Unifying Access and Resource Usage Control over Standard Client-Server Interactions

Tuan Phan and Thu D. Nguyen
{*tphan, tdnguyen*}*@cs.rutgers.edu*
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

Abstract. We propose a novel framework for integrated access and resource usage control over standard client-server interactions. Historically, access control has been developed without considering resource usage. Resource control has thus developed as an *ad hoc server-centric* set of mechanisms (e.g., file system quota, network bandwidth quote, etc.). We believe that resource usage control is strongly related to access control and so should be implemented using a unified, global enforcement framework. We introduce such a framework, where services have resource usage constraints and principals have resource usage histories. To access and use a service, a principal must have the appropriate access *and* sufficient resource usage rights when considering its usage history. Our framework is able to enforce global stateful policies, yet do not require changes to existing message-passing applications. We have built a prototype and used it to specify and enforce an example policy that includes role-based control and delegation. We applied our system to control access and resource usage for three different services, network, DNS, and SMB file systems, to demonstrate its effectiveness and wide applicability.

1 Introduction

Today's enterprise computing environments are increasingly comprised of complex conglomerates of distributed, heterogeneous components assembled to serve different classes of users with widely varying computing needs. For example, a university department has to serve at least faculty, staff, graduate students, undergraduate students, and visitors of many different types. It is not surprising then that a large body of work has explored how to address the complex needs for access control in these environments (e.g., [1, 5, 6, 12]). Researchers have also explored how to better tolerate DoS attacks, especially for services open to public access over the Internet (e.g., [15, 16, 17]). However, relatively little attention has been given to protecting computing infrastructure from resource abuse by insiders.

Users inside an enterprise may abuse its computing resources for a variety of reasons. Abuse may be unintentional. For example, a user may be running a bandwidth-intense application on a mobile device at home (e.g., peer-to-peer content download) and forget to stop it when using the device at work. (This exact scenario has occurred numerous times in our department, placing uncomfortable loads on our wireless network.) Or, a malware may have made its way onto the user's mobile device and is abusing resources in the surrounding environments without the user's knowledge (e.g., [14]). Abuse may also be intentional (although not necessarily malicious). For example, when high-speed network access first became available in our students' dorm rooms, they quickly flooded our Internet access points with downloading activities.

In this paper, we argue that insider resource abuse is a serious problem that thus far institutions have had to address in an *ad hoc* manner. We propose a novel control framework that unifies access and resource usage control to address this problem. We unify the two control problems because they are highly related. Both forms of control depend on users being authenticated and derive from the same set of credentials and/or rights. Logically, users should not be able to use services without having both the rights to access the services *and* to use their resources. Further, the most efficient way to block a user abusing his resource usage rights is to revoke his access rights.

Our presentation is developed as follows. In Section 2, we first present an example policy to show the type of control that is possible using our framework and to provide a concrete context for discussion. This

example policy defines role-based access and resource usage constraints with delegation over three different services: networking, DNS, and file system. (Our control of the network is similar to what is currently achievable with firewalls and use of various tools and *ad hoc* programming. However, our framework offers the advantage of integrated control over the many different resources and services of an enterprise.)

In Section 3, we develop a role-based model for unified access and resource control. While we use RBAC to provide a concrete context, we note that our work is not intrinsically tied to RBAC. Rather, our resource control model can be applied to any system that supports user authentication and the trusted maintenance of per-user state, which is used to hold users' resource usage histories. Along with the model, we discuss a number of challenges and issues concerning resource usage accounting and delegation of resource usage rights, and our solutions.

In Section 4, we then describe an implementable framework. This framework is based on a coordination and control system that we previously designed to enforce *stateful, communal* policies over standard client-server interactions without requiring application changes [1]. Specifically, our framework builds a set of reference monitors from the ubiquitous firewalls in today's computing environments, including per-node software firewalls and dedicated firewalls embedded within the networking infrastructure. The reference monitors are used to protect enterprise services. While clients can use standard software (e.g., SMB file system client), they have to convince the reference monitors to forward messages to the protected services by presenting appropriate access and resource usage rights. We use Law Governed Interaction (LGI) [10] to coordinate and control interactions between clients, reference monitors, and enterprise security services so that, together, they implement a coherent global policy.

To instantiate the framework for a particular system, a core reference monitor would need to be adapted for each protocol to be controlled. The design of each reference monitor would include a configurable set of resources that can be controlled during operations. System administrators would define control policies and configure the reference monitors appropriately at deployment time. In Section 4.4, we describe a prototype instantiation of our framework that can be used to enforce the example policy from Section 2. This prototype shows that our framework can be implemented and deployed with relatively small efforts.

In Section 5, we explore the performance overheads of our control system and its efficiency in protecting against various abuse scenarios. Results show that our system achieves its goals with modest performance overheads.

Overall, our contributions include: (1) developing a unified model for access and resource usage control, (2) defining a framework for enforcing sophisticated policies over client-server interactions external to the implementations of these components, (3) demonstrating that our framework can protect a wide variety of services, and (4) showing that our framework can enforce realistic policies that limit resource abuse with low performance overheads.

2 Motivating Example

We begin by describing an example RBAC policy for controlling accesses to and resource usage of three widely different services to motivate the type of policies that we seek to support.[1] Our example supposes an enterprise with a set of registered principals (users) P, a set of roles R, and a function $PR : P \rightarrow 2^R$ defining the roles that each principal can assume. Then, the enterprise might wish to establish the following policy $\mathcal{P}_E$ to protect services within its computing infrastructure:

Authentication: Each principal must authenticate to establish his identity before accessing most enterprise services from a client machine. Unauthenticated principals may only assume the *unauthenticated* role. An

[1] Our implementation actually explores the control of several resources not mentioned here, including network NAT entries and the number of opened files. We simplified this example policy and the description of our implementation to meet space constraints.

authenticated registered principal p may assume all roles in $PR(p)$. An unregistered principal may also authenticate and assume the *guest* role.

