Authorization and Trust Structure in GENI: A Perspective on the Role of ABAC

Jeff Chase
Department of Computer Science
Duke University
{chase}@cs.duke.edu

June 17, 2011

Abstract

This note outlines a GENI authorization architecture based on Attribute-Based Access Control (ABAC). GENI Aggregate Managers and other GENI-related servers may use ABAC to represent authorization policies declaratively at deployment time. We discuss how ABAC can meet key requirements for flexible authorization in GENI, including delegation of ownership rights, endorsement of slices, external identity providers, and proxying of control interfaces.

We also leverage ABAC as a tool to represent candidate GENI trust structures declaratively. We contend that ABAC authorization policies are sufficiently powerful to incorporate key trust structures as optional deployment-time choices for each aggregate, rather than viewing them as fundamental architectural choices in the control framework.

This draft supersedes and extends an earlier version of this document dated 5/2/11.

1 Introduction

This note outlines a design to use an authorization logic to represent trust structure and authorization policy in GENI. We focus specifically on Attribute-Based Access Control (ABAC) as a candidate authorization logic. We outline some structures that can be expressed using the RT0 ABAC implementation in *libabac*.

The GENI architecture centers on a common Aggregate Manager (AM) interface for experiment control tools. This note focuses on authorization and trust management for AMs, but the principles apply to other servers in GENI as well. Most details of the API are not important for our purposes. We assume as a starting point that every request at a server is issued by an authenticated *subject* (such as a user-experimenter) and operates on a target *object* (such as a slice). Before executing each request, the server verifies that the request is authorized by executing a check (sometimes called a guard or reference monitor). This check evaluates a *credential set* associated with the subject with respect to an *authorization policy* associated with the target object.

The power of ABAC comes from its ability to represent declaratively an authorization policy that reasons from the credential set. An inference engine attempts to "prove" that the subject has the required permission, using inference rules built into ABAC together with rules and facts encoded in the subject's credentials and in the target object's authorization policy. These credentials may also include attributes of the subject asserted by identity providers representing participating institutions and companies (see Section 6).

The ABAC logic can also represent general control framework trust structures. Trust structure is a key area in which designs for GENI control frameworks differ. GENI is conceived as a federation of independent aggregates, and the trust structure determines which entities are empowered to play a role in cross-aggregate

policy decisions. For example, aggregates may delegate various aspects of their local policy to various trust anchors or intermediaries, e.g., to manage subject identities and accounts, monitor and coordinate resource allocation across many aggregates, advertise aggregates to potential experimenters through a common clearinghouse, or provide a GENI-wide kill switch. Users may also rely on various intermediaries to coordinate scheduling and interconnection of resources or determine which aggregates are "safe" to use.

Our goal is to factor these structures out of the control frameworks and represent them as deployment choices within a common control architecture. All of these functions reduce to authorization decisions about whose assertions to believe, whose commands to accept, or what sensitive information to reveal and to whom. An authorization logic such as ABAC offers a powerful formalism for aggregates or other servers to define their authorization policies, and/or delegate any part of their policy to selected intermediaries as they see fit. Owners and operators may specify these choices declaratively as sets of ABAC rules, including selection of trust anchors for identity management, slice endorsement, external operational control, resource brokering, and so on.

This note shows how the various trust structures proposed for GENI can be represented in a library of off-the-shelf ABAC templates. AMs and other servers may draw from these templates to construct their authorization policies at deployment time. In such a system the federated trust structure emerges from the deployment choices made by the AMs and other servers in the federation, and may evolve over time.

2 ABAC Overview

The principals in ABAC are "entities", each possessing an asymmetric keypair and an X.509 identity certificate. Entities and objects are named by unique IDs. For example, an ID could be a GUID, a GUID hash, or a URN. An entity name includes the entity's public key. It is thus possible to authenticate an entity. Every subject is an entity and any entity can be a subject. Every server is an entity and any entity can be a server.

The essence of ABAC is that entities have attributes. Possession of an attribute says the entity is a member of a set associated with some role. We use the terms *attribute* and *role* interchangeably. In general, an ABAC role has a name, but no accompanying value: a given entity either has the role or it does not. More precisely, an entity A either knows that another entity E has the role R or it does not. We can write this knowledge in the form of a basic ABAC statement:

$$A.R \leftarrow E \tag{1}$$

Every ABAC statement has a left-hand side with a role qualified by some entity, and a right-hand side with an entity or other roles. The notation is best understood as a set logic, in which both sides designate sets, and the arrow indicates that the set on the left-hand side contains any entity matching the right-hand side. The statement above can be understood as: "Entity A says that the entity E is a member of the set possessing the role R."

2.1 Inferring the Authorization Condition

Roles capture authorization according to a simple rule: each operation or command C on an object O at an aggregate (or other server) A has a corresponding role C_O. A subject S may perform the operation if and only if S is a member of the set of subjects known to A to possess the role C_O. We can represent this authorization condition as an ABAC statement:

$$A.C_O \leftarrow S \tag{2}$$

How does a server determine if $A.C_O \leftarrow S$? Each entity maintains sets of ABAC statements that say what that entity believes about the subjects and roles it knows about. As in any logic, some statements in ABAC

represent simple facts, and others represent inference rules that can be used to infer new facts from facts already known. ABAC defines several types of inference rules encoded as ABAC statements. We do not define these rules here, although there are many examples in the discussion below. ABAC implementations provide each entity with an *inference engine* that can prove or infer a target query from a set of statements (facts and rules); this set is called the *context* for the authorization decision. The context is the union of a *credential set* provided by S and A's *authorization policy* for the object O. If the inference succeeds, then A permits the requested operation. If A cannot make the inference, then A denies the operation.

2.2 ABAC as a Federated Web of Roles

As explained in the ABAC papers, ABAC is a system for federated authorization that provides, in essence, a "web of roles": any entity may publish or assert whatever it wants within its own name space of attributes (roles), and other entities choose for themselves whether to listen or care.

