require valid-user
Satisfy All
```

The "Allow from 10.0.0.0/255.0.0.0" allows connections only from hosts within the specified IP range. All other hosts will get a "Permission Denied" message. The "Require valid-user" requires that the user enter a username and password. These username/password pairs are stored in a separate file specified by the "AuthUserFile" directive.

5 Adding GAA-API to Enhance the Access Control of the Apache Server

Unfortunately, the current version of Apache does not support flexible fine-grained policies. Within the Apache configuration file, the directive *Satisfy All* specifies that both of the constraints on IP address and user authentication should be satisfied to authorize an access request. *Satisfy Any* means that the request will be granted if either of the two constraints is met. However, these directives can not express a policy with logical relations among three or more constraints. Therefore, new semantics must be introduced to specify a more flexible access control policy. Here are the major advantages of the integration:

1. Besides making decisions of whether a request is accepted or rejected, the GAA-API libraries provide routines that can execute certain actions, such as logging information, notifying administrator, etc. Furthermore, the routines can be activated whether the request succeeds/fails (when defined as request-result conditions) or whether the requested operation succeeds/fails (when defined as post-conditions). Thus, the GAA-API supports fine-tuning of the notification and audit services.
2. The GAA-API is structured to support the addition of modules for evaluation of new conditions. Web masters can write their own routines to evaluate conditions or execute actions and register them with the GAA-API. Moreover, the routines can be loaded dynamically so that one does not need to recompile the whole Apache package to add new routines.
3. The semantics of EACL format supported by the GAA-API can represent all logical combinations of security constraints.
4. The GAA-API supports adaptive security policies, which detect security breaches and respond to attacks by modifying security measures automatically.

[1] This information can be used locally by modules that implement the application level intrusion/misuse detection, as described in Section 7 and/or forwarded the information to IDSs for analysis.

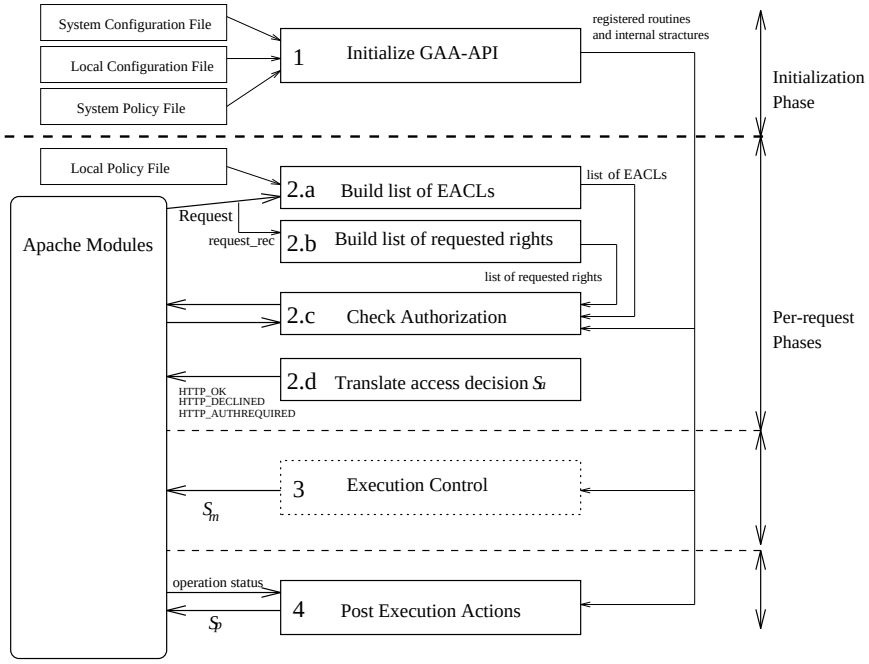

Figure 1. GAA-Apache integration

6 GAA-Apache Access Control

The GAA-API is integrated into Apache by modifying the $check\_dir\_access$ function. The "glue" code extracts the information about requests from the Apache core modules, initializes the GAA-API, calls the API functions to evaluate policies, and finally returns access control decision and status values to the modules. The GAA-Apache integration is shown in Figure 1. The GAA-API makes use of system-wide and local configuration and policy files. The configuration files list routines and parameters for evaluating conditions specified in the policy files. The system-wide policy applies to all applications in the system. The local 3 status values:
1. authorization status S_a indicates whether the request is authorized, not authorized or uncertain.