Delegation: A registered principal p may delegate rights to other principals under any role R_d where $R_d \in PR(p)$. Each delegation specifies the role to be delegated, and optionally, limitations on the delegation to specific rights on specific objects and an expiration time. (This provision applies to all three services mentioned below, although we leave the service-specific definitions of objects and rights until our discussion of our prototype implementation in Section 4.)

Network: Partition the enterprise network into zones, with an Internet zone representing the networks external to the enterprise. There is a distinguished Bootstrapping zone that hosts bootstrapping services such DNS and security services such as authentication. Communication between internal zones and the Bootstrapping zone is allowed for all roles. Communication between internal zones other than the Bootstrapping zone is allowed for all roles except *unauthenticated* and *guest*. Communication between internal zones and the Internet is allowed for all roles except *unauthenticated*. (This provision allows authenticated guests to connect to their home network but not local services.) Each role $r \in R$ may only use up to rate B_r bandwidth when communicating with the Internet.

DNS: All roles may use the DNS service. However, the role *unauthenticated* is constrained to a maximum request rate of D_u, while all other roles are constrained to a maximum request rate of D_a.

File Systems: All roles except *guest* and *unauthenticated* may mount all file systems. Owners of directories and files can define per-object access control policies using the roles in R. Each role $r \in R$ may only have up to M_r simultaneously active mounts and may only create files no faster than rate C_r *aggregated across all file systems*.

While the above policy has been deliberately made simple for ease of presentation, it still demonstrates several important elements for establishing powerful yet flexible access and resource usage control policies. First, it controls the usage of a variety of different resources across three very different services. As shall be seen, the chosen resource constraints require the application of our framework to different types of resources and different ways of resource accounting. Furthermore, if usage of some of these resources were not controlled, abusive users can significantly impact service performance, leading to service crashes in some instances. Thus, the chosen resource constraints are meant to demonstrate the applicability of our approach to all different types of resources in a wide range of services.

Second, the policy's support for delegation empowers users to coordinate service access and usage without having to involve system administrators. For example, a professor can allow his TA to access course-related files by simply delegating rights to an appropriate directory. Further, the professor can specify a delegation expiration time of the end of the term to ensure that the TA does not retain access rights forever. The professor can similarly delegate access rights to visitors and collaborators so that gaining access for these principals becomes just a matter of presenting a set of certificates.

Third, the requirement that resource usage constraints for file systems be enforced on aggregated usage across all file systems demonstrates the *communal* aspect of our framework. In this paper, we concentrate on resource usage control rather than other aspects of our framework. We note, however, that the capability to enforce policies over a community of services–e.g., a single resource limit over multiple servers or revocation of access rights because of violations at multiple different services–is a powerful advancement over server-by-server enforcement.

3 Resource Usage Control

In this section, we first develop an extended RBAC model that includes resource usage control. We then discuss a number of issues that pose interesting challenges to the realization of the extended model.

3.1 Model

A (simplified) RBAC model can be defined as follows.[2] Let:

P be a set of principals, R a set of roles, C a set of credentials, O a set of objects, A a set of actions,
$PR : P \times 2^C \rightarrow 2^R$ a function that maps a principal's identity and a specific set of credentials to the set of roles that he may assume, and
$AP : O \times A \times 2^R \rightarrow \{granted, denied\}$ a function that determines whether an access attempt by some principal that can assume a specific set of roles should be granted. (This function is essentially an access control matrix.)

Then, an access attempt, defined as an action a on an object o, by principal p, is allowed if and only if $AP(o, a, PR(p)) = granted$.

To introduce resource control, we extend the above model as follows. Let:

RS be a set of resources,
$L : RS \times R \rightarrow \mathcal{R}$ a function that specifies the limits of resource usage by roles ($\mathcal{R}$ denotes the set of real numbers),
$U : O \times A \rightarrow \{(rs, n) \mid rs \in RS \wedge n \in \mathcal{R}\}$ a function that maps each (action, object) pair of a request to the amounts (n) of different resources (rs) that would be used to complete that request, and
$p.UH : RS \rightarrow \mathcal{R}$ a per-principal (p) function that gives the principal's resource usage history.

Then, given a request (a, o) by p, the request is permitted if and only if:

$$[AP(o, a, PR(p)) = granted] \wedge [\forall (rs, n) \in U(o, a), \exists r \in PR(p) : (p.UH(rs) + n) \leq L(rs, r)] \qquad (1)$$

In essence, the above condition states that a request is permitted if any combination of roles that the principal can take on gives him the necessary access right and sufficiently high resource constraints.

Note that there are two possible approaches. In specifying the above constraints, we have adopted the approach that the set of access and resource usage rights is the union of all rights conferred by all active roles that the principal may assume. An alternative, more restrictive, approach, is to require that there exists a single role that confers both the needed access and resource usage rights. The choice between these two approaches does not significantly affect our system design and implementation and so we have arbitrarily chosen the less restrictive approach.

Finally, note that $p.UH$ should track the cumulative resource usage of p, and so should change over time based on p's actual resource usage. We do not model this change for simplicity; our implementation, of course, supports the dynamic tracking of resource usage of each principal.

[2] We do not formally define what a credential is for simplicity. Intuitively, a credential can simply be a certificate signed by a well known authority specifying that a principal p can assume some role r. Such certificate-based credentials naturally supports delegation (although our model would need to be extended a bit further to support the delegation constraints defined in $\mathcal{P}_E$).

3.2 Resource Types

In their work on building DoS resistant software [12], Qie et al. distinguished between two types of resources: *renewable* and *non-renewable*. A renewable resource is one whose supply is refreshed over time; examples include CPU cycles and network bandwidth. A non-renewable resource is one with a fixed, limited supply. When non-renewable resources are acquired through the execution of requests on a principal's behalf, they must subsequently be released or else the server will eventually run out of these resources. Examples of non-renewable resources include processes, buffers, and storage space.

We adopt the same categorization in this paper. Then, constraints on usage of renewable resources are specified as rates, where each principal that can assume a particular role may use the resource at up to some maximum rate. Constraints on usage of non-renewable resources are specified as caps. Each principal that can assume a particular role may accumulate up to some maximum amount of a non-renewable resource, beyond which he must release some before acquiring more.