Each ABAC attribute (role) is controlled by some entity, which is an *attribute root* for that attribute. Every entity is an attribute root for its own name space of attributes: each attribute name is qualified by the identity of its attribute root. For example, the ABAC statement $A.R \leftarrow E$ can be understood to say that the entity E is a member of the set whose members possess the role or attribute R in the namespace of attributes rooted in aggregate A.

Each entity controls the roles in its own namespace: an entity may issue any assertions about which subjects possess the roles that it controls. Those assertions may be rules that infer possession of a role from other roles in other namespaces. Entities make these assertions as ABAC statements signed with their private keys. A signed ABAC statement is a *credential*. In the *libabac* implementation, credentials are encoded as X.509 attribute certificates.

Each entity controls only the roles in its own namespace: an entity may not make assertions about which subjects possess roles outside of its namespace. If a credential is signed by A, then it must encode a fact or rule about a role in A's namespace, i.e., the left-hand side of the rule must be some role $A.R$. If a credential encodes a statement with $A.R$ on the left-hand side, and it is not signed with A's private key, then it is not a valid ABAC credential. Invalid credentials are rejected by the ABAC inference engine.

Any entity may choose whether or not to consider a valid credential based on its issuer and the roles that it asserts. The rules that determine which sources and roles to consider are specified by the entity's authorization policies for its objects. These policies are encoded as ABAC statements that infer possession of roles in the entity's own namespace based on other roles in other namespaces rooted in other entities.

2.3 ABAC as a Belief Logic for Global Roles

In this discussion we assume that GENI participants agree by convention on common role names, i.e., a common global namespace of roles. Then we can treat the attribute root qualifier on each attribute as a representation of who says or believes that an entity possesses a global role. For example, the ABAC statement $A.R \leftarrow E$ can now be understood as "A believes or accepts that E has the global attribute (or role) R".

This view of the ABAC authorization logic is equivalent to the "web of roles" view. It is only a change in perspective, and not a change to ABAC itself. The common attribute name space is a convention that simplifies the discussion and implementation.

This document suggests various attributes and exemplary role names and proposes various uses and relationships for the GENI trust fabric. This document can be the basis for a proposal for GENI authorization roles, but it is not intended to serve as a proposal. Its purpose is to show how ABAC can be used to specify roles and inferences that are sufficiently rich to specify the GENI trust fabric and authorization policies declaratively as ABAC statements, rather than baking them into the control frameworks.

We consider several classes of global GENI roles:

- **Operator** roles grant a subject a right to install authorization policies on a given server, or to issue other administrative commands on that entity.
- **Trust anchor** roles represent the status of an entity as a trust anchor or intermediary for cross-aggregate policy, e.g., GENI Portal, GENI Management and Operations Center (GMOC), Slice Authority, identity provider, etc. Trust anchor roles are specified directly by operators, or inferred by statements from other trust anchors installed by operators. For example, each aggregate has policies that define which entities it trusts to act as a Slice Authority, but it may delegate endorsement of authorized Slice Authorities to some other trust anchor, such as a central GENI registry.
- **Identity** roles specify attributes of an identity in the real world. E.g., "this user is a student at Duke and has the unique identifier 0x8472". The sources of identity attributes are trust anchors that maintain user accounts, called *Identity Providers* (IdPs). Each aggregate specifies in its authorization policies which entities it trusts to act as IdPs, and which identity attributes to consider as a basis for authorization decisions. [One challenge is how to represent and use attributes that are properties with both a name and a value; see the RT1-lite discussion below.]
- **Capability** roles represent a subject's possession of a right to issue specific commands on a specific object, e.g., an experimenter's right to create slivers in a given global slice. Capability roles may be issued by some trust anchor, such as a Slice Authority. The holder of a capability role may delegate the role to some other entity under certain conditions, e.g., to grant another experimenter the right to operate on my slice.

2.4 Parameterized Roles and Object Attributes: RT1-Lite and RT2-Lite

ABAC defines a family of "RT" authorization logics of varying degrees of complexity. The implementation of ABAC that we propose to use is called *libabac*, and it is an implementation of the simplest RT logic called RT0. Our proposal requires certain limited features of the more complex RT logics. Our approach is to obtain these features without modifying the RT0 *libabac* implementation or requiring a more powerful (and complex) inference engine. To do this, we use two techniques that turn out to be quite powerful in combination:

- **Syntactic conventions for roles.** We define some simple syntactic conventions for role names that encode certain qualifiers on the role, e.g., the role is delegatable, or the role is associated with a specific object.
- **Templated authorization policies.** The authorization policy for each object is generated from a template at the time the object is created or—in the case of a global object such as a slice—first registered or made known at a server. Each server has a parameterized template for each object type that it supports. The server includes code to customize the template for each specific object in a type-specific fashion, e.g., by filling in parameters, transforming role names in the rules according to the syntactic conventions, filling in role names from other sources of information about the object, or enabling or disabling specific rules from the template based on attributes of the object.

We briefly outline two examples, which we call *RT1-lite* and *RT2-lite*.

2.4.1 RT1-Lite

RT1-lite enables a limited form of *parameterized roles*. RT1-lite is a simple form of RT1, which is an extension of the base RT0 ABAC logic. RT1 allows general parameterized attributes, but is computationally more complex than RT0. In contrast, RT1-lite requires no changes to RT0 ABAC implementations, such as *libabac*.

For RT1-lite, the attribute name is derived by appending a separator character and a parameter value to a base attribute name. In principle, RT1-lite enables ABAC credentials to carry a limited kind of property list, with property-value pairs. The concept is similar to URL query strings. The format is a convention among credential issuers and servers. An entity that knows the format can extract property-value pairs from the ABAC credentials.

