Integrated Access Control and Intrusion Detection for Web Servers

Tatyana Ryutov, Clifford Neuman, *Senior Member*, *IEEE*, Dongho Kim, *Member*, *IEEE*, and Li Zhou

Abstract—Current intrusion detection systems work in isolation from access control for the application the systems aim to protect. The lack of coordination and interoperation 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.

Index Terms—Access control, authorization, security policy, intrusion detection, Apache Web server.

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. The Web servers are an easy target for outside intruders because the servers must be publicly available around the clock. 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 metacharacters 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. The 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 relevant events in the system;**
- **tightening local policies** (e.g., restricting access to local users only or requesting extra credentials); and
- **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 attack (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; however, they cannot stop attacks coming in via authorized ports. In the general case, IDSs

- *The authors are with USC Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292.*
  E-mail: {tryutov, bcn, dongho, zhou}@isi.edu.

Manuscript received 24 Feb. 2003; accepted 15 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118642.

provide only incomplete coverage, leaving sophisticated attacks undetected. Other disadvantages include a large number of false positives and the 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 an 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 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 intrusions and respond to the intrusions in real-time. The policy enforcement takes three phases:

1. Before the 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 process consumes excessive system resources).
3. After the operation is completed—to activate post execution actions, such as logging and notification whether the operation succeeds or fails (e.g., alerting that a particular critical file was written can trigger a process to check the contents of the file).

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.

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 the Apache Web server, SOCKS5, 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 and identify threats that may occur within application and specify intrusion response actions [5]. An EACL is associated with an object to be protected. It specifies positive and negative access rights with an 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 runtime. The latter allows for adaptive constraint specification since allowable 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. All conditions are classified as:

- **Preconditions** specify what must be true in order to grant the request (e.g., access identity, time, location, and system threat level).
- **Request-result** conditions must be activated whether the authorization request is granted or whether the request is denied (e.g., audit, notification, and threshold).
- **Midconditions** 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).
- **Postconditions** are used to activate post execution actions, such as logging and notification whether the operation succeeds or fails.

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 preconditions, a set of request-result conditions, a set of midconditions, and a set of postconditions. 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).

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 have already 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. 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

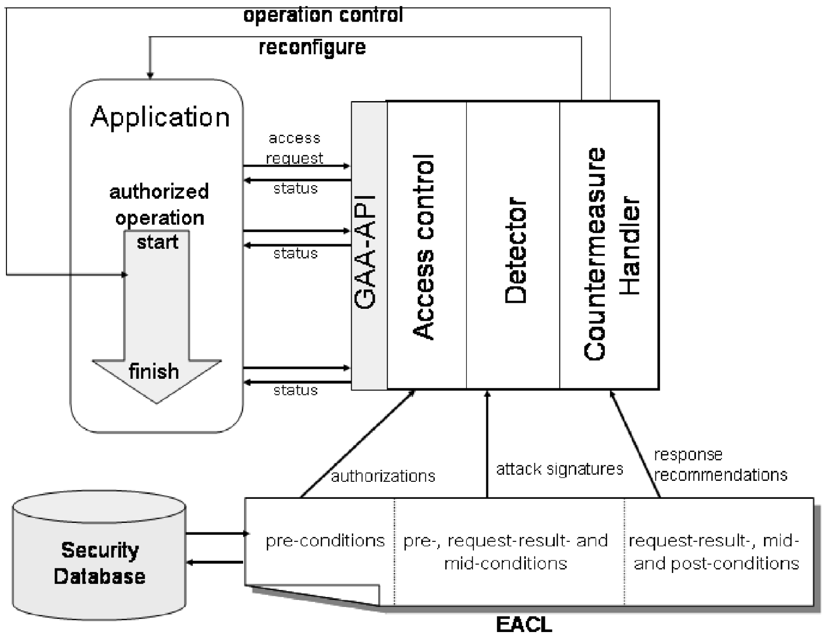


Fig. 1. Generic Intrusion Detection and response.