2. mid-condition enforcement status S_m.
3. post-condition enforcement status S_p.

The status values ($GAA\_YES/GAA\_NO/GAA\_MAYBE$) are obtained during the evaluation of conditions in the relevant EACL entries:
GAA_YES - all conditions are met;
GAA_NO - at least one of the conditions fails;
GAA_MAYBE - none of the conditions fails but there is at least one condition that is left unevaluated. The GAA-API returns GAA_MAYBE if the corresponding condition evaluation function is not registered with the API.

1. **Initialization phase**. When the server daemon of Apache starts, first the GAA-API is initialized by calling $gaa\_initialze$ and gaa_new_sc that extract and register condition evaluation and policy retrieval routines from the system and local configuration files, fetch the system policy file, and generate internal structures for later use.

2. The **access control phase** starts with receiving a request to access an object (e.g., HTML file or CGI script).

   (a) The $gaa\_get\_object\_policy\_info$ function is called to obtain the security policies associated with the requested object. The function reads the system-wide policy file, converts it to the internal EACL representation and places it at the beginning of the list of EACLs. Next, the function retrieves and translates the local policy file and adds it to the list. The system and local policies are composed as described in Section 2.1.

   (b) The request is converted into a list of requested rights. The context information (e.g., system configuration, server status, client status and the details of access request) that may be used by the condition evaluation routines is extracted from the $request\_rec$ structure and is added to requested right structure as a list of parameters. These parameters are classified with "type" and "authority" so that GAA-API routines that evaluate conditions with the same type and authority could find the relevant parameters.

(c) Next, the $gaa\_check\_authorization$ function is called to check whether the requested right is authorized by the the ordered list of EACLs. This function finds the EACL entries where the the requested right appears and calls the registered routines to evaluate pre- and request-result conditions in the entries. If there are no pre-conditions, the authorization status is set to GAA_YES. Otherwise, the pre-conditions are evaluated and the result is stored in the authorization status S_a.

If the request-result conditions are present in the entry, the conditions are evaluated and the intermediate result is calculated. The conjunction of the intermediate result and S_a is stored in the authorization status S_a.

(d) Finally, the status S_a is translated to the Apache format and is passed to the Apache core modules as a return value of the $check\_dir\_access$ function. GAA_YES is translated to $HTTP\_OK$ (Apache can grant the request). GAA_NO is translated to $HTTP\_DECLINED$ (Apache should reject the request). In some cases, the GAA_MAYBE is translated to $HTTP\_AUTHREQUIRED$, in other cases to $HTTP\_DECLINED$.

In particular, the GAA_MAYBE is used to enforce adaptive redirection policies. Apache may use the redirection for minimizing the network delay, load balancing or security reasons. For example, redirect to a replica server that is closest to the client in terms of network distance. The redirection policies encoded in the pre-conditions specify, characteristics of a client, current system state and URL that must serve the client. With this setup, the GAA-API first checks the pre-conditions that encode client's information and system state. The condition of type `pre_cond_redirect` encodes the URL and is returned unevaluated. When Apache receives the $HTTP\_AUTHREQUIRED$, the server checks whether there is only one unevaluated condition of the type `pre_cond_redirect` and creates a redirected request using the URL from the condition value.

3. The **execution control phase** consists of starting the operation execution process and calling the $gaa\_execution\_control$ function which checks if the mid-conditions associated with the granted access right are met. The result is returned in S_m. The implementation of this phase has not been completed yet.

4. During the **post-execution action phase** the $gaa\_post\_execution\_actions$ function is called to enforce the post-conditions associated with the granted rights. This function performs policy enforcement after the operation completes by executing actions such as notifying by email, modifying system variables, writing log file, etc. The operation execution status (indicating whether the operation succeeded/failed) is passed to the $gaa\_post\_execution\_actions$. If no post-conditions are found, GAA_YES is returned, otherwise the post-conditions are evaluated and the result is returned in S_p.

7 Deployments

In this section we describe several examples to illustrate how our framework can be deployed to enable fine-grained access control and intrusion detection and response.