A key example of a parameterized role is an *object-specific role*. An example of an object-specific role is the role C_o needed to perform command C on object O. Object-specific roles may be qualified by a single parameter, which is an object ID. These parameters act as a filter on the credentials and attributes considered by the inference engine for a given object O: if any role R appearing in a credential is qualified with an object O, then a policy for object O' considers that credential only if $O == O'$. This mechanism is implemented outside of *libabac* with logic in the server to customize parameterized rules when the object's authorization policy is generated from a template. Specifically, the server code to generate the authorization policy for O fills in the parameter value O for any object-specific roles appearing in the policy template.

2.4.2 RT2-Lite

We also find it useful to consider object properties in an object's authorization policy. One example discussed below is that a slice may have a property indicating that the slice is a GENI slice. If this property is present on the slice, then a special inference rule is enabled that authorizes any entity with the GMOC role to operate on the slice.

RT2-lite enables a limited form of *object attributes*. RT2-lite is a simple form of RT2, which is an extension of the base RT0 ABAC logic. RT2 allows general object attributes, but is computationally more complex than RT0. In contrast, RT2-lite requires no changes to RT0 ABAC implementations, such as *libabac*. For RT2-lite, we presume that any significant object properties are present and known at the time the server generates the authorization policy for an object, and that the code can read these properties directly from the data structures representing the object. The server code that generates the authorization policy examines the object properties to determine which templated rules to include in the object's authorization policy. It could also fill in parameters in the templated rules with properties from the object.

3 Capabilities in ABAC

Slice Federation Architecture (SFA) frameworks use an approach to authorization that combines a capability mechanism with a hierarchical naming registry. This section shows that ABAC is sufficiently powerful to subsume this capability mechanism. Other parts of this document contend that ABAC serves other objectives that are not served by the capability model: in particular, ABAC enables us to represent a range of trust structures declaratively (Section 4) and integrate external identity providers (Section 6).

A capability system is a special case of an ABAC framework in which roles represent specific privileges for specific objects. As noted above, a *capability attribute* is one kind of attribute supported by ABAC. In fact, we presume that a capability attribute exists for each possible command for each object. As in SFA, the key step in the authorization scheme is a check to ensure that the a server A believes that a subject S attempting to issue a command C on an object O possesses the required capability: $A.C_O \leftarrow S$. What is different is that ABAC permits the capability to be inferred from other attributes, as an alternative to representing it directly in a credential. But ABAC also supports a capability authorization model, in which credentials specify rights and privileges directly by means of capability attributes.

Capabilities minimize the need for an authorization policy at the object or its server: since a capability credential represents directly the privileges that it enables, any entity may determine those privileges by inspecting that credential, after determining that the credential is valid. The authorization policy is implicit in the capability model. To use capabilities in ABAC we must include specific rules in the ABAC authorization policies to encode the semantics of the capability model. This section gives ABAC rules to encode

capability semantics declaratively.

The capability authorization model requires four key properties:

1. **Per-object rights**. Each capability attribute is associated with a specific object. This property is met using the "RT1-lite" convention described above.
2. **Owner control**. The entity that creates an object wields all rights to operate on the object. Section 5 addresses this property.
3. **Delegation**. A mechanism exists for a holder of a capability to pass the capability to another entity at its discretion.
4. **Confinement**. A mechanism exists for the holder of a capability to block a delegate from further delegating the capability to a third party. Lack of confinement is widely cited as a flaw in classical capability models: "the friend of my friend is my friend".

If a subject presents a capability credential, the function of the ABAC inference engine is to ensure that the issuer was authorized to issue the credential. To do this, the inference engine must use ABAC rules in the authorization policy to validate each step in the chain of delegations back to the original owner/creator of the object. It must also validate the trust anchor that conferred the capability on the original owner, as discussed in Section 4.1 and Section 5.

3.1 Capability Delegation

Suppose that subject S_1 possesses a capability attribute C_O, and it wishes to delegate the capability to a subject S_2. To support delegation, we can introduce a *delegatable* form of each capability. For example, we could, by convention, introduce a new form of every capability attribute that adds a superscript $*$, or any other syntactic convention you prefer, to indicate that the capability can be delegated.

Suppose then that S_1 possesses a delegatable form of the capability attribute command C on object O: C_o^*. To delegate the capability to subject S_2, all that is needed is for S_1 to issue a credential for use by S_2:

$$S_1.C_o^* \leftarrow S_2 \tag{3}$$

In words, the credential contains a statement signed by S_1 that says: "S_1 believes that S_2 has the capability attribute C_o^*." Suppose that S_2 presents this credential to a server A in an attempt to issue the command C on object O. Before granting access, A must first convince itself that it too believes the statement in the credential:

$$A.C_o^* \leftarrow S_2 \tag{4}$$

We support validation of capability delegations by adding an ABAC type 3 rule in the authorization policy for O:

$$A.C_o^* \leftarrow (A.C_o^*).C_o^* \tag{5}$$

That says: "if someone who A believes wields C_o^* says that some other entity E also wields C_o^*, then A believes that E wields capability C_o^*". Of course, the inference engine may apply this rule transitively to validate a chain of transitive delegations. For example, to infer that the delegation from S_1 to S_2 is valid, it must also believe that S_1 possesses C_o^*. That inference may require a chain another use of the delegation rule, and so on. The delegation chain is grounded in some trust anchor that binds the object and its capability rights to some initial creator or owner, as described in Section 5.

In addition, we add a second rule to the authorization policy to allow A to infer that any entity E that possesses C_o^* also possesses the base capability C_O, which is necessary to actually perform the command:

$$A.C_o \leftarrow A.C_o^* \tag{6}$$

3.2 Confinement

It should be obvious from the discussion above how to support confinement. Suppose as in the previous example that the subject S_1 possesses a delegatable capability attribute C_o^*, and it wishes to delegate the capability to a subject S_2, but this time it also wishes prevent S_2 from delegating the capability further. All that is needed is for S_1 to issue a credential for use by S_2:

$$S_1.C_o \leftarrow S_2 \tag{7}$$

This credential is similar to the original example, except that S_1 has conferred the non-delegatable form of the attribute. There is nothing to stop S_2 from issuing a credential to delegate the attribute C_O to another entity S_3. But if S_2 does so, S_3 cannot use the capability. Specifically, S_3 cannot convince the server A to believe that S_3 actually possesses the attribute C_O, because the type-3 rule for capability delegation applies to attribute C_o^*, and not to C_O. Moreover, if S_2 attempts to delegate C_o^*, the validation inference will fail because S_2 does not itself possess C_o^*.