3.3 Resource Usage Accounting

Resource control introduces a number of accounting challenges, some of which arise because our framework (introduced below) enforces resource usage constraints external to servers. First, a fundamental difficulty is that it may not be possible to accurately predict the resources that will be consumed by a request. (In many cases, prediction is possible either because requests of a particular type consumes the same amount of resources or because resource usage is computable from an observable attribute of the request; for example, we can predict exactly how much network bandwidth will be used to send a given packet.) Rather, resource usage has to be tallied during or after request processing. External resource control exacerbates this problem because internal server state is hidden from the reference monitor. For example, consider a write request to a file system. Unless the reference monitor has detailed information about the file (e.g., the current length of the file and the position where the current write will take place), it cannot know whether the write will overwrite existing content, and so does not consume additional storage, or will append to the file, and so does use additional storage. When it is not possible to predict the resource usage of a request, we set U to return expected (e.g., average or 0) resource usage. We then use a monitoring system to adjust the subject's resource usage history in the background.

Second, without carefully studying server implementations, it is impossible to derive server-specific non-renewable resources (e.g., internal data structures). We address this challenge by introducing abstract resources. That is, given a request, it is possible to postulate that the request will acquire some amount of non-renewable resources. For example, a mount request sent to a file server would presumably require in-memory data structures to keep track of the mount. We thus define an abstract *mount resource* to represent the fact that a mount request acquires some non-renewable resources inside the server. Then, as long as we constrain the number of abstract mount resources used by a client, we can also prevent abuses of actual resources without knowing exactly what resources and how much is used.

A similar challenge arises for measuring usage of renewable resources. For example, it is impossible to measure the CPU utilization of a specific request external to the server since the server may be processing many different requests simultaneously. We also solve this challenge by using abstract resources. For example, we can postulate that each request would use some amount of resources such as CPU, temporary memory usage, and/or network bandwidth. If we limit the rate of requests, we can indirectly limit the resource usage on the servers. We can even weight resource usage of requests differently if the amount of resource usage is related to an observable attribute of the request. For example, a file service read request asking for 10KB is likely to consume more processing resource than one that asks for 1KB. On the other hand, if requests that look similar to our reference monitors can actually consume significantly different amount of resources, then the accuracy of our control mechanism will be limited; that is, it may be possible

for a user to abuse resource usage on a service even while obeying policy constraints by sending resource-intensive requests that look similar to non-resource-intensive requests.

Finally, non-renewable resources that are acquired through one request and subsequently released through another request–e.g., mount and unmount–may be released implicitly by servers because of timeouts. For example, if the server does not hear from a client for a period of time after a mount request was successfully served, it may decide that the client has silently disengaged; the server may then perform the unmount on behalf of the client and release all acquired non-renewable resources. Our infrastructure needs to account for such implicit resource release and so must be aware of timeouts. How long are the timeouts could be set easily based on the configuration parameters in the server, but the timeout resetting requires per-object states in our infrastructure. Each timeout object will have its own timer, and the timeout timer will be reset when the system receives a corresponding request.

3.4 Delegation

Supporting delegation introduces several interesting questions. Specifically, should it be possible to delegate resource usage rights? While more limiting, it is entirely possible to allow delegation of access rights but not resource usage rights. In this case, principals are limited by the resource constraints placed on their primary (non-delegated) roles. For example, with respect to policy $\mathcal{P}_E$, every guest would be under the same set of network bandwidth usage constraints, regardless of what delegated access rights they have received from registered principals.

Alternatively, allowing delegation of resource usage rights introduces two additional questions. First, should the accumulated amount of resource usage delegated by a principal be controlled? For example, suppose a principal p may take on a role r that gives him the right to use up to m amount of some resource. If we do not control p's ability to delegate resource usage, then p may delegate r to x other principals. If p and all x receiving principals start using the resource simultaneously, then, effectively, we have allowed p to use $(x+1)m$ of the resource, as opposed to just m. On the other hand, ensuring that p and all of his delegatees use no more than m of the resource requires a credit-based tracking scheme, which introduces implementation complexity (but is certainly supportable in our framework). A compromise solution, which we assume in this paper, is to control the number of delegations that p may make. This can be implemented by either having the authority (*ca* in $\mathcal{P}_E$) refusing to validate more than n delegation by p or having the control system refusing to allow more than n delegations from p to be simultaneously active.

A second issue is, should the delegating principal be able to constrain *how* the receiving principal uses the delegated resource rights. In $\mathcal{P}_E$, when delegating a role r, it is possible for the delegating principal to constrain the delegation to just a subset of r's rights. For example, r might have read and write access to all objects in a file system; a delegation of r, however, might only include read access on a directory and its descendants. Extending this delegation model, the principal might want to constrain that the resource rights accompanying r be used only for the delegated rights on the delegated objects. We do not support this flexibility because the required fine-grain accounting system seems overly complex. On the other hand, constraining a delegation to use less than the system's resource limit is relatively easy to support; for example, the delegation of a role r with a limit L on usage of some resource might specify a limit $l < L$.

Finally, suppose that a principal with a role r_1 is delegated role r_2 with a higher limit for the use of some non-renewable resource such as disk space. What should happen when the delegation times out, yet the principal has acquired more of the resource than is allowed under r_1? Since it may be quite difficult to force the principal to reduce his resource consumption immediately, we simply do not allow the principal to acquire more of the resource until his usage has dropped below the limit for r_1. This implies that a delegation may have long lasting consequences, even after the delegation has expired (or been revoked).

