Design and Integration of ABAC and The GENIAPI AM: Version 1

Ted Faber, USC/ISI

John Wroclawski, USC/ISI

Version 1.2

6 January 2011

Corresponding to GENIAPI version 1.0, ABAC 0.1.2

Table of Contents

1 Introduction....3
2 The GENIAPI and ABAC....4
2.1 Return Values....4
2.2 Self-Validating Identities....6
2.3 Credentials in the New Interfaces....6
3 The GENIAPI Credential System....7
3.1 Indirection....7
3.2 Credentials Model....7
4 The ABAC Implementation....8
4.1 Changes to Operational Logic....9
4.2 Encoding Credentials....9
4.3 RT0 and RT1....9
4.4 Returning results....10
5 Summary....10

1 Introduction

This document describes an initial integration of the Attribute-Based Access Control (ABAC) implementation from USC/ISI and Sparta (version 0.1.2)[1] with the current implementation of the GENIAPI AM (version 1.0)[2] from BBN. The implementation passes the GENIAPI tests included with the system, with minimal modification to the test scaffold to support a change in the unstandardized Clearinghouse interface. This document makes recommendations to the GENIAPI specification based on that implementation and describes the next steps in ABAC design motivated by furthering this kind of access control.

The current GENIAPI implementation provides an aggregate manager that makes resources available within the slice-based facility architecture of GENI. The interface between users (or their agents) and the aggregate manager is well-defined, though interfaces to other parts of the system (slice managers, clearinghouses) are still being standardized.

ABAC[3], which had its formal conceptual design done at Stanford with initial implementations and collaborations at Sparta (then Network Associates), is an access control system based on formal derivation of attributes possessed by principals making requests. These attributes and the rules about reasoning with them allow access control to proceed using the same kind of reasoning that real world decisions encompass. Proof of an attribute may involve multiple rounds of negotiation.

In ABAC, principals assign attributes to other principals. This is analogous to USC asserting that Ted Faber is an employee. Principals can also assert rules for deriving attributes from other attributes: if USC says someone is an employee the campus bookstore says they get a discount. The current ABAC library supports the RT0 logic, which allows multi-layer delegation and intersection of attributes as well as simple derivation.

Incorporating ABAC also separates access policy specification and access validation, provides enhanced logging of access control decisions by both the aggregate manager and the clients, and enables multi-round access control negotiation. ABAC access control requests are either satisfied by a proof of access rights in terms that both sides understand, or a partial proof of access that can be used as a basis for continued negotiations. Complete proofs can be logged as an audit trail.

While the GENIAPI interface leaves room in the specification for much of this integration to proceed, the interface must be tweaked in several important ways to get the best out of the more flexible and powerful ABAC system. Key parts of the interface need to be widened to support ABAC negotiation, support added for self-validating identities, and the user/agent to clearinghouse and aggregate manager to clearinghouse/slice manager interfaces need to be standardized.

In the GENIAPI, the general request/response exchange pattern tends to assume most requests succeed, resulting in very little information from failed requests. Once a request has failed, a user has little idea what changes to the request, if any, may result in success. To implement

ABAC's multi-round negotiation, the system must communicate the successful proof of access or alternatively a partial proof to act as the starting point for the next round. In this release we extended the extensible return values (including XMLRPC Faults) to include proofs and propose more general modifications below.

Central to ABAC's scalability are self-validating identities to which attributes are attached by the various players in the system. Currently the GENIAPI implementation depends on standard X.509 hierarchical assignments of names to authentication keys. We relax this to allow self-signed certificates that make use of Transport Layer Security (TLS) to validate requests while using the underlying authentication key as the identity. Code is included in the initial integration release to do this.

Both the GENIAPI developers and the GENI researchers know that the additional interfaces in the architecture need to be fleshed out. Some of the GENIAPI scaffold code that implements an version of these interfaces uses a credential as a name for objects being manipulated. While we recognize that this code is not a standard, we are concerned that future standards may place a requirement for both naming and controlling object onto credentials. ABAC credentials describing principals and policies, which we believe is significantly more powerful. We are interested in making sure that future specifications allow for credentials that do not name objects.