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 cannot fail due to access rejection at the local level.
- **Narrow.** A system-wide policy narrows the access rights so that objects cannot 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 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 Generic Application Level Intrusion Detection Framework

The system detects intrusions by comparing access request patterns against the security policies and taking some actions if the request is judged to be suspicious. Because this applies to any application, this portion of the system can be fairly generic and used for a number of applications. However, the database of known intrusion scenarios, attack patterns, and responses should be customized for different applications. The customization is done through specification of policies expressed in EACL format. Fig. 1 shows a high-level view of our framework.

The **access control** module mediates access requests generated by applications and forwarded to the GAA-API for approval. The **detector** examines access requests and determines the presence of an attack based on the policies. If the detector determines the request to be suspicious, the **countermeasure handler** will take the corrective actions to prevent malicious actions from being executed. The **Security Database** provides information collected from various sources including: user activity, misuse signatures and intrusion scenarios, application audit records, etc. The **Security policy (EACL)** contains:

- Positive and negative authorizations checked by the access control module.

- The information to be analyzed by the detector (e.g., database of attacks, user activity profiles, parameters of an access request and information obtained from monitoring the execution of the requested operation, and a status (success or failure) of the completed operation).
- Actions to be performed by the countermeasure handler for incident response—the system may deny requested access, affect execution of the requested operation (e.g., suspend or kill a process), generate alarms and audit records, update firewall rules, and so on.

4 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 the false alarm rate. The current GAA-API interaction with 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 between the GAA-API and different IDSs.

4.1 "GAA-API to IDS" Interactions

Here are the kinds of information that the GAA-API can report to an IDS:

1. **Ill-formed access requests**. Because the GAA-API processes access control requests by applications, the API can apply application-level knowledge to determine whether the request is properly formed. Ill-formed access requests may signal an attack. For example, consider an application that issues queries to a database. It is assumed that the application makes bug-free database queries. If there are errors in the access request, it may indicate that someone has compromised the application server and is performing ad hoc queries against the database.
2. **Accesses requests with abnormal parameters**. The API can report accesses requests with parameters that violate site policy or are abnormally large.
3. **Denied access**. The API can report even a single instance of access denial to sensitive system objects. The API can report attempts to access nonexistent hosts on a network, which could indicate network scanning or mapping activity and attempts to use critical commands.
4. **Exceeding thresholds**. Examples of types of events that can be controlled by the threshold detectors and reported by the GAA-API include the number of failed login attempts within a given period of time.
5. **Incidents**. The GAA-API can report detected application-level attacks.
6. **Suspicious application behavior**. The API can report unusual application behavior such as read only application creating files.
7. **Legitimate activity**. The GAA-API can communicate access request information to IDS. This information can be used to derive profiles that describe the typical behavior of users working with different applications. An automatically developed profile can be created by an IDS module that collects and processes the information about granted access rights over time and forms a statistically valid sample of user behavior that can be used for anomaly detection.

4.2 "IDS to GAA-API" Interactions

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 proactive 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. When implementing a threshold detector, the obvious difficulty is choosing the threshold number and a time interval of the analysis for a particular event. The values may depend on many factors and can be determined by a host-based IDS and communicated to the GAA-API.

5 GAA-Apache Integration

5.1 Apache Access Control

Apache's access control system [6] 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.

After receiving an access request, the Apache core modules call the *check_dir_access* function in *mod_access* or the *authenticate_basic_user*, *check_user_access* routines in *mod_auth* to check access control policies. A structured parameter *request_rec* is provided to the routines, containing information about the request. Finally, every routine returns the decision to the core modules. Three output values are defined: `HTTP_OK`—the request is granted; `HTTP_DECLINED`—the