However, A's authorization policy needs a second type-3 rule to validate the confined (terminal) delegation:

$$A.C_o \leftarrow (A.C_o^*).C_O \tag{8}$$

That says: "if someone who A believes wields C_o^* says that some other entity E wields C_O, then A believes that E wields capability C_O".

As an aside, we note that ProtoGENI represents the delegatable and non-delegatable forms of each attribute by including a bit in the underlying certificate format that says whether an attribute is delegatable or not. It would be possible to use a similar approach to build the capability model directly into ABAC: if every attribute is qualified with a bit indicating whether it is delegatable or not (off by default), then the inference rules for capability delegation could be hardwired into the ABAC inference engine, making it unnecessary to encode them explicitly in the authorization policies. But it is easy to support the capability model outside of ABAC using the two-step process outlined here: (1) establish a convention for syntactic transformation of delegatable attribute names, and (2) preinstall the capability delegation rules in templates that form the authorization policy for every object.

4 Representing Trust Structures

This section summarizes ABAC encodings for some specific trust structures that have been proposed for GENI. Note that these structures are represented declaratively within the ABAC policies: they do not require changes to ABAC, and do not need to be baked into the control framework. Any aggregate might choose to require any or all of these structures, or not.

First, we consider some intermediate trust anchors. A trust anchor is just an entity E that is accepted by the server (an aggregate A) as possessing a specific attribute R that empowers E to make certain assertions without proof, i.e., A believes those assertions if E makes them. One example of an intermediate trust anchor is an *identity provider* that is trusted to assert attributes of subjects without proof.

The operator of a server may install rules to configure the server with a desired set of trust anchors. For example, the server's operator may install an ABAC rule directly asserting that it includes E in the set possessing attribute R: $A.R \leftarrow E$.

We refer to the trust anchors as *intermediate* because they might obtain the role R indirectly. They are "anchors" only for the purposes of this discussion because we do not specify or limit how they were authorized for the role R. But in practice, arbitrary ABAC machinery can be brought to bear to determine A's view of the set of entities possessing role R. For example, the server may include other rules delegating the power to specify the attribute R to some other trust root.

To illustrate, consider a hypothetical global GENI registry G. If A wishes to trust G to specify which entities E possess attribute R, it can add to its policy an ABAC type 2 inclusion rule:

$$A.R \leftarrow G.R \tag{9}$$

That rule says: "if G asserts that some entity has role R, then A believes it." Now, if G issues an assertion $G.R \leftarrow E$, then A can infer that $A.R \leftarrow E$. For example, some set of aggregates might trust a global GENI registry to designate a set of trustworthy identity providers in this fashion (similar to *incommon.org*; see Section 6). The registry might rely on local operator actions to approve identity providers and generate certificates; e.g., registry operators click on a form to approve the providers, run administrative ABAC commands to generate credentials, or edit a file listing the keys of approved providers.

Of course, designation of trust anchors is not limited to this example either. We could encode ABAC rules that say, for example, "sa.cs.duke.edu is a slice authority for protogeni if and only if somebody who works for Rob and somebody who works for Chip both say it's OK, and it is endorsed by a faculty member at a university with membership in incommon.org".

This section outlines some intermediate trust anchor roles for GENI. Each role is represented by a specific attribute R. A server A accepts an entity E as an intermediate trust anchor if it believes that entity has the role R: $A.R \leftarrow E$.

4.1 Slice Authority

Slice Federation Architecture (SFA) frameworks define entities called Slice Authorities to control the creation of slices. The exact functions of a Slice Authority are not fully specified. The SFA 1.0 and SFA 2.0 documents say little more than that a Slice Authority "names and registers the slices and enables users to access and control their slices." What is clear is that the SA is conceived as a trust anchor: the SA is "a principal that takes responsibility for the behavior of the slice" and "issues a credential" that "vouches for the slice". And aggregates "have the right to select which SAs are empowered to create slices on their resources".

For our purposes, an SA is any entity that is trusted by AMs to issue *slice credentials*, if the AM's authorization policy requires those credentials. We presume that a slice credential is an ABAC statement signed by the SA that identifies some entity as an owner of a global slice object, as described below. We presume that by issuing such a credential, the SA implicitly endorses the slice. *Note*: in e-mail discussions in May 2011, it became clear that the ProtoGENI and Deter teams want an SA to be able issue slice credentials that are valid for some aggregates, but not others. We discuss this variant at the end of this subsection.

An entity E is recognized as a Slice Authority at aggregate A if $A.SliceAuthority \leftarrow E$. Again, we do not care how A infers that E is a valid SliceAuthority, as long as A believes it. For example, A could delegate designation of Slice Authorities to a hypothetical trusted GENI registry G simply by including an ABAC type 2 rule $A.SliceAuthority \leftarrow G.SliceAuthority$ in its authorization policies. As noted above, that is only one simple example of a useful trust structure.

The key authorization rule affected by SliceAuthority is: For subject S to create (register) a slice O at aggregate A, the subject S must present a slice credential showing that it is endorsed as an owner of the

slice (transitively) by a Slice Authority recognized by A. We propose an attribute called *Owner*, which is parameterized by a slice object ID using the RT1-lite technique described in Section 2.4: $Owner(O)$. For example, suppose the engine can infer:

$$\begin{array}{l} A.SliceAuthority \leftarrow E \\ E.Owner(O) \leftarrow S \end{array} \tag{10}$$

Then A infers and believes that S is an owner of O:

$$A.Owner(O) \leftarrow S \tag{11}$$

To enable this inference, the authorization policy for O includes a type 3 attribute-based delegation rule, to allow an accepted Slice Authority E to say who is the owner of any slice that it endorses:

$$A.Owner(O) \leftarrow (A.SliceAuthority).Owner(O) \tag{12}$$

"If an entity that is accepted by A as a Slice Authority says that someone is a *Owner* of some object O, then A believes it." This rule is a parameterized rule in the authorization policy tempate for slice objects. When the slice is made known to an aggregate$A$, the server code generates the authorization policy for O from the template, and fills in the videntity of O. This is a use of the RT1-lite technique described in Section 2.4.