Working directly with the current credential system highlighted some of the ways we believe it should be improved. Specifically, credentials are bound to specific objects, but not always used with those objects; there is a multi-layered extraneous mapping between credential classes and operation permissions encoded in the implementation; and finally some credential uses provide ad hoc semantics outside the descriptions. We discuss these issues in Section 3.

In Section 4, we discuss details of the ABAC integration. The implementation demonstrates a fairly straightforward encoding of the GENIAPI credential semantics. The section describes the extremely minor changes to the reference AM's operational logic, the mechanism for encoding GENIAPI credentials into ABAC credentials, how this encoding argues for an implementation of more powerful ABAC semantics, and finally how the proofs are encoded.

2 The GENIAPI and ABAC

The primary issues with adding ABAC to the current GENIAPI are the request/response model and limited semantics of the return values, and the imposition of hierarchical names. We also comment on the importance of separating access control attributes and object names in coming development of the specifications.

2.1 Return Values

Though ABAC can require multiple rounds of negotiation and the GENIAPI supports request/response interactions, it is possible to support the extended negotiation of the former in the simple model of the later, if the GENIAPI response to a denied request includes enough information. In practice this means expanding the return values for requests carry more access control data. This allows multi-round negotiation because ABAC captures the entire state of the negotiation in its proofs/partial proofs.

A general ABAC exchange consists of several messages negotiating access to a resource where each principal reveals a little more about their attributes until each knows enough to agree that the access is valid. We show such an exchange in Figure 1.

Simple exchanges between principals who have little information to hide look like conventional request/response exchanges that the GENIAPI supports. Longer exchanges can be carried out in the request/response paradigm as long as successive messages requesting or denying the access provide the other player with new information to advance the negotiation.

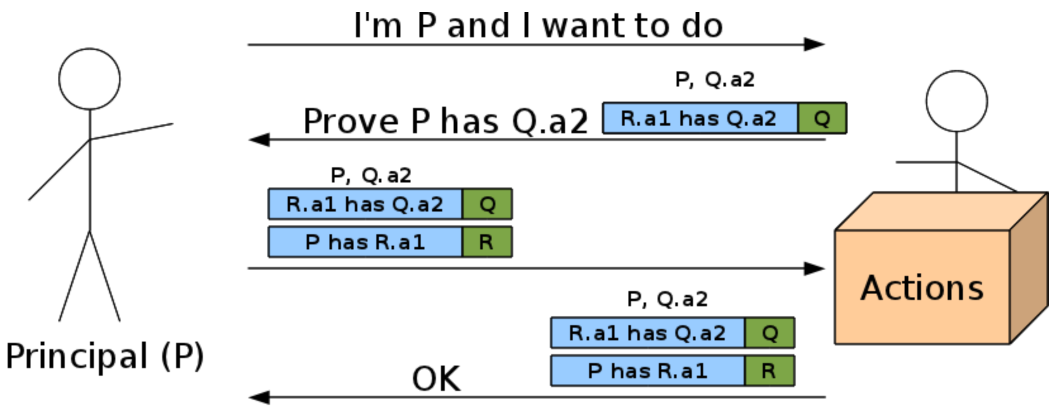

Figure 1: An ABAC exchange

An ABAC partial proof contains the entire state of the negotiation. Passing the partial proof back as a return value in a failed request allows the recipient to extend the proof and try again. The extended proof contains the complete state for the other side to continue the negotiation. The result of a series of denied accesses, extensions, and retries constitutes a negotiation.

In principle, such exchanges are easily integrated with the GENIAPI, but in practice the responses do not carry enough information to support a negotiation. In many cases there is not enough information returned to support detailed auditing information.