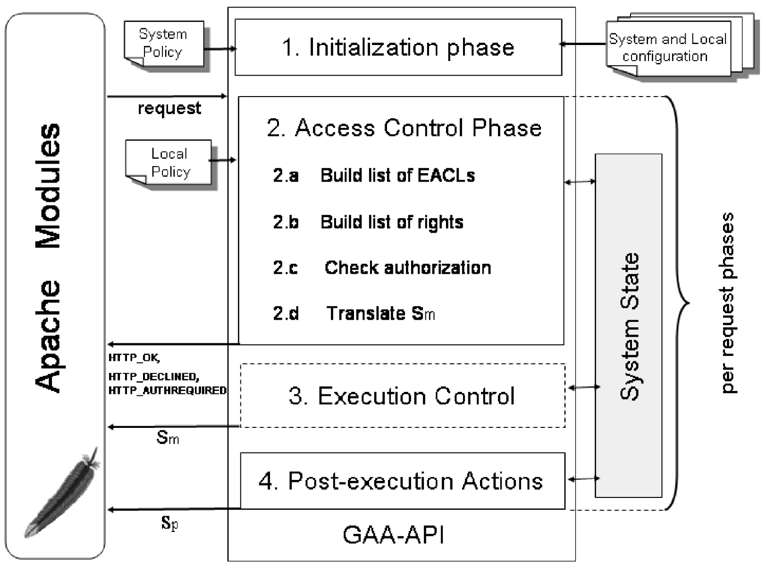


Fig. 2. GAA-Apache integration.

request is rejected; and HTTP_AUTHREQUIRED—user authentication is required to make further decision.

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

The preexisting version of Apache does not support flexible fine-grained policies that can control not only which users or groups and from which locations are allowed access, but also support other conditions, including time, system load, or system treat level. 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 cannot express a policy with logical relations among three or more constraints. With our integration of the GAA-API, these limitations are eliminated. Here are the major advantages of the GAA-Apache integration:

- The GAA-API standard libraries provide routines that evaluate conditions on time, location, token-matching, etc. They can be used to check the access control parameters for Apache. For instance, server, client, and proxy IP address can be evaluated by the location routine. Client request time, creation time, and last modified time of requested resource can be evaluated by the time routine. Protocol version number and browser type can be evaluated by the token-matching routine.
- 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 the administrator, etc. Furthermore, the routines can be activated whether the request succeeds or fails (when defined as request-result conditions) or whether the requested operation succeeds or fails (when defined as post-conditions). Thus, the GAA-API supports fine-tuning of the notification and audit services.
- 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.
- The semantics of EACL format supported by the GAA-API can represent all logical combinations of security constraints.
- The GAA-API supports adaptive security policies, which detect security breaches and respond to attacks by modifying security measures automatically.

5.3 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 Fig. 2.

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 policy file describes security requirements of Apache. The GAA-API returns three status values (GAA_YES/GAA_NO/GAA_MAYBE) to describe policy enforcement process:

1. Authorization status Sa indicates whether the request is authorized (GAA_YES), not authorized (GAA_NO), or uncertain (GAA_MAYBE).
2. Midcondition enforcement status Sm indicates the evaluation status of the midconditions.
3. Postcondition enforcement status Sp indicates the evaluation status of the postconditions.

The status values 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 fail, but there is at least one condition that is left unevaluated because the corresponding condition evaluation function is not registered with the API. Here are the three policy evaluation phases:

1. **Initialization phase**. When the server daemon of Apache starts, the GAA-API is initialized by calling *gaa_initialze* and *gaa_new_sc* to 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).

   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.
   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.
   c. Next, the *gaa_check_authorization* function is called to check whether the requested right is authorized by the ordered list of EACLs. This function finds the EACL entries where the requested right appears and calls the registered routines to evaluate pre and request-result conditions in the entries. If there are no preconditions, the authorization status is set to GAA_YES. Otherwise, the preconditions are evaluated and the result is stored in the authorization status Sa. 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 Sa is stored in the authorization status Sa.
   d. Finally, the status Sa 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 reason (e.g., redirect to a replica server that is closest to the client in terms of network distance). The redirection policies encoded in the preconditions specify, characteristics of a client, current system state, and URL that must serve the client. With this setup, the GAA-API first checks the preconditions 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 midconditions associated with the granted access right are met. The result is returned in Sm. The implementation of this phase has not been completed yet.
4. During the **postexecution action phase**, the *gaa_post_execution_actions* function is called to enforce the postconditions 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 postconditions are found, GAA_YES is returned; otherwise, the postconditions are evaluated and the result is returned in Sp.