The astute reader will notice that *Owner* is a capability attribute, and that the entire discussion in Section 3 applies to *Owner* and related ownership attributes. It follows that "ownership" could also be represented by a set of capability attributes, each conferring a subset of ownership rights. This alternative makes it possible to delegate some ownership rights independently of others. This is called *refinement* in classical capability models. We use *Owner* as a shorthand for any capability on the object. [Perhaps this section should be edited to use C_O.]

Note: in e-mail discussions in May 2011, it became clear that the ProtoGENI and Deter teams want an SA to be able issue slice credentials that are valid for some aggregates, but not others. In principle this is easy to do. One way is to define additional attributes representing validity for specific AMs or arbitrary groups of AMs. The authorization policy for slice registration on an aggregate must check that the SA has issued a suitable credential for the slice. However, there are several wrinkles. First, it should be kept in mind that the SA has only the power delegated to it by the aggregates: any such approach assumes that the aggregates are faithful to any restrictions imposed by the SA. The aggregates hold the power, and they enforce any SA restrictions voluntarily. It may even be impossible for an SA to know if the aggregates are faithful. Second, the most direct approach is for the slice credential to bind the attribute or role to the slice itself, and not to the owner of the slice. The authorization policy must require that the slice has a suitable attribute, and also that the subject has suitable rights to register and operate on the slice. This is unusual in that the slice credential is issued to an object, rather than to a subject with a public key and an X.509 identity certificate, as assumed by *libabac*. Implementing this might require some backbending with *libabac*, if it is truly required. This document does not discuss this issue further.

Note: June 2011 discussions with Aaron Falk suggest that GPO plans to integrate Slice Authority functions into a GENI Clearinghouse, i.e, the Clearinghouse will act as a Slice Authority, but we can still use the term Slice Authority to describe that role of the Clearinghouse, and the *SliceAuthority* attribute to formalize it.

4.2 Slice Tracker and Proxying

It has been proposed that a GENI aggregate should accept CreateSliver only if it can be verified that some trusted entity is tracking sliver requests on the slice, and thus has a global view of the slice's activity across aggregates. One way to achieve this goal might be to have the aggregate notify a designated tracker about

all slivers that the aggregate creates for the slice. For example, the designated tracker might be a Slice Authority that endorsed the slice. One drawback of that approach is that it complicates the aggregates.

Another approach is for the authorization policy to require experimenter-users to proxy CreateSliver requests through a SliceTracker trusted by the aggregate, which can examine the user's requests before passing them through to the AM. In other words, the authorization policy has the restriction that only a recognized SliceTracker can be the subject of a CreateSliver operation (or some other command C on an object O, requiring C_O). Note that since the AM sends the response to the subject, the SliceTracker receives all CreateSliver responses and knows all of the slivers in the slice.

How can the trust structure support proxying of requests for this scenario and others? If user U issues a request through a proxy server S to an aggregate A, then the authorization policy at A must authorize S rather than U as the subject of the operation. Let us suppose that A can infer that U is authorized to perform the operation. Then it is only necessary for S to present a credential showing that U has endorsed S to issue operations on its behalf. We can introduce an attribute called "speaks for" for this purpose. The user U asserts that the tracker speaks for the user, or speaks for the user with respect to a specific slice O.

For example, suppose A knows that U has a capability C_O to perform the command (per the inferences above) and A accepts the subject S of the CreateSliver operation as a SliceTracker.

$$\begin{array}{l} A.C_O \leftarrow U \\ A.SliceTracker \leftarrow S \end{array} \tag{13}$$

Then the policy can infer that S is authorized to operate on the object O if it presents a credential from U showing that the subject "speaks for" the entity U:

$$U.SpeaksFor \leftarrow S \tag{14}$$

The *SpeaksFor* attribute is a general construct to allow a user to issue operations through a proxy intermediary that is trusted by the user. The rule in the authorization policy to support this inference is:

$$A.C_O \leftarrow (A.C_O).SpeaksFor \tag{15}$$

That says: "if some entity that A believes has capability C_O says that some other entity speaks for it, then A believes that other entity has capability C_O".

[*Questions.* Why is this better than having the user delegate its capabilities to the proxy? Because the user is accountable for actions taken by a proxy speaking for the user. Also, the user might not have a delegatable capability. What happens if we refine SpeaksFor so that the proxy speaks for the user with respect to specific objects? That is a simple extension. What is the computational cost, given that the inference engine may need to enumerate all users that the proxy speaks for, searching for the one that has the necessary capability? The proxy can specify whom it is speaking for, and the server can form the context for the authorization decision to include only the credentials associated with the intended subject.]

Note: in e-mail discussions in May 2011, the informal security working group appears to have generally agreed that SliceTracker functionality is not required for GENI. But I'm leaving this section here because the *SpeaksFor* attribute for proxying is relevant in other scenarios as well. Also, June 2011 discussions with Aaron Falk suggest that GPO plans to integrate Slice Tracker functions into a GENI Clearinghouse, i.e, the Clearinghouse will act as a Slice Tracker, but we can still use the term Slice Tracker to describe that role of the Clearinghouse, and the *SliceTracker* attribute to formalize it.

