ABAC Rules for GENI Authorization

Ted Faber, USC/ISI

John Wroclawski, USC/ISI

Version 1.1

14 February 2011

Table of Contents

1 Introduction

This document explains how to encode the existing GENIAPI[1] authorization structure, which is based in part of the draft Slice-based Facility Architecture[2], in terms of credentials in the ABAC attribute-based authentication system[3] that are interpreted by the existing libabac[4] toolset. We intend this primarily as a demonstration of the power of the ABAC model and toolset rather than an endorsement of that authorization structure. This work can be a starting point for integrating that toolset into the GENI ecosystem without changing the authorization logic. The encoding presented is clean and uses only the simplest ABAC logic. The rulesets described here are included in an associated code release of both ABAC credentials and demonstration code that shows its correct function with libabac.

This document has two intended audiences: the designers of present and future control frameworks and the developers integrating systems into their running prototypes. To the designers of current and future frameworks we intend to show that ABAC can simply encode an access mechanism of the complexity and power of the GENI access control in a clear and usable way, and is a viable basis for large-scale access control. To developers we intend this document to explain the details of that encoding and implementation. In serving these two purposes, the document is considerably longer than it would be for either audience individually. Section 1.1 breaks down the content of the document with that in mind.

ABAC is an attribute based authorization system developed at Stanford and Network Associates Laboratories[3] that encodes authorization rules in a formal logic based on attributes that principals assign to other principals. Statements in the logic capture the trust relationships between the principals and the rules for making access control decisions. Policies expressed in the system can be audited and analyzed by outsiders and decisions include a description of the reasoning that led to them. We believe that using such a system as the basis for GENI authorization will lead to a more scalable and reliable authorization system.

ISI has implemented a version of ABAC called libabac[4]. This includes a library accessible from many compiled and interpreted languages, including perl and python, as well as a unified toolset for creating and manipulating ABAC credentials. The rules described in this document are all implemented and have been tested using libabac.

Our earlier release of an ABAC-enabled GCF aggregate manager[5][6] includes a set of ABAC credentials that also encode the SFA, though more coarsely and only in the aggregate manager. That code runs with minimal modifications to the GCF code. The set that accompanies this document is a cleaner, more complete encoding of the SFA into ABAC, but will require some recoding of the applications to use in practice. In effect, this is a second implementation of the GENI authorization, aimed more at clarity and correctness than at simplicity of integration with an existing codebase. It can also be integrated.

The purpose of this exercise is not to advocate for the existing GENI authorization, but to demonstrate that ABAC is powerful enough to encode it. In fact, we believe that there are many shortcomings to the that authorization framework.[6] The ABAC encoding presented here captures the full semantics of the GENI access control, including credential delegation in the simplest ABAC logic, RT0. We believe that the credential delegation rules, as expressed in ABAC are clearer than in the implementation. That delegation is not a special case of the ABAC description, but used rules in the same logic as the rest of the system.

1.1 Reading This Document

Because this document presents sufficient background material for both readers unfamiliar with the GENI authentication structure and ABAC to understand it, it may seem longer and more complex than we intend for readers familiar with these complex systems. To those unfamiliar, the GENI system is intricate enough that explaining it requires some space, and the ABAC logic, though small, requires explanation as well.

For readers who are familiar with GENI authorization and ABAC, Section 5 contains the encoding of GENI authorization into ABAC, including GENI hierarchical trust and credential delegation. Section 6 explains how to use the machine readable certificates and libabac to explore that encoding.

For readers who would like more background before seeing ABAC rules, that material is provided in the earlier sections. Section 2 describes the GENI credential format and delegation model and Section 3 reviews how those GENI credentials are used in creating a slice. Section 4 reviews the ABAC logic for representing rules and credentials. The discussion in Section 5 should be comprehensible and convincing with that background, but if readers are interested in nuances the source GENI API and ABAC documents may be necessary.

2 GENI Credentials

The GENI credentials defined in the SFA and implemented in the GENIAPI are assertions by a trusted principal that a second principal has a particular set of rights to a target principal. In English a GENI credential might be interpreted as, “The GPO Clearinghouse says that user faber has a set of rights at the GPO slice authority.” Each principal is identified by a GENI identifier (GID), which is implemented as an X.509 certificate binding a key to a UUID. We briefly discuss the principals and the credential formats below.

There are a couple sources for the SFA authorization model, both the SFA specification and the specifications and implementations of GENI control frameworks. While these agree at a high level, when there are differences in detail, we generally follow the example set by the GENIAPI[1] or the ProtoGENI control framework[7]. We believe the GENIAPI to represent significant consensus in the community and ProtoGENI to be the running codebase that most closely adheres to it, and follow their lead for that reason.

Because this document is primarily concerned with ABAC credentials, we will explicitly use “GENI credential” when talking about GENI credentials.

2.1 Principals

GENI principals are identified by GID, which is a hierarchically attested binding of UUID to public key of the principal. Principals act as users, administrators, slice authorities, aggregate managers and a few other things. Slices and slivers are identified as principals, albeit ones that are only acted upon.

The ABAC model is amenable to identifying principals using that system, though we prefer using self-certifying identities rather than adding a trust hierarchy for identity assignment. Crossing or merging trust hierarchies is generally harder when two similar structures must be coordinated. The ABAC rules in Section 5 assume the self-certifying names.

2.2 Credentials

A GENI credential maps certain abilities to a *source GID* with respect to a *target GID* based on the credential's *type*. Additionally, the rights of the credential may be delegated if the credential indicates so, through a *delegate* flag. The source GID identifies the actor, the target GID identifies the object to be operated on, and the type implies rights that determine what operations are permitted.

We show the rights in Table 1, as derived from the GENIAPI implementation (v1.2) of GENI credentials.[5] There have been some other recommendations as well, but all share the spirit captured here, if not the particular mappings.