6 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.

6.1 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 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.*
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 the mandatory requirement: "No access is allowed when system threat level is high" that cannot 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." For example, if password authentication is required, a user will be asked for a username and a password.

6.2 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 the mandatory requirement that members of the group BadGuys are denied access. Evaluation of the precondition `pre_cond_group` includes reading a log file of the suspicious IP addresses and trying to find an IP address that matches the address from which the request was sent. Entry 1 in the local policy contains a precondition `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 precondition 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 the logs fast.

The precondition `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 precondition `pre_cond_expr local` $> 1,000$ checks that the length of input to a CGI script is no longer than 1,000 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, which checks for vulnerabilities not yet known, can still be blocked. Further, since this blacklist is specified in a system-wide policy, the list is shared by many of the hosts that improves the overall security of the system.

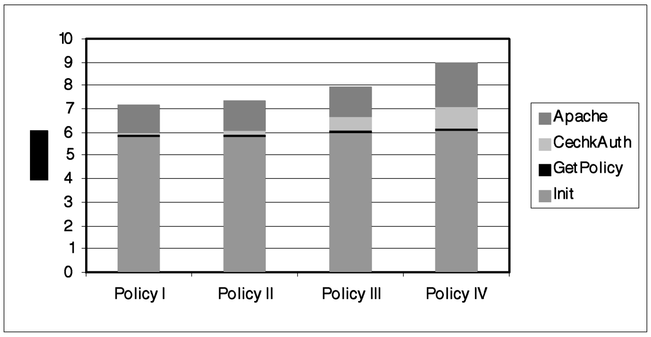


Fig. 3. Performance evaluation results.

7 Performance Evaluation

The performance of GAA-API integrated Apache server was evaluated by using four different types of policy files. Policy I does not have any conditions and always grants access. Policy II includes simpleconditions that do not need any file access. Policy III includes conditions that require reading and writing variable files and log files. Policy IV contains more expensive conditions that check user authentication and perform asynchronous e-mail notification to the system administrator. The sample policy files can be found in the Appendix.

GAA-API function calls consist of three major phases: 1) "Initialization" phase that reads the configuration and system policy files for GAA-API, 2) "GetPolicy" phase that reads the local policy file associated with the object for which the access request is submitted, and 3) "CheckAuthorization" phase that returns authorization decision.

This experiment was conducted on a PC with an Intel Pentium 4, 1.8GHz, running RedHat Linux 7.3. Fig. 3 shows the result of the experiment. The values on the table are average values of 10 runs. The entry "Apache" is the execution time the original Apache modules incurred. The "Overhead" percentage was calculated based on the values in "Apache."

TABLE 1
Performance Evaluation Results

	Policy I	Policy II	Policy III	Policy IV
Init Phase	5.8432 ms	5.8469 ms	5.9445 ms	6.0472 ms
GetPolicy Phase	0.0805 ms	0.0919 ms	0.0957 ms	0.1051 ms
CheckAuth Phase	0.0241 ms	0.1332 ms	0.6401 ms	0.9731 ms
Apache	1.2348 ms	1.2779 ms	1.2960 ms	1.8570 ms *
Overhead with Init	481.68%	475.15%	515.46%	383.70%
Overhead w/o Init	8.47%	17.61%	56.77%	58.06%

* In Policy IV, GAA-API forks a new process that sends email asynchronously. Thus, the Apache process took around 50% more time (1.86 ms vs. 1.25 ms on the average) to run because of the child process for e-mail running in parallel.

As shown in the figure, "Initialization" is the most expensive phase. However, for each GAA-Apache process, initialization needs to be executed only once at the first time GAA-API is called. The figure shows the overheads that GAA-API introduces with the first request (Overhead with Init), and the overheads for the subsequent requests in each process (Overhead w/o Init).