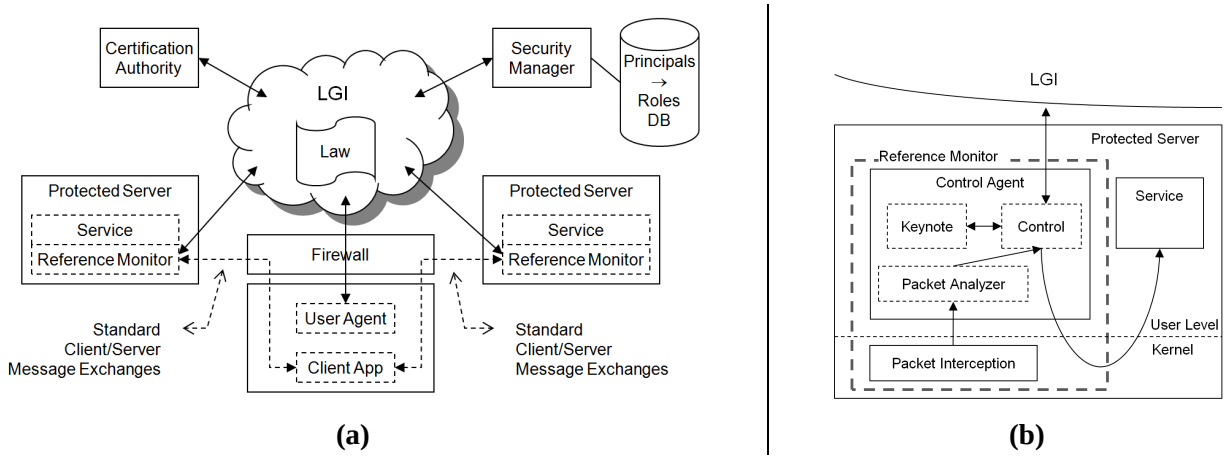


Figure 1: (a) Overview of the control framework, and (b) Internals of the reference monitors.

4 Implementation

We now present a framework that can be used to support access and resource usage control policies such $\mathcal{P}_E$. In this section, we first give an overview of the framework. (Note that the framework, without resource usage control, was developed in previous work [1]. Nevertheless, we give an overview of the entire framework to provide needed context.) We then briefly describe LGI and KeyNote, two existing tools that our framework uses. Next, we describe a generic reference monitor that can be adapted to work with different protocols. Finally, we describe an instantiation of our framework that enforces policy $\mathcal{P}_E$.

4.1 Overview

As shown in Figure 1(a), our framework comprises a set of reference monitors, a set of user agents, and a set of security services that coordinate using Law-Governed Interaction (LGI) to enforce the enterprise security policies. In general, each node hosting software components subjected to enterprise policies would also host a reference monitor. This reference monitor filters all messages, passing through only those allowed by the policies in effect. Figure 1(a) shows the reference monitors running only on the servers because this is sufficient to enforce policies that only control accesses to enterprise services such as $\mathcal{P}_E$. Existing message passing applications such as SMB clients and servers will then interact as normal. However, this interaction is only possible if the client's user can convince the reference monitor protecting the server to pass the client's requests through to the server.

Users can convince reference monitors to accept their clients' requests by obtaining and presenting appropriate credentials. When resource usage control is in effect, users must also present their resource usage histories. These processes involve interactions between the user agents, reference monitors, and security services. We use LGI [10], a mechanism for controlling message exchanges between a community of software agents according to an explicit *law*, as a secure platform for mediating these interactions.

Each user agent is some user's gateway to our framework. With respect to $\mathcal{P}_E$, each user would use his user agent to authenticate and establish his roles and resource usage history.

The security infrastructure is typically comprised of components such as the certification authority and security manager shown in Figure 1(a). With respect to $\mathcal{P}_E$, these components perform functions such as certificate generation and mapping of principals to their roles.

Each reference monitor is a combination of a kernel-level packet interception module and a user-level control agent (Figure 1(b)). The interception module intercepts and forwards packets to the control agent. The control agent reassembles these packets into meaningful protocol-level requests using a protocol an-

alyzer such as Ethereal (http://www.ethereal.com/). For each request, the control agent then extracts the information needed for controlling the request, namely the principal, the objects to be accessed, and the action being attempted. It also derives the resources that will be used and computes the predicted usage amounts.

To determine whether a request should be permitted, the control agent gathers all information that may affect the decision from appropriate components in the framework. With respect to $\mathcal{P}_E$, the control agent would retrieve the roles and resource usage history for the requesting principal from the principal's user agent. Currently, each reference monitor also acts as a service manager for the service it is protecting. In this role, it already holds the per-object access control policies and resource constraint policies. For services comprised of multiple servers, it may be desirable to separate this functionality from the reference monitor.

Once it has gathered the necessary information, the control agent uses KeyNote [4] to compute whether the request should be permitted. If the request is permitted, then the message will be forwarded to the server. Otherwise, the request will be denied.

Information also flows from the control agent to user agents, the security infrastructure, and the service managers. With respect to $\mathcal{P}_E$, a user's resource usage history will need to be updated if a request is allowed. While not required by $\mathcal{P}_E$, in other instances, misbehaviors observed at a reference monitor might be reported to allow global system responses. For example, the global policy might stipulate that a user machine should be blocked entirely from accessing the enterprise computing infrastructure if misbehavior is detected at multiple control points.

For efficiency, we assume that control agents can cache information so that they do not need to communicate with remote services to mediate every request. This information is refreshed periodically, with the refresh period being configurable in the controlling LGI law. User agents can actively push information to reference monitors to avoid lags when the managed rights of their users change. However, in the worse case, the system may not react until reference monitors refresh their caches. This is no different than when revocation is supported in a distributed system.

As discussed in [1], the above framework can support a rich domain of *stateful communal* policies in a variety of environments. Our work in this paper demonstrates that this richness can be extended to enforcement of resource usage control.

4.2 Background: LGI and Keynote

LGI. As already mentioned, LGI is a control mechanism that governs the interaction between a community of distributed agents according to an explicit law. To differentiate between the agents, the law can optionally control the management of (small amounts of) per-agent trusted state. LGI governs inter-agents interactions by requiring messages to be forwarded through a set of trusted agents, called LGI controllers, and trusted state to be maintained at the controllers. Laws are written as event-driven programs in a specialized Prolog- or Java-based language. A law can determine whether messages arriving at the controllers should be forwarded, update trusted state based on observed messages (and other events), and emit messages (e.g., to exchange information about the managed trusted state).