The GENI credential's rights are modified by the target principal. A GENI credential conferring authority rights targeted to a clearinghouse allows one to resolve information at that clearinghouse, but not an associated slice authority or aggregate manager.

Rights	Operations Authorized
authority	register, remove, update, resolve, list, getcredential, *
refresh	remove, update
resolve	resolve, list, getcredential
sa	gettticket, redeemslice, redeemticket, createslice, createsliver,deleteslice, deletesliver,updateslice, getsliceresources, loanresources, startslice, stopslice, renewsliver
embed	gettticket, redeemslice, redeemticket, createslice, createsliver, renewsliver, deleteslice
bind	getticket, loanresources, redeemticket
control	updateslice, createslice, renewsliver, sliverstatus, stopslice, startslice, deleteslice, deletesliver, resetslice, getsliceresources, getgids
info	listslices, listnodes, getpolicy
ma	setbootstate, getbootstate, reboot, getgids, gettrustedcreds
operator	gettrustedcerts, getgids

Table 1: GENI Credential Rights & Operations

Each GENI credential has a type, from which its rights are derived. Those types and their rights are listed in Table 2. The operations allowed are the union of those allowed by each right in the list. Table 2 is a little confusing in that GENI credential types and rights share some names, though the name spaces are distinct.

When evaluating whether a GENI credential enables a particular operation, the principal that will perform the operation confirms that it is the target and that the requester is the subject. Then the type of the GENI credential is converted into the rights conferred by it, and the rights converted into the list of valid operations. The target then confirms that the requested operation is in the list.

Type	Rights
user	refresh, resolve, info
sa	authority, sa
ma	authority, ma
authority	authority, sa, ma
slice	refresh, embed, bind, control, info
component	operator

Table 2: GENI Credential Types & Rights

We generally find the multi-level hierarchy more confusing than enlightening and through the following discussions and the ABAC derivation, we treat the permission to carry out an operation as the fundemental access control primitive, rather than the permissions above. We also include ABAC credentials that implement this two-level hierarchy in Appendix 2.

2.3 Delegating GENI Credentials

Under the SFA/GENIAPI credential system, a user who holds a GENI credential that entitles them to a set of rights can pass all or some of those rights to another user. A user can only pass on rights encoded by a GENI credential if that GENI credential has its delegate flag set.

The mechanism for doing this is that the delegating user creates a credential for the delegatee that is a copy of the GENI credential to be passed on, and signs both the new credential and a package containing both the new and old GENI credentials. The new GENI credential can contain a subset of the rights of the original, and may or may not have its delegate flag set. If the new GENI credential does have its delegate flag set, it may also be delegated, as long as it is packaged with the delegated credential, which preserves the signed delegation chain to the original issuee.

Any delegator can stop further delegation by clearing the delegate flag in subsequent credentials.

3 The SFA/GENIAPI Operation Flow

This section lays out the basic flow of control and of credentials in GENI for creating and manipulating a slice and shutting down a slice. We generally show these from the user's perspective, because these are the control paths that are most standardized. We focus on the flow of credentials, because although ABAC expresses the atoms of access control differently, the flow of authority will remain the same.

The basic players in these interactions are the *Clearinghouse* sometimes referred to as the Registry in other documents, *Slice Authority*, *Aggregate Managers*, and the *User* (or its agent). Conceptually these do the following:

- A Clearinghouse keeps data about users and the federations with other clearinghouses. clearinghouses are generally used as trust roots for the other components in the system. They certify users and vouch for their identities and issue credentials valid at other principals that trust the clearinghouse.[1]

- A slice authority keeps information about which slices are active and which users can access them. It also may coordinate with aggregate managers to keep track of the resources allocated to a slice, though this is not well standardized.

- Aggregate mangers allocate resources to users and attach them to slices, though the interface through which the Slice Manager becomes aware of this binding is underspecified.

- Users are the creators of slices at slice authorities and the entities that bind resources from aggregate managers to slices. Information about them may be collected at clearinghouses.

3.1 Creating And Using A Slice

In Figure 1 we show the basic flow of control for creating a slice. As we discussed in Section 2, this is based on the SFA, GENIAPI and ProtoGENI control flow.

When creating a slice, a user initially retrieves identity-based credentials from the clearinghouse, uses those credentials to create a slice at the slice authority, and then uses those slice credentials to acquire slivers from various aggregate managers that respect them. The identity-based credentials describe the rights of the user with respect to the slice authority. The slice authority issues a credential that describes rights with respect to the slice, which the

[1]The authors believe that the clearinghouse is extraneous to the architecture, but this document shows that ABAC can support the entity.

aggregate managers interpret as granting rights to make resource allocations. Upon sliver creation the ProtoGENI aggregate managers return sliver credentials as well, though the current GENIAPI does not.

In more detail:

1. The user gets the set of GENI credentials from this clearinghouse based on its identity (GID). These will be the credentials that confer rights to the slice manager and perhaps the aggregate managers under this clearinghouse, though in practice it is a credential to one or more slice authorities.

2. The user creates a slice at the slice manager using the register operation, assuming it has credentials that allow that. The user is given a GENI slice credential by the slice authority.

3. Resources are allocated using the slice credential. This is a strange use of the credential format, as the credentials used in practice do not name the aggregate managers as objects, but the slice as an object; there is an inference made outside the given rules for this authorization. The ABAC encoding in Section 5 replaces this implicit credential with an explicit one.