The "Get Policy" phase is almost constant with low values because it just reads the local policy files. The only phase whose performance is affected by having different types of policies is the "Check Authorization" phase.

From Table 1, for the first call of GAA-API in a GAA-Apache process, GAA-API incurs an overhead of more than 400 percent because of the initialization phase. However, for the subsequentcalls of GAA-API in the same process, GAA-API skips the initialization phase and significantly reduces its overhead. For the policies with conditions that do not require file access (e.g., Policy II), the overhead from GAA-API function calls to the Apache Web server is lower than 20 percent. For more expensive policies with conditions that require file access, encryption, or process forking (e.g., Policies III and IV), the GAA-API's overhead was more than 50 percent.

The sample policy files used for the evaluation are fairly short in length, but we believe that they represent most of the possible cases. The individual policy files cannot grow huge because a local policy file is associated with an object or a group of objects. This means that, even if the system-wide policy could become complex, the performance of the system will not degrade linearly because the system will evaluate only the policy file that is specifically associated with the object for which the access request is submitted.

8 Related Work

AppShield [7] 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 Web servers and cannot 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 cannot directly interact with a Web server and, thus, cannot stop the ongoing attacks.

9 Discussion

Our application-level, policy-based approach to intrusion detection and response offers several important advantages over traditional host and network-based approaches:

1. *Customization.* Instead of having an IDS look for a restricted set of predefined signatures or time-variant statistical profiles, this approach allows each organization to define suspicious events in terms of policies for accessing application-level objects. The policies take into account the organization's and application's security requirements.
2. *Flexibility.* Security policies supported in our system can be defined in terms of acceptable and unacceptable access patterns to protected resources. For example: A *Closed World* policy states that everything that is not explicitly authorized is unacceptable and may indicate suspicious behavior. It might be possible to define a minimum set of ssh commands that are allowed and then define the presence of all other commands as a violation of the policy. An *Open World* policy defines that everything that is explicitly denied is unacceptable and may indicate suspicious behavior. A *Mixed World* policy may recognize some explicitly authorized access patterns as suspicious.
3. *Preemptive response.* By being integrated with the application and having the ability to control the three processing steps of a requested operation, the system can respond to suspected intrusion in real-time. For example, the system can deny the operation, suspend the operation execution, and notify about the success or failure of the completed operation.

4. *Elimination of several IDS vulnerabilities*. Traditional IDS is susceptible to desynchronization attacks since usually the IDS does not actively participate in the connection it monitors. With the proposed approach, the attempts to desynchronize the Detection engine from application will fail because the application is able to pass information to the engine through the GAA-API. As the system monitors events at the user level of abstraction, it is not vulnerable to traffic tampering attacks such as insertion and evasion. Fast attacks on IDS (that seek to exploit application's vulnerability before the IDS can apply counter measures) will not succeed because the system processes access requests by applications, and the application waits for the result.
5. *Reduction of false negatives and false positives*. The advantages of looking for the attacks at the application level include the ability to access decrypted information about a request. A request transported to the application through an encrypted channel is not visible to a network based IDS. The ability to interface with the application directly, with significant application-specific knowledge, allows application-based intrusion monitoring to detect suspicious behavior due to authorized users exceeding their authorization or exploitation of application-specific vulnerabilities. Using this approach could potentially result in detecting a custom attack that has never been observed in the past, thus reducing the number of false negatives. Another advantage is that information on how the request is handled by the server is available at the application level (e.g., whether the requested file is interpreted as a CGI script or HTML file). Both network and host-based IDSs could not make this distinction and if configured to look for strings matching "phf.cgi" and "test-cgi," they may produce false positives.

10 Future Work

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 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 midconditions for protection from compromised or badly written CGI scripts processed at the server. We plan to design a policy-controlled interface for establishing a subscription-based communication channels to extend the GAA-API and IDSs communication.