Many requests include parameters of sets of attribute value pairs, for example the options field of the `ListResources` operation, which provides an extensible interface. Return values should have a similar structure that includes ABAC partial proofs. Where such fields already exist, current code appends that information.

The current reference AM expresses access control failures as XMLRPC faults, which have limited message capacity. To support ABAC, encode all access denials in the same extensible format, including error code and ABAC partial proofs. Faults are too format constrained for this purpose.

Though this change is motivated by adding ABAC, there are other good reasons to add multi-valued return codes. Some failures can be retried for reasons other than presenting insufficient access controls – e.g., a busy internal resource. Other failures could include hints for a modifying subsequent requests, for example, requesting less constrained resources or lower bandwidth. Providing such iterative interfaces benefits the API in many ways going forward.

2.2 Self-Validating Identities

Our ABAC implementation is designed around principals representing themselves by lightweight, self-validating identifiers. These identifiers contain minimal information beyond being a unique identifier to which attributes can be ascribed. Our implementation represents each principal by a public key. Those public keys are passed around in self-signed X.509 certificates to take advantage of existing SSL code that binds an identity (in a certificate) to an action (XMLRPC or SOAP request).

ABAC credentials are similarly simple: an assertion of a rule or attribute signed by the asserter. Anyone with the asserter's public key – which is basically their identifier – can validate the assertion. The current ABAC implementation uses hashes of the keys as identifiers, so in practice additional keys are passed around.

The ABAC identity system allows principals to simply create an identity and join the GENI ecosystem, though an identity with no principals willing to speak for it is not very useful. It does provide an identity independent of any particular GENI facility (or federation of facilities), which should facilitate interoperation between facilities.

The GENIAPI faces a similar problem in binding identities to requests, but has so far chosen an identity system that relies on trusted identity certifiers that hierarchically delegate the rights to identify principals. Confirming an identity requires the validator to confirm the entire chain of signed identity assertions until a trusted asserter is found. This is a traditional X.509/PKI setup.Their credentials are signed assertions about rights in this namespace, that are standardized within GENI.

The current ABAC library uses these simple, self-validating identities. The initial implementation includes server code that accepts such names and operates on them. That implementation can also be configured to accept only identifiers within the trusted hierarchy.

We strongly believe that self-validating identities provide better scaling properties as well as a clear demarcation between identity and authorization information. It is possible to extend the ABAC library to encode principals as hierarchically assigned names, and therefore interface with the existing GENIAPI identity structure more directly, but tying identity to control framework impedes cross-control-framework operations.

2.3 Credentials in the New Interfaces

The GENIAPI developers agree that an important next step is developing the interfaces between slice manager (or clearinghouse) and aggregate manager and slice manager and user/client. We strongly agree. As those interfaces are created, we believe that the roles of credentials in those interfaces need to be clearly defined.

Though ABAC attributes are generally named mnemonically, all users of ABAC use the same rules and only those rules to prove the validity of accesses. The attribute names convey no implicit meanings nor are inferences made outside the ABAC logic system. This insures that both players in an ABAC negotiation agree on the construction of a proof and believe its validity; third parties can validate that accesses followed policies set out by the players. Other systems include special cases or implicit inferences – for example many Unix systems restrict use of the `su` command to members of the wheel group, enforced by the `su` executable itself.

While the GENIAPI AM stays away from these traps, we notice that some of the scaffold code in the sample clearinghouse does not. When a slice is created at the sample clearinghouse its name is communicated back to the caller in a credential, rather than as a separate result.

We recognize that code as test code and implementation scaffolding. We do, however, use its existence as an excuse to reiterate the point that credentials – ABAC or otherwise – need to be used only for access control, not to transmit other information as a side channel.

3 The GENIAPI Credential System

This section discusses ways that the current GENIAPI credential system is defined or used in ways that weaken it as a basis for access control. While we have a preference for ABAC, these issues need to be addressed in any sound access control system.