We will limit our description of LGI here because it is not central to the contributions of this paper. We use LGI because it is an elegant decentralized mechanism that allows our framework to scale to large and/or geographically distributed enterprises. (Our use of LGI is described extensively in [1].) For this paper, LGI can simply be viewed as a service that supports the trusted management and exchange of rights (roles and delegations) and resource usage histories. For example, a user would established that he has some valid delegated rights by presenting delegation certificates to LGI under the controlling law. The law would then validate the certificates and maintain the user's roles as part of his trusted state. Subsequently, a reference monitor can obtain the user's roles through LGI to verify that the user has the right to perform some request. By interacting through LGI, reference monitors can trust the roles and resource usage presented by user

agents, reference monitors and the security services can trust each other in order to collaborate to enforce global policies, and user agents can trust resource usage updates from reference monitors.

KeyNote. KeyNote is a trust management tool built to answer the question: does a set of credentials C and attributes A prove that a request r from principal p is allowed by a security policy P? Credentials and policies are written using KeyNote's assertion language, where each assertion is essentially a delegation of some rights. Credentials are cryptographically protected and validated during a KeyNote evaluation while policies are trusted. Attributes are trusted name/value pairs.

Each KeyNote assertion has three parts, an *Authorizer* field that identifies the delegating principal, a *Licensees* field that specifies a set of receiving principals, and a *Conditions* field that specifies the rights delegated, subjected to a set of conditions on the attributes. Rights conferred by policies appear as assertions with *POLICY* as the authorizer and are said to be root assertions.

When invoked, KeyNote performs a depth-first search through (C, A, P) to find a directed assertion chain connecting a root assertion to one whose *Licensees* contains p's identity, where: (1) for each connected pair of assertions $a1 \rightarrow a2$, *Licensees* in $a1$ contains the *Authorizer* in $a2$, and (2) all conditions along the chain evaluate to *granted*. If such a chain exists, then the request is *granted*. Otherwise, it is *denied*.(Example described in the next section.)

4.3 Reference Monitor

Currently, the reference monitor has to be adapted specifically for controlling each different protocol. This task, however, is eased considerably by a significant core that is shared by all reference monitors. We expect that each reference monitor designed to control a specific protocol would include a set of resources that can be controlled. Further, the implementation should be configurable so that system administrators can choose which of the resources will be controlled at instantiation time.

Request Interception. The reference monitor request interception module is implemented using NETFILTER (http://www.netfilter.org) and a kernel module. While abstractly each reference monitor intercepts and control all messages, in practice, it is possible to relax this requirement in two ways to reduce overheads: (1) automatically pass through messages that do not need to be controlled, and (2) perform resource control in the background, out of the critical path of request processing.

SMB *read* and *write* requests are examples of when the first optimization can be applied. These requests do not need to be intercepted if there is no associated resource usage control because access control was already performed when the file was opened and the server itself will refuse the request if it is inappropriate. Thus, a reference monitor would typically be programmed to intercept only the needed subset of messages (although automatic forwarding is limited to those messages whose packets can be easily reassembled and interpreted, e.g., single packet messages, to limit the complexity of the kernel module).

The latter optimization may be applied in cases where some inaccuracy is acceptable (and an access control check is not necessary). When a request is received, the interception module would extract the needed information for resource usage control and forward the request *before* passing the extracted information to the control agent. If there is a resource usage violation, then the control agent will set a flag in the interception module to disable future forwarding. This blockage can be implemented very inexpensively if denied requests can simply be dropped at the networking level. As a further optimization, for resources where each request is expected to use only a small amount of resource, the interception module might batch information across a number of requests before forwarding to the control agent.

Enforcement. As already mentioned, the control agent reassembles intercepted packets into request messages and extracts the principal, action, and object from the request and computes the predicted resource consumption. Next, the control agent gathers all the information necessary to decide whether the message should be forwarded, including user credentials, history of resource usage, and the target objects' access and

resource usage control policies. It then performs the checks detailed in Equation 1 using KeyNote, where the principal's roles are passed as its identities, delegations and per-object policies as policies, and the action and resource usage as attributes.

For example, suppose that a principal p has been delegated a role R_d and attempts a *read* request on file f. Further, suppose that f has an attached policy specifying that R_d has *read* permission and the file system has a constraint that specifies R_d can use up to Max_{rs} of some resource *rs* needed to perform the *read*. Then, the control agent would create the policy assertion (a) below, while the file's policy would contain assertion (b), and the file system's policy would contain assertion (c).

Authorizer: "R_d"	Authorizer: "POLICY"	Authorizer: "POLICY"
Licensees: "K_p^{pub}"	Licensees: "R_d"	Licensees: "R_d"
Conditions:	Conditions:	Conditions:
("true") $\rightarrow$ "granted"	(operation == "read") $\rightarrow$ "granted"	($U_{rs} \leq Max_{rs}$) $\rightarrow$ "granted"
(a)	(b)	(c)

The control agent would invoke Keynote, passing it the set of policy assertions {assertion (a) $\cup$ f's policy}, K_p^{pub} as p's identity (role), and (*operation*, *read*) as an attribute. Keynote would reply with *granted*.

The control agent would then invoke KeyNote a second time, passing it the set of policy assertions {assertion (a) $\cup$ the file system's policy}, K_p^{pub} as p's identity, and (U_{rs}, u) as an attribute, where u is equal to p's historical usage of *rs* plus the amount predicted will be used to perform the *read*.

Resource Usage Accounting. If the usage of a specific resource is predictable from the request, then the control agent can update the requesting principal's resource usage history. It does this by forwarding an LGI-controlled message to the principal's user-agent, which will cause LGI to update the trusted state corresponding to the user's resource usage history. The control agent may accumulate resource usage across a number of requests before updating a principal's history to minimize messaging overheads. If resource usage is unpredictable, then we would need an external monitoring system that can observe and update resource usage histories in the background.

The external monitoring system should be authorized by the LGI law before updating unpredictable resource usage. Overtime, the external monitoring system will update the user's resource usage history stored at user agent. The reference monitor's cached information can be updated either when the user agent pushes new history information (see Section **??**) or when the reference monitor discard its cache information because of age and request new information from the user agent. This weak synchronization may cause the inaccurate enforcement situation in which users may overuse resources.