4.3 GMOC

It has been proposed that a designated operational entity called GMOC should have the power to stop an experiment by disabling slivers in a GENI slice. For example, suppose A believes that the object O is a GENI slice or sliver. We could write this as $A.GENI \leftarrow O$, making use of a conceptual "RT2-lite" extension allowing objects to have properties. We do not specify how A comes to know that O is a GENI slice, but it might record this property if the Slice Authority for the slice is recognized as a GENI Slice Authority. In this case, A could record a type-2 inclusion rule in the authorization policy that is active for slices or slivers with the GENI property:

$$A.Disable(O) \leftarrow A.GMOC \tag{16}$$

Then, if A believes the subject S for a disable operation is an authorized GMOC entity:

$$A.GMOC \leftarrow S \tag{17}$$

then its policy will allow S to disable the slice or sliver even if S does not have a credential showing ownership or an owner-delegated capability to operate on the object.

4.4 Identity Provider

It has been proposed that GENI aggregates should allow access only for subjects whose real-world identities and roles are known. Even if some aggregates allow anonymous access, it is clear that certain operations or allocations will require real-world roles.

An *Identity Provider* (IdP) is any entity that is empowered to assert without proof that a subject S has certain attributes that represent the binding of S to an identity and/or roles in the real world. An aggregate or other server A may accept these attributes and use them as input for its ABAC authorization policies.

A server's ABAC authorization policies can specify which entities to accept as having an IdP role, or delegate that choice to other intermediate trust anchors in the trust structure. The use of ABAC to incorporate IdPs into the trust structure is the cornerstone of a proposal to factor identity management out of GENI, and delegate it to Shibboleth or some other external Single Sign On (SSO) framework. This goal is important because it will allow GENI to benefit from the work that many others have done to address the many deep problems of federated identity, which is required for GENI.

Since the GENI authorization framework will be based on entities acting with public keys (e.g., ABAC, SFA capabilities, ORCA tickets), GENI has a further requirement that IdPs must issue public-key credentials, e.g., ABAC credentials. Since Shibboleth/SAML IdPs bind attributes to an HTTP session and not to a public key, these IdPs cannot be used directly within GENI. One way to use attributes from these IdPs within GENI is to establish one or more *Identity Portals* that bridge federated identity providers into a GENI federation by issuing credentials with attributes harvested from an HTTP session. In essence, the Identity Portal acts as a proxy for external IdP, and is trusted as an IdP by aggregates and other GENI services. We use the terms *identity provider* and *IdP* to refer to any identity provider, without regard for whether that IdP is an identity portal that proxies external IdPs, or whether it speaks any specific identity protocols such as Shibboleth/SAML. Section 6 discusses identity portals and issues raised by the portal approach in more detail.

The key point here is that servers may use ABAC authorization policies that examine attributes asserted by external IdPs, and use those attributes as a basis for authorization decisions.

4.5 Slice Manager

In ORCA, a service called *Slice Manager* (SM) acts as an agent for a user-experimenter. SM provides a Web GUI for monitoring slices and supports user-selectable or user-defined slice controllers. The controllers can adapt a slice automatically, or expose alternative interfaces to create, monitor, and adapt sices. For example, the SM runs a standard GENIAPI controller that exports a front-end interface for GENI experiment control tools. The SM core also orchestrates stitching by means of label propagation along a dependency graph.

A user-experimenter can instantiate a private SM that speaks with its private key. In this case there are no special authorization requirements to support SM. In this scenario, the SM is in essence an experiment control tool running with the user's identity.

In another scenario, the SM service might be hosted by some other party, and speak with its own keypair. In this case, SM amounts to a proxy for some or all of the user's requests. The user may issue an ABAC credential to the proxy asserting that the proxy $SpeaksFor$ the user. This is similar to the example discussed in Section 4.2.

4.6 Resource Allocation

One issue that needs more discussion is the relationship of ABAC to resource allocation.

It is clear that resource allocation may involve an authorization decision, and this document proposes that an authorization logic such as ABAC serve as the basis for all authorization. So resource allocation policy may consider attributes of the subject. The ORCA team has demonstrated some simple examples at the summer 2010 GEC. The full power of ABAC authorization policies and inference can be brought to bear on this problem.

It is also clear that resource allocation may involve cross-aggregate or federation-wide policies. For example, we might want some form of quotas or fair-sharing policies or even market-based mechanisms that apply across the entire GENI federation. Although controversial, this has been a mandated requirement for the GENI project since its inception. ABAC provides a powerful formalism to define trust structures for federated policy, such as delegation of an aggregate's policy decisions to other entities trusted by multiple aggregates, such as identity providers, slice authorities, etc. It is natural to imagine that we might use these mechanisms for federated resource allocation.

Even so, I have taken the position that "resource allocation policy is more than authorization". ABAC is useful but it is not sufficient. Additional mechanisms and structures are needed. To be sure, federated resource allocation requires pluggable policies and delegation, and may incorporate authorization decisions using ABAC. But resource allocation policies may also consider additional dynamic information, such as resource status or resource usage by the subject. It is an open question how to represent or reason about this dynamic information using ABAC. It is an open question how to represent resource allocation policies declaratively.

The ORCA team has given a great deal of attention to this problem. ORCA is based on SHARP [1], which introduces a ticket representation and ticket delegation mechanisms to support brokered resource allocation. An entity called a *broker* implements federation-wide policy. The broker issues signed credentials called *tickets* to endorse/authorize resource requests for a slice. A ticket may be presented to an aggregate to prove to the aggregate that the broker, which is trusted by the aggregate, has authorized the request. The ORCA lease manager [4] combines the SHARP brokering architecture with pluggable policies for federated resource allocation [2]. This architecture has been used as the basis for various dynamic resource management policies including market-based allocation with a federation-wide virtual currency [3].

SHARP and ABAC are roughly contemporaneous and are based on similar foundations: PKI with signed security assertions, and hierarchical delegation validated by an inference engine that walks chains of delegations back to their root. However, the semantics of the credentials and the logic for reasoning about them are different in the two systems. The difference is that SHARP deals with delegations of claims on quantities

of typed resources (tickets), rather than attributes. We do not believe that the ABAC attribute model is sufficiently powerful to represent resource tickets.