7.1 Network Lockdown

We first show how our system adapts the applied authentication policies to require more information from a user when system threat level changes. Consider an organization with the following characteristics:

- Mixed access to web services. Access to some web resources require user authentication, some do not.
- An IDS supplies a system threat level. For example, low threat level means normal system operational state, medium threat level indicates suspicious behavior and high threat level means that the system is under attack.
- Policy: *When system threat level is higher than low, lock down the system and require user authentication for all accesses within the network.* Strong authentication protects against outside intruders. To some extent, authentication may help to reduce insider misuse. In particular, insiders are discouraged if the identity of a user can be established reliably.

System-wide policy:

```
eacl_mode 1 # composition mode narrow
# EACL entry 1
neg_access_right * *
pre_cond_system_threat_level local =high
```

Local policy:

```
# EACL entry 1
pos_access_right apache *
pre_cond_system_threat_level local >low
pre_cond_accessID_USER apache *
```

The system-wide policy specifies mandatory requirement "No access is allowed when system threat level is high" that

can not be bypassed by a local policy. The local policy specifies that all Apache accesses have to be authenticated if the system threat level is higher than "low".

7.2 Application level Intrusion Detection

We next show how the system supports prevention of penetration and/or surveillance attacks by detecting a CGI script abuse.
System-wide policy:

```
eacl_mode 1 # composition mode narrow
# EACL entry 1
neg_access_right * *
pre_cond_accessID_GROUP local BadGuys
```

Local policy:

```
# EACL entry 1
neg_access_right apache *
pre_cond_regex gnu ’’’*phf*’ ’*test-cgi*’’’
rr_cond_notify local
on:failure/email:sysadmin/info:CGIexploit
rr_cond_update_log local
on:failure/BadGuys/info:IP
# EACL entry 2
pos_access_right apache *
```

Entry 1 in the system-wide policy specifies mandatory requirement that members of the group `BadGuys` are denied access. Evaluation of the pre-condition `pre_cond_group` includes reading a log file of the suspicious IP addresses and trying to find an IP address that matches the address the request was sent from. Entry 1 in the local policy contains a pre-condition `pre_cond_regex` that examines the request for occurrence of regular expressions `*`phf`*` and `*test-cgi*`. If no match is found, the GAA-API proceeds to the next EACL entry that grants the request.

If this condition is met, the request is rejected.
The `rr_cond_notify` condition sends e-mail to the system administrator reporting time, IP address, URL attempted and a threat type.
Next, the `rr_cond_update_log` updates the group `BadGuys` to include new suspicious IP address from the request.

New signatures can be specified using regular expressions and numeric comparison. For example, the following pre-condition detects a particular DoS attack:

```
pre_cond_regex gnu '*//////////////////*'
```

Evaluation of this condition includes checking the request for presence of a large number of "/" characters that most likely indicates an attempt to exploit a well-known apache bug that slows down Apache and fills up logs fast.

The pre-condition `pre_cond_regex gnu '*%*'` detects malformed URLs (part of the URL contains the percent character). This may indicate ongoing attack, such as NIMDA. NIMDA exploits Microsoft IIS vulnerabilities by sending a malformed GET request.

The pre-condition `pre_cond_expr local >1000` checks that the length of input to a CGI script is no longer than 1000 characters. This condition detects a buffer overflow attacks, e.g., Code Red IIS attack.

Adding suspicious hosts to the `BadGuys` may allow our system to stop attacks with unknown signatures. Often vulnerabilities are tested by scripts that generate a number of requests. Each request exploits a particular bug. If the system identifies requests from an address as matching known attack signature, then subsequent requests from that host (initiated by the same script), checking for vulnerabilities we might not yet know about, can still be blocked. Further, since this blacklist is specified in a system-wide policy, the list is shared by many of our hosts that improves security of the system overall.

8 Performance

In our experiment, we used the system-wide and local policy files shown in Sections 7.1 and 7.2, respectively. The experiment was performed 20 times on a PC with an Intel 1.8GHz Pentium 4 CPU, running RedHat Linux v7.1.

On average, GAA-API functions took 5.9 milliseconds (ms) without email notification (53.3 ms with email notification) while running Apache functions including GAA-API functions took 19.4 ms (66.8 ms with email notification). The overhead introduced by the GAA-API is 30% if email notification is not taken into account. If the email notification is enabled, the overhead increases to 80%.