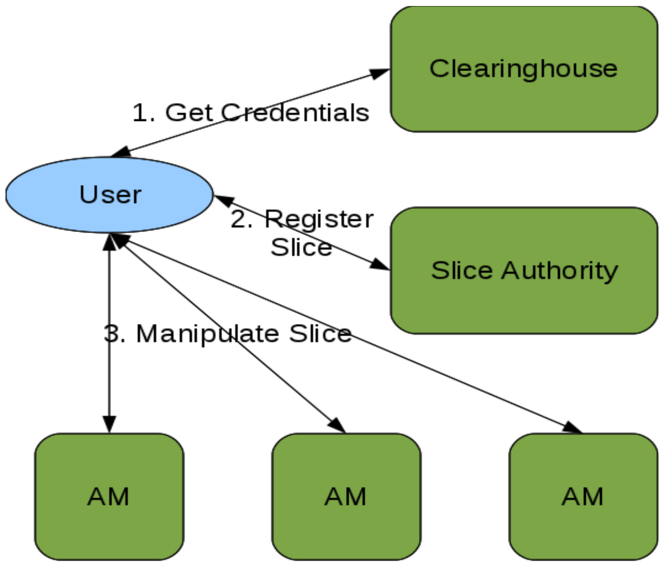

Figure 1: Slice Creation Flow

Step 2 may involve several operations on the slice authority. In particular, *Resolve* and *Register* are both probably used to confirm that the slice name to be used is unique and to create the slice. The *discover* operation may be used to find AMs known to this slice authority.

Step 3 may involve using *ListResources* to discover the available resources controlled by the AM and then a sequence of *CreateSliver* and *SliverStatus* operations to build and confirm the operation of the sliver. The more general SFA interface may expand the operations by adding operations on tickets – unrealized resource reservations – as well as the more direct creation calls. Over the lifetime of a slice and its slivers, the *RenewSlice* and *RenewSliver* operations on the SA and AM may be used as well. When the slice/slivers are released, the *DeleteSliver* operation will be called on the AMs and *Remove* on the AS. The ProtoGENI implementation simply times slices out, so the *Remove* is not required.

While we have shown a single user in Figure 1, these operations may be carried out by other users to whom the first has delegated authority, if that authority is delegable.

3.2 Shutdown

A slice is shutdown when it seems to be misbehaving. It is more interesting than the control flow above because the user that shuts down the slice may not have been given the right explicitly by the user that created the slice. How these credentials are created and passed around is somewhat underspecified, but we will suggest a reasonable mechanism below.

4 ABAC RT0 Semantics

The RT0 logic implemented by libabac v0.1.2 is simple and powerful, but the notation is somewhat terse. This section reviews the semantics and contrasts the RT0 expression of parameterized expressions with that of RT1. Section 5 specifies the rules in RT0, because that is what is implemented now.

4.1 RT0 Logic

ABAC's RT0 logic[3] allows one to attach an attribute to a principal, define a direct delegation rule and define a rule linking the possession of an attribute to the ability to delegate attributes. This section reviews the notation and semantics.

Libabac uses the SHA1 hash of the principal's public key, called a *key identifier* in the libabac documentation, as that principal's identifier. In this document we present the various principals symbolically; an aggregate manager principal may be written as AM1 rather than a long string of hex digits.

In ABAC's logic an attribute is a string attached to a principal by another principal. If an aggregate manager identified as `AM` wishes to attach the `ListResources` attribute to a user identified as `U` we say that `AM` has attached `AM.ListResources` to `U`. Only `AM` can assign attributes from the `AM.` space. Furthermore `AM1.ListResources` and `AM2.ListResources` are distinct.

There are three ways to attach an attribute to a principal:

1. **Direct assignment**. Example: `U` has attribute `AM.ListResources`. Notation:
   `AM.ListResources ← U`

2. **Delegation**. Example: All principals with attribute `AM2.ListResources` have `AM1.ListResources`. Notation:
   `AM1.ListResources ← AM2.ListResources`

3. **Linked Delegation**. Any principal that has resource `ListResources` assigned to it by a principal with the `AM2.Linked` attribute has the `AM1.ListResources` attribute. Notation:
   `AM1.ListResources ← (AM2.Linked).ListResources`

Direct assignment is pretty straightforward. A principal binds another attribute to another principal. If we take `AM.ListResources` to indicate the ability to call the ListResources operation on `AM`, the example in 1. says that `AM` has explicitly granted that ability to user `U`.

In the second example, `AM1` has expressed a rule delegating the ability to assign principals the `AM1.ListResources` attribute to `AM2`. `AM2` exercises that delegation by assigning its `AM2.ListResources` attribute. Any principal that knows `AM1.ListResources ← AM2.ListResources` and `AM2.ListResources ← U` knows that `U` has `AM1.ListResrources`.

In some cases, `AM1` and `AM2` will want to coordinate closely about such a delegation, but in others `AM2` may be oblivious to the delegation. If `AM2` is a well known certifier or `AM1` and `AM2` have a general relationship where they agree on the semantics of `ListResources`, there is no reason to discuss the delegation. In any case, ABAC does not require any coordination to make the delegation.

The last example adds a second indirection. This delegates `AM1.ListResources` to a number of other principals that have an attribute assigned by `AM2`, not to `AM2` itself. In this case a principal must know that

`AM1.ListResources` ← `(AM2.Linked).ListResources` and `AM2.Linked` ← `V` and `V.ListResources` ← `U` to know that `U` has `AM1.ListResources`.

This generally allows a principal to appoint agents, other principals that will assign an attribute on its behalf. The example above showed one principal (`AM1`) directly delegating that authority to the agents of another (`AM2`). A ruleset more like `AM2.ListResources` ← `(AM2.Linked).ListResources`, `AM1.ListResources` ← `AM2.ListResources` lets `AM2` express its creation of agents and `AM1` delegate to the other principal. The first rule is controlled by `AM2`, because it controls the `AM2.ListResources` attribute, and the second rule is controlled by `AM1` for the analogous reason.