4.4 Case Study

To demonstrate the flexibility and power of our proposed framework, we have instantiated an implementation that can enforce the example policy $\mathcal{P}_E$. Our instantiation includes an LGI law $\mathcal{L}_E$, a security manager, a user agent, and reference monitors to control networking, DNS, and SMB file system services. Authentication is performed by mapping a user's public key to his client machine's IP address.[3] Delegations are conveyed by certificates from a well-known certificate authority.

The security manager is an LGI-aware application that maintains a database of registered users and their assigned roles using the Berkely DB (http://www.sleepycat.com/). Each registered user can assume at least one role named by his public key. The user agent is a simple LGI-aware application that allows its user to present certificates for authentication and gaining delegation rights. Its trusted LGI state holds the user's

[3]This authentication method has two disadvantages: (1) it is vulnerable to IP spoofing, and (2) it cannot differentiate between multiple principals logged into the same client machine. The first issue can be solved in a variety of ways, including the use of IPSec [1]. The second issue can be addressed by recent proposals such as the user-based IP accounting in [11] and the application-level IPSec [18].

resource usage history. Each reference monitor is an adaptation of the core described above to handle the specific requests of a protocol. Some relevant details are described below. $\mathcal{L}_E$ controls the authentication and validation of delegation certificates, and trusted management and exchange of roles, delegated rights, and resource usage histories between the user agents, security services, and reference monitors.

Networking. The adaptation of the reference monitor to control networking traffic was quite simple as the needed functionality closely match already existing netfilter capabilities. We only modified the kernel module to extract the number of bytes from UDP and TCP packets, and batch the reporting of bandwidth usage to the control agent. Access control can be efficiently enforced by dynamically adding and removing netfilter rules, either at dedicate firewalls protecting subnets or at software firewalls on individual hosts. When resource violations occur, the offending user is blocked by removing the appropriate netfilter rules. Our adaptation required 470 lines of lightly commented code, 280 user-level and 190 kernel-level, most of which was to control the number of NAT entries (which is not discussed here because of space constraints). This count does not include libraries such as Ethereal.

To implement $\mathcal{P}_E$, we define two types of objects for our enterprise network: *zones*, where a zone is a group of one or more IP subnets, and *flows*, where a flow is defined as communication stream belonging to a specific protocol (e.g., TCP) between two zones, possibly limited to a specific port range in each zone.

We then define: (a) four zones for subnets hosting clients, bootstrapping network (e.g., DHCP and DNS) and security services (called the *bootstrapping* zone for short), other enterprise services (called the *service* zone), and the Internet; (b) six flows, two for UDP and TCP between the client and bootstrapping zones, two between the client and service zones, and two between the client and Internet zones; (c) a renewable resource representing bandwidth to/from the Internet; (d) a security policy for each client-bootstrapping and client-Internet flow allowing access for all defined roles except for *unauthenticated*, (e) a security policy for each client-service flow allowing access for all defined roles except for *guest* and *unauthenticated*, (f) a policy that specifies the bandwidth usage constraints given in $\mathcal{P}_E$, and (g) a function that maps bandwidth usage to the number of bytes in intercepted packets.

The enforcement of the policies and resource constraints above are then performed exactly as described in Section 4.3.

DNS. The adaptation of the reference monitor to control DNS requests was also simple, as there are only a few different request types. All requests are intercepted and forwarded to the control agent. Our adaptation required about 150 lines of user-level code. To support $\mathcal{P}_E$, we defined a renewable *request-rate* resource. We then defined a function that maps all request types to usage of one unit of the request-rate resource, and two policies to specify the resource usage constraints.

SMB. The adaptation of the reference monitor to control SMB requests was somewhat more challenging, requiring 810 user-level and 50 kernel-level lines of code. Part of this adaptation was done in previous work [1]. The main implementation challenges included: (1) mapping file (directory) ids to file names, and (2) rewriting requests to force the server to grant or deny the request as needed by the reference monitor. The first challenge arises because we need file (directory) name to look up the per-file access control policies. Fortunately, Ethereal already implemented the logic necessary to extract these mappings. Further, the SMB protocol includes explicit open and close requests, which simplifies the management of the mapping. The second challenge arises because SMB is implemented on top of TCP, so that our reference monitor cannot deny requests by sending an error to the client and dropping the request. Doing so can break the semantics of TCP streams and affect the TCP congestion control, respectively.

We defined three types of objects: file systems, directories, and files. Each object can have an attached access control policy. We implemented a policy database using the Berkeley DB to store these policies. Policies are inserted and deleted as directories and files are created and removed. Users can access and modify policies attached to directories and files that they own.

To support $\mathcal{P}_E$, we defined an access control policy for the file system object that allows mount access

for all defined roles except for *unauthenticated* and *guest*. We defined two resources, *mount* (non-renewable) and *create-rate* (renewable). We defined a function that maps mount requests to using one unit of the mount resource and create requests to using one unit of the create-rate resource. Fortunately, maintenance of the non-renewable *mount* resource is straightforward since the acquisition and holding of this resource is directly mappable to an active TCP connection. The count of acquired *mount* resource is deducted whenever an unmount request is seen or the associated TCP connection is terminated.

5 Evaluation

In this section, we show that our framework, as instantiated in Section 4.4, can effectively control resource usage and prevent abuse. Specifically, we compare system behaviors with and without our framework under various usage and abuse scenarios for the three protected systems, networking, DNS, and SMB file system.

All experiments were run on the Emulab testbed (http://www.emulab.net). The experimental system comprised 1 LGI controller, 1 security manager, 1 firewall, 1 SMB service (SMB server and reference monitor), 1 DNS service (DNS server and reference monitor), 2 clients (client software such as the SMB client and a User Agent), and 1 external server. The external server represents a node external to the enterprise's computing system is used in experiments exploring the control of Internet bandwidth usage. The firewall is placed at the boundary between the enterprise system and the Internet. Each component ran on a single node, with the nodes partitioned into three subsets, clients, basic, and service. All nodes were 3.00GHz Xeon PC with 2GB of memory, running Linux v2.6.12. The security manager used the Berkeley DB version 4.2.52 to hold the public keys and roles of registered principals. The reference monitors used iptables 1.3.0, Ethereal 0.10, and KeyNote 2.3. The SMB server runs samba-3.0.22 and the DNS server runs Bind-9.4.0.