9 Implementation Status and Future Work

The GAA-API implementation is available at http://www.isi.edu/gost/info/gaaapi/source.
The API has been integrated with several applications, including Apache, sshd and FreeS/WAN IPsec for Linux.

To improve efficiency of the GAA-Apache integration we will add support for caching of the retrieved and translated policies for later reuse by subsequent requests. We will investigate a possibility of implementing a simple profile building module and anomaly detector (implemented using conditions) to support anomaly-based intrusion detection in addition to the signature-based.

We plan to implement the execution control phase for Apache. We will explore the utility of mid-conditions for protection from untrusted downloaded code, such as Java applets and Netscape plug-ins. The mid-conditions will control actions of the downloaded content on a client machine throughout the execution of the content.

We plan to design a policy-controlled interface for establishing a subscription-based communication channels to allow GAA-API and IDSs to communicate.

10 Related Work

AppShield [5] is a proprietary policy-based system that protects web servers. The AppShield intercepts and analyzes all requests and dynamically adjusts its security policy to prevent attackers from exploiting application-level vulnerabilities. It uses dynamic policy not by looking for the signatures of suspicious behavior but by knowing the intended behavior of the site and rejecting all other uses of the system.

Emerald architecture [2] includes a data-collection module integrated with Apache Web server. The module extracts the request information internal to the Apache server and forwards it to an intrusion detection component that analyzes HTTP traffic.

Both AppShield and Emerald systems are designed specifically for the web servers and can not be used for other types of applications. In contrast, the GAA-API provides a generic policy evaluation and an application-level intrusion detection environment that can be used by different applications.

Almgren, et. al., [1] provide an overview of the occurrences of web server attacks and describe an intrusion detection tool that analyzes the CLF logs. The tool finds and reports intrusions by looking for attack signatures in the log entries. However, the monitor can not directly interact with a web server and, thus, can not stop the ongoing attacks.

11 Conclusions

Traditional access control mechanisms have little ability to support or respond to the detection of attacks. In this paper we presented a generic authorization framework that supports security policies that can detect attempted and actual security breaches and which can actively respond by modifying security policies dynamically. The GAA-API combines policy enforcement with application-level intrusion detection and response, allowing countermeasures to be applied to ongoing attacks before they cause damage. Because the API processes access control request by applications, it is ideally placed to apply application-level knowledge about policies and activities to identify suspicious activity and apply appropriate responses.

12 Appendix

We use the Backus-Naur Form to denote the elements of our EACL language. Items inside round brackets, `( )` are optional. Curly brackets, `{}`, surround items that can repeat zero or more times. A vertical line, `|`, separates alternatives. Items inside double quotes are the terminal symbols. An EACL is specified according to the following format:

```
eacl ::= (composition_mode) {entry}
entry ::= pright conds | nright
pre_cond_block rr_cond_block
pright ::= "pos_access_right" def_auth value
nright ::= "neg_access_right" def_auth value
conds ::= pre_cond_block rr_cond_block
mid_cond_block post_cond_block
pre_cond_block ::= {condition}
rr_cond_block ::= {condition}
mid_cond_block ::= {condition}
post_cond_block ::= {condition}
condition ::= cond_type def_auth value
composition_mode ::= "0"|"1"|"2"
cond_type ::= alphanumeric_string
def_auth ::= alphanumeric_string
value ::= alphanumeric_string
```

References

[1] M. Almgren, H. Debar, and M. Dacier.
A lightweight tool for detecting web server attacks. *In Proceedings of NDSS 2000, Network and Distributed System Security Symposium.* The Internet Society, February 2000.

[2] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 22-36, 2001.

[3] R. Bace and P. Mell.
Intrusion Detection Systems. *NIST Special Publication on Intrusion Detection Systems.* National Institute of Standards and Technology, August, 2001.

[4] T. V. Ryutov and B. C. Neuman.
The Set and Function Approach to Modeling Authorization in Distributed Systems.
In Proceedings of the Workshop on Mathematical Methods and Models and Architecture for Computer Networks Security, May 2001, St. Petersburg Russia.

[5] Sanctum, Inc. *http://www.sanctuminc.com/*