The requirements for a delegation – the right hand side of the arrows above – can include conjunctions. For example, `AM.CreateSlice CH.CreateSlice & SA.CreateSlice` says that AM will assign the AM.CreateSlice attribute to a principal that has demonstrated that it has both `CH.CreateSlice` and `SA.CreateSlice.`

Each of these operations, the assignment of an attribute or the creation of a delegation rule, is expressed in a *credential*. A credential is a signed statement of the rule or assignment in RT0 logic, signed by the principal that controls the attribute being assigned or delegated. That is, a credential is signed by the principal whose identity is attached to the attribute on the left side of the arrow.

Such credentials form the basis of proofs in the ABAC system, and are understandable by any entity that can confirm the signatures. Libabac represents these in X.509 attribute certificates, and the examples use that format.

4.2 RT0 and RT1

The current ABAC implementation implements RT0 logic, which is described in Section 4.1 above. We expect to implement RT1 in the near future, which supports the same basic delegation rules, but with parameterized attributes. For example the `AM.DeleteSliver` attribute may be scoped to a single sliver by asserting an `AM.DeleteSliver(sliver3)` attribute.

Such parameterized attributes may be reasoned about just as unparameterized attributes are, or with some constraints on their parameters. For example, an attribute like `AM.Level(4)` can appear in a rule as `AM.Level(?)`, indicating any level is acceptable (but that an AM.Level must be assigned) or as `AM.Level(?:[2..5])` indicating that the parameter must be in the range 2 to 5 inclusive; constraints must limit parameters to finite static sets.

The rules for encoding current GENI access control do not rely on level comparisons, but they do benefit from scoping to a given slice or sliver name. In an RT1 world, we would express this like `AM.DeleteSliver(uuid)`; lacking RT0 we express this scoping in the name of the attribute. For example, that attribute is `AM.DeleteSliver_uuid` .

Because the rules for GENI access never require arithmetic or other operations, just matching, and only principals who issue scoped attributes need to interpret them, we can guarantee that no confusion results.

5 ABAC for GENI Access Control

This section describes the ABAC implementation of the GENI access rules. These rules include both the underlying policy rules and the specific attributes created by the principals in each of the major operations. This section begins by explaining how the principal and attribute used to prove access to an operation are selected. Then we describe the ABAC expressions for hierarchical attribute propagation that implement GENI's hierarchical trust assignment, then describe the static policy at each principal, and then the dynamic attributes issued for operations that create system objects. Finally we describe the complete operation of a standard slice lifetime and also a shutdown.

5.1 What to Prove

ABAC provides rules for proving principals have attributes; in order to use that for access control, the provider of a service needs to decide what to prove about what principal to make the decision. In the simplest case, each operation is mapped to an associated attribute in the service providing principal's attribute space. In most of the GENI operations below we do just that.

Some operations must be parameterized by the object or type of object on which they are operating. Allowing any principal that possesses `SA.DeleteSlice` to delete any slice at slice authority `SA` is clearly unreasonable. We parameterize such calls with the target slice. In the descriptions below we use UUIDs for those parameters because the GENI interface is defined in those terms. We could just as easily use self-certifying names.

Finally a few operations, for example Resolve, are controlled not by the specific object being addressed but by the type of object being addressed. Administrators can resolve users while users can only resolve slices. We parameterize these by type: `Resolve_users`.

In practice the mapping from ABAC attribute to operation is part of the configuration of the system. In the descriptions that follow, the attribute for a given operation is always the same as the name of the operation, parameterized as above if necessary.

The other question is which principal to prove holds the attribute. In the GENI API, the principal making the call is always known because the principal is bound to the request by the SSL protocol used to contact the service.

5.2 Hierarchical Attribute Propagation

After this section, many of the access control rules are straightforward applications of the obvious policies, but the propagation of trust through a hierarchical structure is common, powerful and not obvious to people the first time they see ABAC. This section shows that hierarchy is both straightforward to encode and economical to implement.

In the GENI access control, a slice authority or aggregate manager accepts a credential as valid if the credential has the familiar X.509 signature chain to a trusted root. Any principal either in that set, or with a key signed by that set can act as a credential issuer.

This is expressed in ABAC by the following rule at an aggregate manager (`AM`)[2]:

```
AM.clearinghouse ← (AM.clearinghouse).clearinghouse
```

This says that any existing entity that `AM` says is a clearinghouse can designate another principal as a clearinghouse. Consider 3 clearinghouses, `CH`, `CH1` and `CH2` . Let `AM` have a rule `AM.CreateSliver ← (AM.clearinghouse).CreateSliver`. That is, the `AM` will respect a `CreateSliver` attribute from any `AM.clearinghouse`.

In order to establish a trust root at `CH`, `AM` issues a credential that says `AM.clearinghouse ← CH`. When `CH` wants to make `CH1` a clearinghouse, it does so by issuing a credential that says `CH.clearinghouse ← CH1`. Just to keep all this straight, we number the credentials:

```
1) AM.clearinghouse ← (AM.clearinghouse).clearinghouse

2) AM.clearinghouse ← CH

3) CH.clearinghouse ← CH1
```

[2]The aggregate manager is just used as an example of a principal that respects hierarchical delegation. As we will see in Section 5.4 both aggregate managers and slice authorities do this in the current GENI semantics.

```
    4) AM.CreateSliver ← (AM.clearinghouse).CreateSliver
```

Now there are two cases that will give the flavor of the function here. Assume `AM` requires a principal to show `AM.CreateSliver` to call `CreateSliver`, and has ruleset above in place.

- **Case 1:** a requester `R` appears calling `CreateSliver` holding a `CH.CreateSliver ← R` credential. `AM` needs to prove that `CH` has a `AM.clearinghouse` attribute so that rule 4) above will map `CH.CreateSliver` (which R has) into `AM.CreateSliver`. That requirement is met directly by rule 2) and the call can proceed.