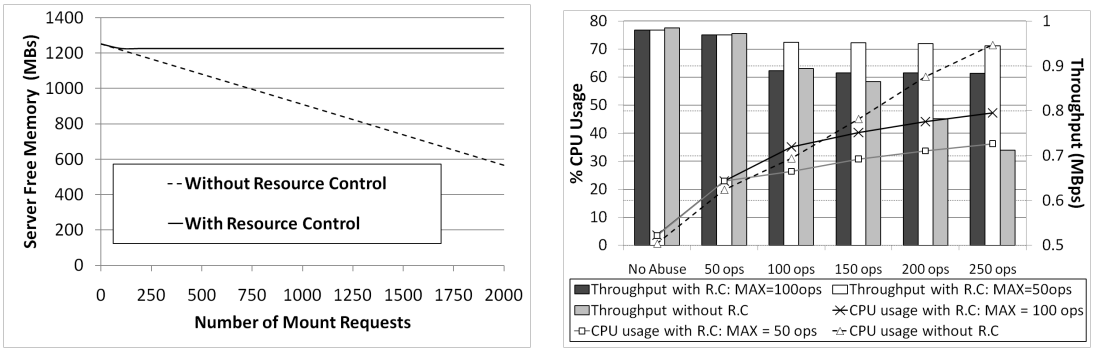
For each of the three protected services, we first present some microbenchmark measurements to show the overheads of our system. In all experiments, we assume that the control agent is already caching the needed user credentials (i.e., roles and resource usage history). The cost of doing this through LGI is $\sim$60ms but should be amortized across many requests. KeyNote requires $\sim$20us and $\sim$40us when evaluating against a root policy and against a root policy plus a delegation, respectively.

5.1 SMB File System

We measured the per-operation response times of the create, read, and write operations with and without our control framework. These times were measured using a small benchmark that creates a new file, writes 1B, closes and re-opens the file, and reads 1B. Recall that when a file is created, it is given a default access control policy. Thus, the time for create includes the time for creating this policy and writing it to the policy database. The times for read and write do not include accesses to the policy database since the needed policy has already been cached. The observed overheads for read and write were negligible (0.2% degradation); this is because our framework merely pass these requests through (as explained above). The overhead for create is somewhat higher at 7% (2.5ms compared to 2.34ms).

Figure 2 shows our framework protecting the SMB server in two different abuse scenarios. In the first (Figure 2(a)), the amount of server free memory, a non-renewable resource, is plotted against the number of mount requests. Observe that without protection, the server memory decreases linearly with the number of mounts; this behavior has been reported as a possible DoS vulnerability [7] and remains in today's Linux SMB server. When protected with our framework using the abstract *mount* resource (Section 4.4), however, a single user cannot abuse the server; observe that the memory usage decreases until 100 mount requests have been processed. Beyond this limit, the client must release a mount resource before another mount request will be accepted, so that the server free memory stays steady.

In the second scenario (Figure 2(b)), a user is abusing the system by rapidly creating files in the same



(a) **(b)**

Figure 2: (a) Server free memory as a function of the number of mount requests. (b) Server CPU utilization and client throughput as functions of an abusive client's request.

directory. When this occurs, the user can drive the server's CPU utilization, a renewable resource, toward saturation even at fairly low rates (e.g., 70% utilization at 250 op/s). The bars in the figure show the throughput of the dbench benchmark (http://dbench.samba.org/), set to generate a load pattern that tries to achieve 1 MB/s read/write throughput to the server, running at a second client. Observe that, without protection, the abusive user can significantly impact other users–the dbench throughput for the second client decreases from 0.985 MBps to 0.712 MBps when the abusive client increase the creation rate from 0 to 250 op/s. While not shown here, the response time degrades badly as well, more than doubling. With protection, however, once the abusive user reaches the maximum allowed rate of create (controlled by the abstract *create-rate* resource), the second user is minimally affected as the abusive user attempts to further increase the rate of creation. The CPU utilization does not completely flatten out after the abusive user has reached the allowed limits because for SMB, our system still needs to forward the requests to the server, but rewriting it in a way that will cause the SMB server to reject the request. This processing is much less expensive than the creates that they replace, however, making it much harder for the abusive client to impact server performance. Furthermore, if the abuse continues, our system can block the client at the networking level, preventing access to the server altogether.

Figure 3: Internet Bandwidth with user credential change

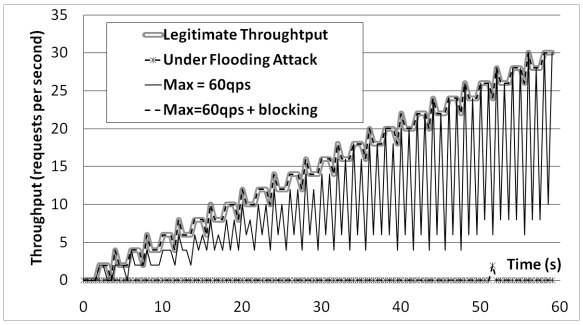


Figure 4: Preventing DNS flooding

5.2 Networking

We measured the round-trip times for ping from our testbed to a machine on the West coast. The average ping time without our control system ranged from 18.1ms for a 1B ping message to 33.2ms for a 40KB ping message. When using our control system, performance degradations were consistently between 0.4

and 0.6% (0.1ms for a 1B message), which is essentially negligible.

Figure 3 shows our bandwidth usage control in effect. Throughout the experiment, the client and external server run iPerf (http://iperf.sourceforge.net), a benchmark for measuring the achievable network throughput between two end points. Initially, the user is a guest and so is subjected to a bandwidth limit of 200KB/s. Observe that iPerf only achieves throughput slightly above this limit; recall that the inaccuracy in control arises from the fact that our system imposes control out of the critical path, using a batch resource accounting method to keep overheads as low as possible. Note that the resource usage constraint in this experiment is set at a much finer granularity than we expect in practical settings to limit the experiment time. For example, at our university, bandwidth constraints are set at daily rates. At these larger granularity settings, our inaccuracies should be negligible.