3.1 Credentials Model

GENIAPI credentials grant privileges to a principal with respect to a target. The principal and the target have GENI IDs and the privileges are as described in Section 3.2. While this model is attractive in some ways, the current implementation reveals some shortcomings as well. Again, we recognize that this model comes partially from existing testbeds.

The privilege/target system is unwieldy in accessing AM resources on behalf of a user. This is apparent in the `ListResources` call in the reference aggregate manager. The existing code verifies that the credential passed in is valid, but confirms neither the target nor the privileges of the credential. Any user with a credential valid for any purpose can make this call. The relationship between the credential used to gain access and the operation is vague and difficult to explain or audit.

Underlying this vague decision seems to be the idea that currently credentials flow from the clearinghouse/slice manager to the user for use with their slice. Either the clearinghouse must issue credentials with a `ListResources` privilege to every AM it knows about, or there must be a way for an AM to infer the right from some other credential. Currently the second choice is made.

ABAC opens the possibility that parties other than control frameworks or clearinghouses can assign attributes. Such cross-framework attributes can simplify cross-framework access policies because the policies can be expressed independently.

Another similar confusion results in the `CreateSliver` call. In the reference implementation, the only credential that a requester presents to an AM is one granting permissions to manipulate a slice – a resource that the AM may be unaware of and cannot control in any case. Again, an inference is made from the right to manipulate a slice to the right to allocate local resources.

This inference follows naturally from the idea that the clearinghouse is the source of credentials.

Because the clearinghouse sources all credentials, passing users the credentials necessary to allocate resources either must be done explicitly or the AMs must deduce the user's rights. If the clearinghouse passes out explicit subject/object/action credentials allowing allocations at each AM, then clearinghouses and AMs are tightly coupled. Clearinghouses must know the AMs in order to issue credentials and AMs must know clearinghouses to validate them.

If the assignment of allocation rights is implicit to the clearinghouse issuing a slice credential, the clearinghouse is freed of its requirement of knowing all AMs, at the expense of the addition of implicit semantics to a slice credential; it becomes a “slice and resources” credential. AMs that provide different classes of access to resources must set policy on something other than the credentials from the clearinghouse (or interpret the privileges differently). Such access control decisions are difficult to capture and audit.

Because ABAC credentials just describe principals, an AM can both interpret the credentials according to its policy and have the policy codified. If the AM owner wishes to implement the policy that possession of a slice credential implies the right to allocate resources, it can. It can also integrate other ABAC-coded information – e.g., a requester must have a slice credential and have paid a fee to an insurer – into its policy. The policy and evidence that led to an AM's decisions will be available to requesters in the partial proofs displayed during negotiation.

The trust relations are similar in both cases, but the ability to express real world concerns in a formal way is possible because ABAC credentials are less constrained.

If the (subject, object, rights) model is to be kept, we must understand how it works – and what guarantees it makes - in the specific cases of allocating and viewing AM resources.

3.2 Indirection

Current GENI credentials define privileges in terms of groups of operations to which the credential entitles the named principal (on the named object). These privileges map one to many to operations in the system, which also map to various API calls in the GENIAPI.

This is a lot of internal indirection for an access control system. The GENIAPI AM defines a clear set of operations, most of which take a target parameter that is encoded in a GENIAPI credential. It seems unnecessarily confusing to do anything but have the privilege(s) represented in a credential map into the operations in the API.

As mentioned in Section 3.1, AMs wishing to provide multiple classes of service may interpret the privileges differently, which makes the situation more confusing.

We understand that the source of this indirection is in importing credential formats from pre-GENIAPI testbeds. Moving forward, this seems an important area for simplification. Credentials' meaning must be unambiguous.

4 The ABAC Implementation

This section discusses a few key points about how ABAC was rolled into the GENIAPI code in such a way to support the existing credential model. The specific usage of rights to a GENI slice implying rights to local AM resources argues that the implementing a more expressive logic (RT1) than the logic in the current implementation (RT0) would be practically useful.