- **Case 2:** a requester `R` appears calling `CreateSliver` holding a `CH1.CreateSliver ← R` credential. First we show that `CH1` has `AM.clearinghouse`. Rule 1) and rule 2) show this. Then because CH1, an `AM.clearinghouse`, says `P` has `CH1.CreateSliver`, rule 4) allows the access. See Appendix 1 for more step-by-step proof of this.

Case 2 above is really an inductive step. Once a principal is an `AM.clearinghouse` it can designate others as the same. An ABAC credential and a signature on a traditional X.509 certificate are similar cost operations, so the cost of delegating trust to an associated clearinghouse is comparable in both systems.

The general format of rule 1) is used below to give hierarchical inheritance of trust.

5.3 GENI Credential Delegation

Delegation of GENI credentials (Section 2.3) is also hierarchical, though expressed through a separate mechanism – chains of signed credential objects, rather than the signatures in an X.509 certificate. The mechanism is further complicated by the ability to delegate rights but not the right to further delegate. ABAC encodes this separate mechanism using the same logic as trust delegation, though with small differences that capture the distinction of delegating a right and delegating the right to further delegate.

Expressing this in ABAC takes the form:

```
    AM.attr ← (AM.delegate_attr).attr
```

Where the creator of the attribute (CH, above) also creates a rule for hierarchical transfer of the delegation right that looks like rule 1) above:

```
    AM.delegate_attr ← (AM.delegate_attr).delegate_attr
```

The first rule says that any delegator can pass on the attribute by assigning it in their local attribute space. If the delegator wishes the delegatee to be able to pass thr ight along further, a second attribute must be passed on. As a concrete example, we repeat the CreateSliver analysis from Section 5.2 again under these rules, adding a failed delegation:

```
 1) AM.delegate_CreateSliver ←
    (AM.delegate_CreateSliver).delegate_CreateSliver

 2) AM.delegate_CreateSliver ← CH

 3) CH.CreateSliver ← CH

 4) CH.delegate_CreateSliver ← CH1

 5) CH.CreateSliver ← CH1
```

6) `CH1.CreateSliver ← CH2`

7) `CH2.CreateSliver ← CH3`

8) `AM.CreateSliver ← (AM.delegate_CreateSliver).CreateSliver`

If a principal `CH1` shows up at `AM` (who expects to prove `CH1` has `AM.CreateSliver` to let it make the call) the call will be allowed. `CH` has `AM.delegate_CreateSliver` (rule 2)) and it is capable of delegating the `CreateSliver` right to `CH1` using rule 8).

If a principal `CH2` shows up at `AM`, the call will be allowed. `CH1` has `AM.delegate_CreateSliver` (fom the reasoning in the previous paragraph) and therefore rule 8) recognizes `CH1.CreateSliver` (which rule 6) gives to `CH2`) as equivalent to `AM.CreateSliver`.

If a principal `CH3` shows up at `AM`, the call will fail. `CH2` does not have `CH.delegate_CreateSliver` and therefore is not in the set of principals that rule 8) recognizes as being able to confer `AM.CreateSliver`; `CH2`'s attempt to do so via rule 7) fails. `CH2` can use (previous paragraph) but not delegate the GENI credential.

Three comments about this delegation structure: First, compared to the GENI credential delegation, this is fairly straightforward to express. GENI delegation requires separate semantics from the (somewhat convoluted) access control logic, while the ABAC formulation depends on the same rules and uses the same engine to validate both kinds of delegation (heirarchical and explicit).

Second, this section was framed in terms of the low level attribute CreateSliver rather than the GENI rights from Section 2.2. Hopefully this presentation was simple enough to follow; the basic format of the dual attribute inheritance works for rights as well, but the additional rules to deduce operation permissions from GENI rights would only cloud things.

Third, this structure requires more signed data than the GENI implementation.

5.4 Policies and Credentials

This section describes the ABAC policies that implement the GENI semantics at each player. We also describe credentials that are issued as the result of successful calls to the API.

5.4.1 Clearinghouse

When a user is enrolled at a clearinghouse, `CH`, the clearinghouse issues the user a credentials allowing the user to register itself and others as well as to list components at `CH`. For a principal `P` these credentials are:

1. `CH.GetCredential ← P`
2. `CH.Register_user_UUID ← P`
3. `CH.Register_slice ← P`
4. `CH.Resolve ← P`
5. `CH.ListComponents ← P`

The specification is vague on how entries in the registry are access controlled. The intent of the above is to allow the user to edit their user information, keyed by UUID, and to create/register slice records. Generally a user will create those records through a slice authority, which we discuss in Section 5.4.2.

When an entry is successfully made by `P`, CH returns a credential of the form `CH.Remove_UUID ← P` where the UUID identifies the registry entry that was created. This allows a user to remove the entries it creates.

Administrators at `CH` will need to call Shutdown, List, Register, and Remove on the various pieces of data in the registry. `CH` should issue credentials to administrators of the form `CH.admin ← A`, where `A` is an administrator. When a registry entry is created, in addition to issuing a credential to the user creating the entry, `CH` records a credential of the form:

6. `CH.Remove_UUID ← CH.admin`

And when a slice is registered (see Section 5.4.2), the clearinghouse issues a credential of the form

7. `CH.Shutdown_UUID ← CH.admin`

In addition, these credentials must be in place:

8. `CH.List_slices ← CH.admin`
9. `CH.List_users ← CH.admin`
10. `CH.List_authorities ← CH.admin`
11. `CH.List_components ← CH.admin`

The `CH.List_`*x* attributes give access to the various List commands. That set allows administrators to manipulate the registry and data while restricting normal users.