The ORCA team believes strongly that the GENI AM-API should permit resource tickets and ticket delegation in a way that is compatible with the SHARP model. These mechanisms will enable resource brokering as an approach to federation-wide resource allocation policy. ABAC can be used to define the brokering trust structure, and brokers and/or aggregates may use ABAC within their resource allocation policies. SHARP and ABAC are complementary and interoperable, and both are needed to address the full range of challenges for federation-wide policy.

Note: an alternative means to implement federation-wide resource allocation policy is to proxy requests through a *Slice Tracker*, as described in Section 4.2. The SliceTracker applies the federation-wide policy, and rejects requests that do not comply with the policy. This alternative is useful if the policy agent must track all details of allocated slivers (e.g., VLAN tags, IP addresses) in addition to applying the policy. The Teagle project is based on this approach. Discussions with Aaron Falk in June 2011 suggest that GPO is considering the Teagle/SliceTracker approach as a function of the GENI Clearinghouse. The ORCA team continues to plead that the ticket/broker approach should not be excluded. We argue for a ticket brokering service within the clearinghouse, coupled with an asynchronous publish/subscribe event feed from aggregates to GMOC and/or other subscribers.

5 Representing Ownership

The basic authorization rules for an aggregate are simple. A subject can operate on an object if the subject created the object, and hence "owns" the object. We might also allow an owner to endorse or recognize other principals as owners, or otherwise endorse them to operate on the object. We call this *delegation of ownership*. Possession of any ownership role entails some privilege to operate on the object.

In essence, this view amounts to a generalized capability model for dynamically created objects. Of course, some objects might offer limited forms of "ownership" that confer privilege for subsets of operations. Specifically, ownership may be represented by one or more capability attributes $[C_o^1, ..., C_o^i]$ that are independently delegatable, rather than grouping all rights on an object O into a single capability role called $Owner(O)$. We can combine these models if we add a few new rules in the authorization policy for O, stating that any owner has all capability rights.

$$\begin{array}{l} A.C_o^1 \leftarrow A.Owner(O) \\ ... \\ A.C_o^i \leftarrow A.Owner(O) \end{array} \tag{18}$$

Of course, we might want to extend that with delegatable and non-delegatable forms of the capabilities; see Section 3. In any case, the rest of this section centers on a hypothetical $Owner(O)$ attribute, but the discussion applies to any capability role.

Delegation of rights implies that any entity with an ownership role or capability role obtained this role through a chain of delegations anchored in the object's original creator. It follows that the authorization policy for an object at a server must know the identity of the object's creator, so that it can accept the creator as an anchor for a chain of delegations of ownership. Thus the server procedure that creates or registers an object must determine the object's original creator and generate rules conferring ownership rights on the creator. It can either store these rules in the authorization policy for the object, or issue them as credentials to the creator.

For example, let us suppose that S was the subject of the operation that created O, and that this means that S is the owner of O and is empowered to delegate any and all ownership rights. The server procedure that creates the object generates the following rule, which may be issued as a credential or added to the authorization policy for O:

$$A.Owner(O) \leftarrow S \tag{19}$$

If the create request was proxied through a subject that *SpeaksFor* a user, the policy rule can take that user to be the owner of the object. There may be more complex conditions for ownership, e.g., in the case of global objects such as slices, as discussed below.

5.1 Ownership of Slivers

For GENI aggregates, we propose that ownership of a slice implies ownership of all of its slivers. That is, an operation on a sliver should be allowed to any owner of the slice, or an entity with a capability conferring suitable rights to the containing slice. This is simple to represent: the privilege to operate on a sliver is allowed to any entity that is an owner of the containing slice. For example, for a sliver that is a virtual node N in slice O:

$$A.Restart(N) \leftarrow A.Owner(O) \tag{20}$$

That says "an owner of a slice may restart any of its slivers". This rule does not require an RT1 extension to ABAC, since the rule can be generated automatically at the time the sliver is created. The new rule is parameterized at generation time by the containing slice O, and is referenced in the authorization policy for N.

An alternative implementation is for the slivers to incorporate the authorization policy of the containing slice by reference. This seems straightforward.

5.2 Ownership of Slices

It remains to determine how an aggregate infers ownership of a slice. Slices are global objects whose creators are determined outside of any aggregate that learns of the slice. In this case, the aggregate must take care to recognize the original creator of the slice, and initialize the authorization policy to ensure that an entity can operate on the slice if it has capability or ownership rights obtained through a transitive chain of delegations rooted in the slice's creator. Alternatively, a subject may obtain its privilege through some other rule (such as the GMOC rule above).

Recognizing the object's creator might be tricky, because the subject that first registers the slice at an aggregate (e.g., as a side effect of creating a sliver for the slice) might be someone other than the original creator of the slice, or it might be speaking for someone other than the original creator of the slice.

We take it as given that the slice is originally created by some entity whose identity is bound to the slice name by some secure mechanism. The *self-certifying identifier* (scid) construct (a GUID concatenated with a public key) in SFA 2.0 is one means to establish this binding. The scid construct enables any entity to create a global object named by a GUID, and prevent any other entity from hijacking the object's name, because the name is qualified with the creator's public key. In this case, the creator may generate its own credentials for the global object, and the servers will recognize them.

Another way to establish the binding is to require that the global object is endorsed by some entity that is recognized as a trust anchor by all servers that might operate on the object, as in the SliceAuthority proposal (Section 4.1). In this scenario, the policy that registers the slice might require some specific assertion of slice ownership and/or capability rights rooted in the endorsing SA. The endorsed owner can be taken as the original creator of the slice.

6 External Identity Providers

This section summarizes the plan for bridging GENI to external identity providers through one or more instances of an intermediate trust anchor called a GENI identity portal, and some related issues. The GENI identity portal acts as an identity provider within the GENI federation, but obtains some or all of the identity attributes from external IdPs. We use the terms IdP and identity provider generally to refer to any entity that is trusted by some other entity to assert attributes of a subject identity without proof. In particular, a GENI identity portal is an IdP.