4.1 Changes to Operational Logic

Most of the ABAC substitution was straightforward. The AM and clearinghouse code each use a `CredentialValidator` class to make access control decisions. We overloaded that class to use ABAC credentials rather than GENI credentials. It also returns proofs or partial proofs to the caller. Those callers only inspected the credentials returned by the original `CredentialValidator` to determine the duration of allocations, so we included identical timing information in the proofs.

In the Clearinghouse we added code to return the slice identifier and the credentials used to access it independently, rather than as one credential. We described this in Section 2.3. The AM was modified to confirm a `list` attribute when making a `ListResources` call.

4.2 Encoding Credentials

Rather than construct a new attribute hierarchy more suited to ABAC, we reflected the target/privilege model into ABAC credentials. In the test scenario, at initialization time the user, clearinghouse, and AM all have an identity created and initial credentials allocated. The only real difference between the standard setup and the ABAC setup is that that ABAC setup includes a credential marking the clearinghouse as being a clearinghouse known to the AM.

Credentials are encoded as strings encoding the privilege and the target. When a clearinghouse issues a credential to the user that allows them to delete slice X that is encoded as the clearinghouse attesting that the user has attribute `delete_X`.

When the user creates a slice, a set of ABAC attributes authorizing actions on the slice are generated and returned. These include the direct privileges taken as input by the `CredentialValidator` class, not the doubly indirect privileges of Section 3.2.

4.3 RT0 and RT1

The difference between RT0 and RT1 is that attributes can be parameterized. That is that instead of using the string `delete_X` as a marker for the right to delete slice X, one can parameterize the delete attribute with an object: `delete(X).` This is useful for two reasons: proofs can constrain parameters (e.g., prove that a principal has an age attribute that has a parameter greater than 35) and that rules can be constructed generically (e.g., a principal could assert that having the attribute write(x) implies the attribute to read(x) for all x).

While much of the time, RT1 parameters are a notational convenience, it is this second case that is needed to directly encode the rule that the right to create a slice is the right to allocate resources. The AM would like to encode the rule that says if a principal that the local AM attests is a clearinghouse attests that a user has the right to create a slice, that user has the right to allocate local resources to that slice. The problem is that the credentials for create all include the slice name; the AM needs to create a rule for deriving a credential (`create_X`) that it does not know exists (clearinghouse says user has `create_X` implies AM says user has `create_X`). The AM needs to know all the values for X.

The real solution to this is an RT1 rule that covers all the possible parameters. To use the current implementation, whenever a request is made for a specific URN, credentials encoding rules for deriving credentials related to that URN are added to the ABAC ruleset. This is an ad hoc mechanism to simulate RT1, but the resulting proofs all are valid. A client cannot distinguish that a rule has been added in this way, and the inferences remain binding on the server.

4.4 Returning results

Returning results are subject to the problems described in Section 2.1. The proof objects returned from the `CredentialVerifier` have a simple XML-derived encoding, and they are tucked into a `geni_proof` field of results that return structures. Other positive return values must be expanded.

Access control failures are returned as XMLRPC faults. While this constrains what can be sent, it was possible to encode the proof objects into the error string. Again, this is not the ideal strategy; access control failures need to be RPC returns with an error code set.

5 Summary

This document summarizes design of and the the lessons learned from the initial integration od ABAC with the GENIAPI. The initial implementation was straightforward, but illuminates some areas for future design in both the GENIAPI itself, specifically in its credential model, and the ABAC library in terms of the semantics it must support for a large scale system.

References

[1]LibABAC Home Page, *http://abac.deterlab.net*, 2010.

[2]GCF Project Home Page, *http://trac.gpolab.bbn.com*, 2010.

[3]Ninghui Li, John C. Mitchell, and William H. Winsborough, “Design of a Role-Based Trust Management System,” *in Proceedings of the 2002 IEEE Symposium on Security and Privacy*, (May, 2002).