Shutdown is an interesting case, in that slice authorities may be willing to allow clearinghouse administrators to call Shutdown on slices. To facilitate this, the clearinghouse issues credentials of the form `CH.shutdown ← A` to administrators it trusts to make such direct slice authority calls. Of course, depending on the trust relationship between the clearinghouse and the slice authority, the slice authority may prefer to issue such attributes.

Most of the parameterized credentials can remain at the clearinghouse. They will be used internally for proofs and are not really exported into other clearinghouse name spaces.

Finally, CH needs to declare itself a clearinghouse:

12. `CH.clearinghouse ← CH`

When another clearinghouse becomes affiliated with CH, it issues a credential to that clearinghouse in accordance with Section 5.2: `CH.clearinghouse ← CH1`.

5.4.2 Slice Authority

The first set of rules at a slice authority, SA, recognizing the clearinghouse hierarchy (as from Section 5.2)

1. `SA.clearinghouse ← (SA.clearinghouse).clearinghouse`
2. `SA.clearinghouse ← CH`

The combination of rules 1. and 2. complete the hierarchical trust delegation to `CH` and the children `CH` designates.

From the clearinghouse attributes, the SA bootstraps others:

3. `SA.GetCredential ← (SA.clearinghouse).GetCredential`

4. `SA.GetKeys ←(SA.clearinghouse).GetCredential`

5. `SA.Register_slice ← (SA.clearinghouse).Register_slice`

6. `SA.Resolve ←(SA.clearinghouse).Resolve`

7. `SA.DiscoverResources ← (SA.clearinghouse).ListComponents`

These are straightforward recognitions of clearinghouse issued rights. This version of GetCredential is an analog of the clearinghouse call of the same name and an slice authority's DiscoverResources is a close analog of ListComponents[3]. GetKeys is an essentially parameterless call of the same form as GetCredentials. The Register_slice credential recognizes the current situation where the right to register a slice record at a clearinghouse is equivalent to the right to do so at a slice authority.

When a user exercises that right and creates a slice, in addition to the slice authority's bookkeeping, the user (still called P) receives credentials for manipulating the slice. These have the form:

8. `SA.GetCredential_UUID ← P`

9. `SA.Remove_UUID ← P`

10. `SA.Bind_UUID ← P`

11. `SA.Renew_UUID ← P`

12. `SA.Shutdown_UUID ← P`

13. `SA.CreateSliver ← P`

These simply allocate the rights to manipulate the new slice to the creator. To be complete, and somewhat depending on the policy set by the owner of the SA, these credentials may all be delegable. In which case, they are issued with a corresponding set of `delegate_` credentials as described in Section 5.3. For simplicity, we omit the delegation structure here.

Credential 13. tells aggregate managers that trust this slice manager that a valid slice exists and that slivers can be bound to it. The current meaning of this in GENI is confused, as it is unclear which player does the binding of sliver to slice. This simple representation reflects this unclarity. Completing this step shows that affiliated aggregates should respect the right to allocate resources, but there are multiple failure modes depending on how slice/sliver binding is ultimately specified.

Implicit in slice creation is the registration of slice records with the clearinghouse. It is strongly implied that slice records can be resolved at either the clearinghouse or the slice authority and explicitly stated that Shutdown can be called on a slice at either entity. To facilitate this, a clearinghouse issues credentials to all slice authorities of the form:

14. `CH.Register_slice ← SA`

And when a slice authority registers one on a clearinghouse, it should issue a credential of the form:

15. `SA.Shutdown_UUID ← CH`

That will allow the clearinghouse to make the call at the slice authority.

[3]ProtoGENI implements DiscoverResources as a call on an associated clearinghouse's ListComponents method

In the same way that the clearinghouse will issue administrator credentials, a slice authority may also do so. Rules 8.-12. will be replicated with SA.admin on the right side, and as with the clearinghouse, need not be propagated outside the slice authority. As in Section 5.4.1, the slice authority can designate it believes are trustworthy enough to call Shutdown on slivers by assigning them the `SA.shutdown` attribute.

Shutdown remains a special case. In addition to allowing the clearinghouses themselves to call the interface as 15. allows, the slice authority may want the subset of clearinghouse administrators identified in Section 5.4.1 to do so as well. That rule takes the form:

```
16.      SA.Shutdown_UUID ← (SA.clearinghouse).shutdown
```

It also can remain at the slice authority.

The slice authority can then propagate the trust in clearinghouses shutdown administrators using the rule:

```
17.      SA.shutdown ← (SA.clearinghouse).shutdown
```

5.4.3 Aggregate Managers

In the same way that slice authorities trust clearinghouses, aggregate managers trust slice authorities. These rules recognize that delegation and establish one root:

```
1. AM.slice_authority ← (SA.slice_authority).slice_authority

2. AM.slice_authority ← SA
```

The rules for sliver creation are similar to slice creation:

```
3. AM.ListResources ← (AM.slice_authority).DiscoverResources

4. AM.CreateSliver ← (AM.slice_authority).CreateSliver
```

After successful sliver creation, credentials of the following form are passed on to the user:

```
5. AM.DeleteSliver_UUID ← P

6. AM.SliverStatus_UUID ← P

7. AM.RenewSliver_UUID ← P

8. AM.Shutdown_UUID ← P
```

As with slice authorities, credentials 5. through 8. could be delegated and we omit the additional credentials and rules to do so.The same set of credentials are replicated to local administrators, and shutdown credentials issued to the slice authority's designated shutdown administrators:

```
9. AM.Shutdown_UUID ←(AM.slice_authority).shutdown
```

6 The Credential Distribution

This section describes the machine readable and rules that demonstrate that libabac implements the rules described above correctly. This code could reasonably be used as a starting point to bring ABAC code into production GENI entities.

The credential distribution is a set of python scripts to generate the configurations is Section 5, and to modify them as the result of adding a clearinghouse, getting user credentials, registering a slice, or creating a sliver. In addition we provide a a script to create and explore the GENI credential delegation scenario in Section 5.3.