Users log in to a GENI identity portal using some kind of Web-based authentication system. The IdP issues credentials for use with GENI services, and GENI services have local authorization policies that trust the portal to reflect that identity information into GENI. The portal-issued credentials are ABAC credentials that assert arbitrary attributes or roles of the subject S. In one scenario, the Web-based authentication system for the portal is a federated single sign on (SSO) system, such as Shibboleth/SAML, which is widely deployed at US universities. In this case, the ultimate sources of the attributes for each subject S are members of a network of participating SSO identity providers, each representing some institution or enterprise. The portal reads assertions from these SSO identity providers and issues an ABAC credential attesting to the asserted attributes.

Consider a specific example. A user S connects to a GENI portal with an HTTPS Web session or other SSL session, authenticated through Shibboleth. The GENI portal is a Shibboleth "Service Provider" (SP) configured to trust institutional identity providers affiliated with some Shibboleth federation, such as InCommon (*incommonfederation.org*), which defines a name space of attributes and roles. One affiliated identity provider issues an assertion to the GENI portal that says: "the Duke Shibboleth server *shib.oit.duke.edu* says that this session is controlled by user *chase@cs.duke.edu* and this user is a faculty member affiliated with Duke". Once logged into the portal, the user S uploads a public key K within the authenticated session. The portal then issues an ABAC credential saying"GENI IdP believes that public key K is controlled by user *chase@cs.duke.edu* and this user is a faculty member affiliated with Duke". The user S clicks a Web form to retrieve the credential, stores it in a file, and logs out of the session. S may then run hands-free experiment control tools with its user identity, allowing those tools to retrieve the credential from the file and incorporate it into their credential sets.

This proposal is approved by principals of the Shibboleth effort, but we are all mindful that three key technical concerns remain:

- **Privacy.** The ABAC credentials issued by the GENI Portal may reveal Shibboleth attributes of S to GENI-affiliated services (such as Aggegates) other than the portal itself, which is the only Service Provider approved by the Shibboleth identity provider (*shib.oit.duke.edu*) in this example. Strictly speaking, this possibility violates a core precept of Shibboleth: the user's identity provider controls whether any given user attribute is revealed to any given service provider.

  Response. when S logs in to the GENI Portal, S must be made to understand that the portal is a proxy for other GENI-affiliated services, and that any user S attributes may be revealed to any GENI-affiliated service that S uses. ("Click here to approve the GENI privacy policy.") In Shibboleth terms, all of GENI is viewed as a single SP.

- **Zombie credentials.** The ABAC credentials issued by the GENI Portal have a lifetime given by the expiration time in the X.509 credential. These credentials may outlive the SSO-authenticated session from which the attributes are harvested. Strictly speaking, this possibility violates a core precept of Shibboleth: the Shibboleth-asserted attributes are bound to a session, and must be reasserted if the session expires or the user logs out and/or logs in again.

  Response. The credential lifetime must be chosen carefully, and some means should exist for any GENI-affiliated server to obtain revocation status for an ABAC credential. Since an ABAC credential is an X.509 digital certificate, the proposal is compatible with use of the Online Certificate Status Protocol (OCSP) of RFC 2560 for this purpose.

- **Anonymity.** Some GENI participants believe that the binding of any GENI user to a real-world identity should be known to every GENI service that the user interacts with, so that everyone knows "which throat to choke" if something goes wrong. Shibboleth is explicitly designed to conceal a user's real-world identity, at the discretion of the user's Shibboleth identity provider. If an identity provider chooses not to reveal a user's identity in the attributes, then it might be impossible to hold the user accountable for actions taken within GENI, unless the institutional security administrators are willing to identify the user and/or help with the choking.

  Response. Any participating GENI service may, at its discretion, have a local policy that requires the user's real-world identity to be revealed. If the Shibboleth identity provider for S does not reveal the identity of S, then S will be unable to use any parts of GENI whose operators insist on knowing user identity.

And there are some other issues that have been discussed:

- **Attribute provision.** Operational testbeds that participate in GENI may have other specific requirements for user attributes (e.g., contact phone number). It must also be possible to manage attributes in some reasonable way without interfacing with institutional security teams. Grouper and CoManage. In the identity discussion, we have been talking about what identity attributes can be collected through an IdP system based on shib, and whether other attributes of the user need to be collected after the cert is issued. I think ABAC helps here: any other entity that collects any attributes for an identity and its corresponding public key may feel free to issue additional separate ABAC credentials asserting those new attributes. That is, other entities may serve as ABAC attribute roots.
- **Identifying the IdP.** When somebody logs into the portal, it's not always clear who their home IdP is. We think they should just enter their e-mail address, and the system should redirect to the default IdP for the home domain.

Acknowledgements

Some of the ideas in this note originated with Prateek Jaipuria, and many were derived from a 2/14/11 GENI-ABAC proposal written by Ted Faber. Some ideas were surfaced and refined in discussions with Steve Schwab, Ted Faber, Rob Ricci, Aaron Helsinger, Tom Mitchell, Victor Orlikowski, and/or Andrew Brown. We acknowledge the assistance of the ABAC (*libabac*) team, including Ted Faber, Steve Schwab, and Mike Ryan. The ideas pertaining to Shibboleth resulted form extensive discussions with Steven Carmody in 2009-2010, with additional contributions from Ken Klingenstein. In addition to GENI funding, Chase and Jaipuria received some funding from NSF CNS-0910653 (Trustworthy Virtual Cloud Computing) for related work involving the use of authorization logic in federated clouds.

References

[1] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[2] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[3] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-Recharging Virtual Currency. In *Proceedings of the Third Workshop on Economics of Peer-to-Peer Systems (P2P-ECON)*, August 2005.

[4] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Technical Conference*, June 2006.