In this paper, we have considered simple attacks that require a single action (malicious request) in order to achieve the attacker's goal. More complex and stealthy attacks require a series of actions that constitute an attack scenario. In order to detect such attacks, we will extend our system with the support for attack signatures that describe a sequence of access requests and system state conditions that represent an attack. To implement detection of such complex signatures, we will use hypothesis generation techniques. In particular, we will study the application of Bayesian methods [4] to classify observed events into attack scenarios.

In the current framework, we assume that conditions are evaluated consecutively and that authorization requests do not overlap. These two assumptions enable us to concentrate on a single condition evaluation per each time interval and, therefore, avoid the problem of coordination of multiple condition evaluation processes. However, this approach results in inefficient policy evaluation process and leads to systems that cannot scale to large numbers of objects. The future directions for this research include exploring extensions to the framework to support: concurrent requests, replication of the evaluation mechanism, concurrent evaluation of conditions within the same request, and distributed policy enforcement. At this point, the issues of spatial and temporal relationships among the policy computations become critical. Policies that govern the same object may have nontrivial interdependencies which must be carefully analyzed and understood.

Another limitation of the current framework is reliance on a policy administrator for defining condition evaluation order, which is then enforced by the framework. The limited awareness of the spatial and temporal dependencies among security policies may cause inconsistencies and undesirable system behavior. In many cases, administrators may not have a clear picture of the ramifications of policy enforcement actions; therefore, enforcing these policies might have unexpected interactive or concurrent behavior. Automation is essential to minimize human error, and it can only be used safely when there is a formal model that explicitly addresses both the spatial and the temporal aspects of dynamic authorization. Much research has been done in the area of integration of active mechanisms into relational and object-oriented DBMSs. We plan to test the applicability of methods and concepts from the field of active database systems to develop static and dynamic analysis techniques for adaptive policies. The reuse of techniques developed in the database community is necessary to apply best practices and to avoid repeating mistakes.

Finally, in order to put the developed formalism into practice, the researchers will implement a set of tools that provide graphical interfaces supporting both static activities such as:

- **A specialized interactive policy analyzer/editor**—a development tool that provides compile-time examining and detection of policy rule problems. The tool will be used to create policies with strong security guarantees, eliminating guesswork in the design, and deployment of dynamic authorization.
- **A runtime monitor** that provides runtime support for the execution rules derived from the semantic restrictions to maintain the policy processing automatically, asynchronously, and correctly.

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. The GAA-API implementation is available at http://gaaapi.sysproject.info. The API has been integrated with several applications, including Apache, SOCKS5, sshd, and FreeS/WAN IPsec for Linux.

APPENDIX

The four different types of policies used in the Section 7.
Policy I

```
pos_access_right apache *
```

Policy II

```
pos_access_right     apache *
pre_cond_access_host apache "127.0.0.1 OR 128.9.0.0/16
                            OR usc.edu"
pre_cond_access_time apache "01/01/03-12/31/05
                            MON-FRI"
pre_cond_check_regex apache "#apache.uri =' *.html' "
```

Policy III

```
neg_access_right     apache *
pre_cond_check_equal apache "%(#remote_ip.threat
                              level) = HIGH"
rr_cond_inc_variable apache "%(#remote_ip.reject
                              count)"
rr_cond_append_log   apache "%LogMsgReject"
```

Policy IV

```
pos_access_right            apache *
pre_cond_access_user        apache "%InspectedUser
                                    List"
rr_cond_async_email_notify  apache "root@localhost"
```

ACKNOWLEDGMENTS

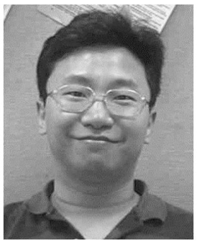
This effort was sponsored by the US Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, US Air Force Materiel Command, USAF, and the Xerox Corporation under the following agreements: 1) F30602-00-2-0595, Dynamic Policy Evaluation of Containing Network Attacks Project (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); 4) DE-FC03-99ER25397, Diplomat project; and 5) HE1254-97, XAUTH Project. The US 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 US Defense Advanced Research Projects Agency (DARPA), the US Air Force Research Laboratory, US Department of Energy, the US Government, or the Xerox Corporation. Figures and descriptions are provided by the authors and are used with permission.