These scripts both show that current software implements the expressive rulesets described in Section 5 and give implementers a departure point for experimenting with their own extensions to those rulesets.

The rest of this section describes those scripts in some detail.

6.1 Installing Software

The demonstrations all require a recent version of python (we recommend 2.6) and libabac, which also has some dependencies. Follow the installation instructions at http://abac.deterlab.net (particularly at http://abac.deterlab.net/browser/doc/INSTALL and http://abac.deterlab.net/browser/doc/dependencies) .

The scripts use both the ABAC prover in libabac and the creddy command line tools for creating and manipulating attribute certificates. Those tools all represent principals by their public keys, but the scripts translate these long key identifiers back to symbolic names. When you use the basic ABAC software, the long IDs will appear.

6.2 The GENI Access Control Layout

The scripts all represent principals as a directory named for the principal filled with credentials. Those credentials are X.509 certificates encoding the various ABAC rules and assignments. In addition, the implementation requires identity certificates to validate the signatures on the credentials. The scripts move the identity certificates into the correct places, but when implementers begin playing with their own versions, missing ID certificates are a common error.

To instantiate a copy of the basic GENI layout for one clearinghouse (`CH`), one slice authority(`SA`), one aggregate controller (`AM`) and one user (`P`), run the script `basic.py`. That creates the identity certificates for the principals and instantiates the basic rules from Section 5.4.

This looks like:

```
$ python ./basic.py
```

The script has created four directories, one for each principal which contain:

```
$ ls CH SA AM P
AM:
AM_ID.pem      root0.der      rule3.der
AM_private.pem rule1.der      rule4.der

CH:
CH_ID.pem      CH_private.pem

P:
P_ID.pem      P_private.pem

SA:
SA_ID.pem      root0.der      rule3.der      rule5.der      rule7.der
SA_private.pem rule1.der      rule4.der      rule6.der
```

The files ending in pem are the identity certificates and private keys of each principal. The der files are the policy rules, and their name corresponds to the rule they implement from Section 5.4. For example AM/rule4.der contains rule 4. from Section 5.4.3 (the aggregate manager section).

The root0.der files represent the explicit connection to the root of a trust hierarchy. For example, SA/root0.der contains the rule `SA.clearinghouse ← CH`.

To view the contents of a credential with an assignment or rule in it (a der credential), use the creddy command:

```
$ creddy --roles --cert SA/root0.der
d414577e3b4054c6aeb56893062c2054e53f4670.clearinghouse <-
fe94a8ed2d9cbcf97c39100d3e26a9e0cefc82e0
```

This reports the rule in terms of the public keys of the principals – libabac identifies principals that way, rather than by attaching other information. To simplify using these examples, we provide a tool to translate back to symbolic names, `keyid_to_principal.py`. This is a filter that looks for the structure established by basic.py (a principal in each directory), gathers the keys from the directories and maps the keys back to names.

Here is the same creddy command piped through `keyid_to_principal.py`:

```
$ creddy --roles --cert SA/root0.der | python ./keyid_to_principal.py
SA.clearinghouse <- CH
```

6.3 Proving Attributes

The proof.py script proves attributes within the structure established above. It takes the target principal and attribute to prove as well as the proving principals and returns either a proof that the target principal has the attribute or a false indication. All credentials known to the proving principals can be used in the proof. The proof.py script takes input and produces output in the symbolic name space.

Here is an example call:

```
$ python ./proof.py --principal CH --attr SA.clearinghouse SA CH
True
SA.clearinghouse <- CH
```

The `--principal` argument names the target, the `--attr` argument names the attribute to test and the `SA` and `CH` are directories containing the proving principals' credentials. The proof itself is straightforward, as we know from above that the `SA` directory contains the direct assignment that comprises the entire proof (in `SA/rule2.der`).

6.4 Taking Actions

There are several scripts that manipulate the credentials as though operations in Section 5.4 had occurred. These are listed below.

Script	Action	Reference
`get_cred.py` *user clearinghouse*	User gets credentials from clearinghouse	Section 5.4.1
`register_slice.py` *user slice_authority*	User registers a slice.	Section 5.4.2
`create_sliver.py` *user aggregate_manager*	User creates a sliver	Section 5.4.3
`delegate_ch.py` *new_clearinghouse parent_clearinghouse*	Hierarchically pass trust to a new clearinghouse	Section 5.2

Table 3: Action Scripts

These generate new credentials and put them in the principals' directories to be used in proofs. Of course these scripts just model the manipulation of credentials, not the other details of these operations.

6.5 Extended Example

Here is an example that creates a child clearinghouse, and lets a user registered at that child clearinghouse create a slice and proves the user has the right to do so.

The commands to create the environment, delegate trust to the new clearinghouse and get the credentials for user `P` are:

```
$ python ./basic.py
$ python ./delegate_ch.py CH1 CH
$ python ./get_cred.py P CH1
```

The call to `basic.py` is familiar. The `delegate_ch.py` call both creates the directory for the new clearinghouse and adds the credentials making the trust delegation from `CH` to `CH1`. Finally `get_cred.py` collects credentials for `P` signed by `CH1`.

Now we test `P`'s right to create a slice at `SA`.

```
$ python ./proof.py --principal P --attr SA.Register_slice P SA
True
CH1.Register_slice <- P
CH.clearinghouse <- CH1
SA.clearinghouse <- CH
SA.clearinghouse <- SA.clearinghouse.clearinghouse
SA.Register_slice <- SA.clearinghouse.Register_slice
```

The right is confirmed and the credentials used in the reasoning (described in Section 5.2) are output. We could now continue to test authorizations using ./proof.py and make further changes to the principal's authority using the scripts in Table 3.