In the middle of the experiment, the client presents a delegation certificate that allows him to assume a non-guest role, which is allowed to use up to 400KB/s. Our system responds to the change almost immediately, allowing iPerf to ramp up to the new limit in about 1 second.

5.3 DNS Service

Finally, we show the ability of our framework to protect a DNS service. The overheads of our system is somewhat higher ($\sim$14%) for DNS because DNS request processing is relatively inexpensive ($\sim$500us) when the needed data is already cached at the server. Adding 70us does not seem significant given that most clients do not make heavy use of DNS. (We studied our departmental DNS server logs for over one month and the maximum request rate ever observed was 30 requests/s.)

DNS services can be vulnerable to flooding attacks because they must service client requests before the clients are authenticated. Figure 4 shows client throughput in the face of a flooding attack on a DNS server. Without protection, the client's throughput drops to 0 because most requests timeout. With protection, when we are still allowing 60 requests/sec for all users, the client's throughput shows considerable variations but is much higher than 0 (on average, about 75% of the requests succeed). In this case, the server is still affected by the flooding attack because we run the reference monitor on the server itself. Thus, the server host is still busy because requests are only discarded after they have been intercepted and forwarded to the user-level control agent. When our system blocks the attacking client at the networking level–i.e., configures netfilter to discard packets from the offending IP address, then the flooding attack has no noticeable impact on legitimate client throughput.

6 Related Works

Hitherto, most research efforts considering resource usage abuse have focused on protection against DoS attacks. For example, a number of efforts have studied how to protect against network-based DoS attacks [8, 13, 9]. These efforts typically explore methods and mechanisms for detecting and dropping or limiting flow rates of malicious network flows. Our work is very different from these efforts since we are seeking to control the resource usage of authenticated users.

Other efforts [2, 17] at protecting against network-based DoS attacks have leveraged authentication as well, but the attention there has been on identifying legitimate packets in order to drop packets from malicious flows.

In [15] Srivatsa et al. had an interesting discussion about application-level DoS filter, which is difficult to detect at network level. They proposed a client-transparent framework to protect Web Server from application-level DoS attacks by using cookies to store trust tokens, which will be used to filter and rate control requests from clients. In [16], they added the port-hiding enhancement by challenging the client and using Javascript to drive legitimate user's communication into secret server ports. They also used firewalls

to filter out the network-level DoS attack packets that do not contain the secret server ports. The trust token was used to represent the history usage of legitimate users at application-level, similar as our resource counters. But this approach is specific for Web services only, and is not applicable for other services where it is difficult to attach usage information in the client's request. Our framework attempts to be more general, allowing resource usage control to be applied to various different kinds of server applications.

In [12], Qie et al. introduced a toolkit to help developers build DoS-tolerant services. The toolkit comprises libraries and a set of annotation primitives to track and control resource usage inside a server. They introduced renewable and non-renewable resources, which we adopt. Otherwise, their work is complementary to ours. Our framework controls resource usage external to servers but could greatly benefit from servers that track and export information about per-request (or per-client) resource usage.

In [3], Banga et al. introduced the OS-level resource container abstraction, which allows the development of robust servers with fine-grained resource management in server systems. Again, this work is complementary to ours. Also, abstractly, one can view our system as associating a resource container with each user, so that his resource usage within an enterprise system can be controlled.

7 Conclusions

In this paper, we have argued that insider resource abuse is a serious problem that needs to be addressed in a systematic manner. Further, we believe that resource usage control is highly related to access control, and so have proposed an integrated framework that addresses both problems. Our framework centers around an extended RBAC model that considers resource usage histories and rights together with access rights to decide whether client requests should be allowed. Its implementation uses a set of reference monitors, coordinated via LGI, to forward or deny client requests to protected services.

We demonstrate the feasibility and power of our framework by providing a concrete prototype implementation. We used this implementation to enforce an example policy that includes role-based with delegation control over three widely different services. Micro-benchmark measurements show that our system imposes low overheads. Experiments with various abusive scenarios show that our system can effectively protect services against user resource abuse. We thus conclude that our framework can effectively express and enforce sophisticated policies without requiring changes to existing client-server applications.

References

[1] —. Information removed for anonymous review.

[2] D. G. Andersen. Mayday: Distributed Filtering for Internet Services . In *4th Usenix Symposium on Internet Technologies and Systems*, 2003.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3th Symposium on Operating Systems Design and Implementation*, Feb 1999.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704, Internet Engineering Task Force, Sept. 1999.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *ACM SIGCOMM*, 2007.

[6] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Usenix Security*, Aug 2006.

[7] CERT-313836. Samba fails to properly handle multiple share connection requests. http://www.kb.cert.org/vuls/id/313836.

[8] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC (Proposed Standard) 2267, Internet Engineering Task Force, Jan. 1998.

[9] J. Ioannidis and S. M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2002.

[10] N. Minsky and V. Ungureanu. Law-Governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2000.

[11] C. Morariu, M. Feier, and B. Stiller. LINUBIA: A Linux-supported User-Based IP Accounting. In *18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), San Jose, USA, October 2007.*, Oct. 2007.

[12] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[13] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the ACM SIGCOMM Conference*, pages 184–194, Aug. 2000.

[14] SQL Slammer Worm. http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99.

[15] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu. A Middleware System for Protecting Against Application Level Denial of Service Attacks. In *Proceedings of 7th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2006.

[16] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu. Mitigating Application Level Denial of Service Attacks on Web Servers: A Client Transparent Approach. *ACM Transactions on the Web (TWEB)*, (15), Oct 2008.

[17] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting Network Architecture. *IEEE/ACM Transactions on Networking (ToN)*, (6), Dec 2008.

[18] H. Yin and H. Wang. Building an Application-aware IPsec Policy System. In *USENIX Security Symposium*, Aug. 2005.