REFERENCES

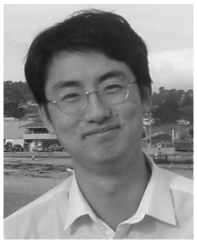
[1] M. Almgren, H. Debar, and M. Dacier, "A Lightweight Tool for Detecting Web Server Attacks," *Proc. Network and Distributed System Security Symp.*, 2000.
[2] M. Almgren and U. Lindqvist, "Application-Integrated Data Collection for Security Monitoring," *Proc. Fourth Int'l Symp. Recent Advances in Intrusion Detection*, pp. 22-36, 2001.
[3] R. Bace and P. Mell, "Intrusion Detection Systems," *NIST Special Publication on Intrusion Detection Systems*, Nat'l Inst. of Standards and Technology, 2001.
[4] D.J. Burroughs, L.F. Wilson, and G.V. Cybenko, "Analysis of Distributed Intrusion Detection Systems Using Bayesian Methods," *Proc. IEEE Int'l Performance Computing and Comm. Conf.*, Apr. 2002.
[5] T.V. Ryutov and B.C. Neuman, "The Specification and Enforcement of Advanced Security Policies," *Proc. Conf. Policies for Distributed Systems and Networks*, 2002.
[6] R. Thau, "Design Considerations for the Apache Server API," *Proc. Fifth Int'l World Wide Web Conf.*, 1996.
[7] Sanctum, Inc., http://www.sanctuminc.com, 2003.

Tatyana Ryutov received the MS degree in applied mathematics from Moscow State University, Russia, in 1991, and the MS and PhD degrees in computer science from the University of Southern California, USC, in 1999 and 2002, respectively. She joined USC/ISI in 1996 working as a graduate research assistant, and focused on the development and implementation of the access control framework for distributed systems that supports active policies, policy composition, and is sensitive to network threat conditions. Currently, Dr. Ryutov is working as a computer scientist at the University of Southern California's Information Sciences Institute with Dr. Clifford Neuman on the Dynamic Policy Evaluation for Containing Network Attacks (DEFCN) project.

Clifford Neuman received the bachelors degree from the Massachusetts Institute of Technology and, subsequently, worked at Project Athena. He received the MS and PhD degrees from the University of Washington. He is the director of the Center for Computer Systems Security at The Information Sciences Institute (ISI) of the University of Southern California (USC), associate division director of the Computer Networks Division at ISI, and a faculty member in the Computer Science Department at USC. Dr. Neuman conducts research in distributed systems, computer security, and electronic commerce. He is the principal designer of Kerberos authentication system, which, among other deployments, provides user authentication for Microsoft's Windows 2000 and Windows XP. He also developed the NetCheque® and NetCash systems, and the Prospero Directory Service. His current research focuses on the use of dynamic security policies in distributed systems that can support the formation of dynamic coalitions of cooperating organizations while adapting and responding to perceived network threats. He is a senior member of the IEEE.



Dongho Kim received the BS degree in computer engineering from Seoul National University in 1990, the MS degree in computer science from the University of Southern California (USC) in 1992, and the PhD degree in computer science from USC in 2002. He is a computer scientist at the University of Southern California's Information Sciences Institute (USC/ISI). He has been working on the Dynamic Policy Evaluation for Containing Network Attacks (DEFCN) project for three years as a member of Global Operating Systems Technology (GOST) group in Computer Networks Division of USC/ISI. He has been an instructor for the graduate-level Advanced Operating Systems course at the USC during Fall semesters since 2001. He is a member of the IEEE and the IEEE Computer Society.



Li Zhou received the BS degree in computer science from Beijing University in 2001. He is a PhD student in the Computer Science Department, University of Southern California (USC). Currently, he is working in the Generic Operating System Technology (GOST) Group, Information Science Institute, USC, as a graduate researching assistant.