6.6 GENI Credential Delegation

The credentials established by basic.py above are not the delegatable GENI credenials described in Section 2.2, but we do include another script, `cred_delegation.py`, that establishes an environment to show the implementation described in Section 5.3. That script establishes the principals and credentials in Section 5.3 and allows us to run proofs.

Here's a sample:

```
#Clear old principals
$ rm -rf ? ?? ???
$ python ./cred_delegation.py

#View principals

$ ls ?? ???
AM:
AM_ID.pem      AM_private.pem rule1.der      rule2.der      rule8.der

CH:
CH_ID.pem      CH_private.pem rule3.der

CH1:
CH1_ID.pem      CH1_private.pem CH_ID.pem       rule4.der       rule5.der
```

```
CH2:
CH1_ID.pem      CH2_private.pem rule4.der
CH2_ID.pem      CH_ID.pem       rule6.der

CH3:
CH2_ID.pem      CH3_private.pem rule4.der
CH3_ID.pem      CH_ID.pem       rule7.der
```

This script has establsihed the conditions in Section 5.3, and again, filenames correspond to the rules in that example. The two more interesting tests are:

```
$ python ./proof.py --principal CH2 --attr AM.CreateSliver AM CH2
True
CH1.CreateSliver <- CH2
CH.delegate_CreateSliver <- CH1
AM.delegate_CreateSliver <- CH
AM.delegate_CreateSliver <- AM.delegate_CreateSliver.delegate_CreateSliver
AM.CreateSliver <- AM.delegate_CreateSliver.CreateSliver

$ python ./proof.py --principal CH3 --attr AM.CreateSliver AM CH3
False
```

The proofs show that `CH2` has been delegated the right to create slivers, but not the right to delegate that to `CH3`. Furthermore the proof of `CH2`'s right shows that it has been delegated the right by `CH1`, an intermediate that does have delegation privileges.

7 Conclusions

We have presented an implementation of the SFA/GENIAPI authorization system in the ABAC attribute-based access control system. We believe that the unified expression of ABAC makes the authorization rules easier for developers to understand, especially GENI credential delegation and avoiding the GENI credential types/rights indirection.

The code packaged with this document realizes this encoding in a direct manner suitable for both exploring the rules that define access and for inclusion in working prototypes.

We believe this is a first step in moving to both a more universally understood expression of GENI policies using ABAC logic, and a step in creating clearer more powerful policies.

1 Appendix: Formal Delegation Proof

Given:

```
1) AM.clearinghouse ← (AM.clearinghouse).clearinghouse

2) AM.clearinghouse ← CH

3) CH.clearinghouse ← CH1

4) CH1.clearinghouse ← CH2

5) AM.CreateSliver ← (CH.clearinghouse).CreateSliver

6) CH2.CreateSliver ← R
```

Show `AM.CreateSliver ← R`.

Line	Proposition	Reason
1	AM.Clearinghouse ← CH2	Given 1) & Given 3) & Given 4)
2	AM.CreateSliver ← CH2.CreateSliver	Line 1 & Given 5)
3	AM.CreateSliver ← R	Line 2 & Given 6)

2 Appendix: GENI Credential Rights as ABAC

While we prefer using attributes that directly reflect the operations they authorize, the indirect assignment of operations to rights to types described in Section 2.2 is directly expressible in ABAC. Rather than reproduce the entire mesh of assignments in Table 1 and Table 2 we encode one type, slice, below. We use the specific GENI operations permitted rather than the more abstract SFA terms in Section 2.2. Some of those SFA operations are as yet unmapped to the GENIAPI, and we omit them.

A slice credential confers powers over a specific slice, so some of the attributes we encode are parameterized by the target GID, which refers to the slice. That GID is represented by `target` below. If we used RT1 logic, some of these credentials would be simplified; we use RT0 to indicate the encoding is possible in the simpler logic. To confer the powers of a slice credential with target GID `target` to user `U` from slice authority SA, it issues the following ABAC credentials:

First the refresh rights:

1. `SA.Remove_target ← SA.refresh_target`
2. `SA.RenewSlice_target ← SA.refresh_target`

Embed (skipping ticket operations, that are absent from the GENIAPI):

3. `SA.CreateSliver ← SA.embed_target`
4. `SA.RenewSlice_target ← SA.embed_target`
5. `SA.Remove_target ← SA.embed_target`

Bind (none of these have GENIAPI equivalents, but as a placeholder):

6. `SA.LoanResources_target ← SA.bind_target`

Control:

7. `SA.CreateSliver ← SA.control_target`
8. `SA.RenewSlice_target ← SA.control_target`
9. `SA.Remove_target ← SA.control_target`

Info:

```
10.      SA.DiscoverResources ← SA.info_target
```

Then binding the rights to the type

```
1. SA.refresh_target ← SA.slice_target

2. SA.embed_target ← SA.slice_target

3. SA.bind_target ← SA.slice_target

4. SA.control_target ← SA.slice_target

5. SA.info_target ← SA.slice_target
```

And finally, assigning all of these to the source (holder):

```
1. SA.slice_target ← User
```

Hopefully the process of encoding other types is clear. Hopefully the more direct encoding of operations is more attractive.

References

[1]The GENI API, http://groups.geni.net/geni/wiki/GeniApi.

[2]Larry Peterson, Robert Ricci, Aaron Falk, Jeff Chase, Slice-Based Federation Architecture, July 2010, http://groups.geni.net/geni/wiki/SliceFedArch.

[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).

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

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

[6]Ted Faber, John Wroclawski, Design and Integration of ABAC and the GENIAPI AM: Version 1,*http://groups.geni.net/geni/attachment/wiki/TIED/ABAC_GENIAPIv1.2.pdf*, January 2011.

[7]ProtoGENI, *http://www.protogeni.net/trac/protogeni/wiki*.