**ABAC Credentials Report**

March 25, 2011

Darwin Witt

GENI credentials operate on assertions by a trusted principal that a second principal has a particular set of rights to a target principal. A GENI credential maps certain abilities to a source GID with respect to a target GID based on the credential's type.

GENI credentials each have a distinct type that allow it a certain set of rights that in turn allow the holder of the credential to perform a certain set of operations. The following tables of type to right to operations relationships are based on the GENIAPI v1.2. Note that some of the types and rights share the same name, but are of distinct name spaces.

| Type | Rights |
|---|---|
| user | Refresh, resolve, info |
| sa | Authority, sa |
| ma | Authority, ma |
| authority | Authority, sa, ma |
| slice | Refresh, embed, bind, control, info |
| component | operator |

**GENI Credential Types & Rights**

| Rights | Operations Authorized |
|---|---|
| Authority | register, remove, update, resolve, list, getcredential, * |
| Refresh | remove, update |
| Resolve | resolve, list, getcredential |
| sa | getticket, redeemslice, redeemticket, createslice, createsliver, deleteslice, deletesliver, updateslice, getsliceresources, loanresources, startslice, stopslice, renewsliver |
| Embed | getticket, redeemslice, redeemticket, createslice, createsliver, deleteslice, deletesliver |
| 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 | settrustedcerts, getgids |

## GENI Credential Rights & Operations

The principal that performs an operation first confirms that it is the target and that the requester is the subject. Next, the type of the GENI credential is translated into the set of rights and the rights into a set of valid operations that can be conducted. Finally, the principal confirms that the operation in question is in this valid set before proceeding. This authorization process exemplifies a multi-level hierarchy that may be more complicated than necessary.

ABAC is an attribute based authorization system developed at Stanford and Network Associates Laboratories 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.

ABAC has two versions of logic, RT0 and RT1, the major difference being that parametrization of attributes is allowed in RT1. This will become more clear after a more in depth discussion of the RT0 logic. The following discussion and examples are based on the code implementation libabac v 0.1.2.

Libabac uses SHA1 hashings of the principal's public-key, usually referred to as a key identifier in the libabac documentation, as that principal's identifier. These principals are presented symbolically in the logic examples to avoid using a long string of hex digits.

An attribute is a string attached to a principal by another principal. The following example shows a statement where an aggregate manager (`AM`) has attached its `ListResources` attribute (`AM.ListResources`) to a user (`U`).

```
AM1.ListResources ← U
```

There are three distinct methods to attach an attribute to a principal: direct assignment, delegation, and linked delegation.

Direct assignment is simply when a principal binds one of its attributes to another principal. The example above of `AM` attaching `AM.ListResources` to `U` demonstrates this action perfectly.

Delegation occurs when a relationship between principal's assignments wishes to be made. An example is a delegation where all principals with attribute `AM2.ListResources` have `AM1.ListResources`.

```
AM1.ListResources ← AM2.ListResources
```

The effects of this delegation can be seen when combining this statement with a direct assignment statement concerning `U`. Any principal that knows `AM2.ListResources ← U` and `AM1.ListResources ← AM2.ListResources` knows that `U` has `AM1.ListResources`.

Lastly, linked delegation delegates an attribute to principals that have an attribute assigned by a designated principal, but not to the designated principal itself. An example is as follows: Any principal that has attribute `ListResources` assigned to it by a principal with the `AM2.Linked` attribute has the `AM1.ListResources` attribute. In the notation used to express ABAC statements, this translates to:

```
AM1.ListResources ← (AM2.Linked).ListResources
```

The effects of such a statement are a little harder to see. 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. Generally, this allows a principal to appoint other principals to assign an attribute on its behalf.

This logic also allows for conjunctions on the requirements, or right-hand-side, of a delegation. For example, AM.CreateSlice CH.CreateSlice & SA.CreateSlice means that AM will assign the AM.CreateSlice attribute to a principal that has demonstrated that it has both CH.CreateSlice and SA.CreateSlice.

Note that each of these statements are represented in the system as credentials, signed by the principal whose identity is attached to the attribute on the left-hand-side. These credentials in turn form the basis of proofs in the ABAC system, and are understandable by any entity that can confirm the signatures.

RT1 has the extended ability to parametrize attributes, therefore allowing for the attribute AM.DeleteSliver(uuid)instead of the RT0 statement AM.DeleteSliver_uuid which in turn carries with it the penalty of having to create a new attribute for each instance of a sliver having unique uuids to identify them.

In essence, from these delegations and their implied proof structure, we can derive sets of delegation credentials for each principal in GENI. We would then arrive at a new foundation of allowed behaviors distinct from the previously stated Types, Rights, and Operations tables at the beginning.

**Clearinghouse**

1. CH.GetCredential ← P
2. CH.Register_user_UUID ← P
3. CH.Register_slice ← P
4. CH.Resolve ← P
5. CH.ListComponents ← P
6. CH.Remove_UUID ← CH.admin
7. CH.Shutdown_UUID ← CH.admin
8. CH.List_slices ← CH.admin
9. CH.List_users ← CH.admin
10. CH.List_authorities ← CH.admin
11. CH.List_components ← CH.admin

Where P is any general user and CH.admin denotes assigned administrators of a clearinghouse.

**Slice Authority**

1. SA.clearinghouse ← (SA.clearinghouse).clearinghouse
2. SA.clearinghouse ← CH
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
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
14. CH.Register_slice ← SA
15. SA.Shutdown_UUID ← CH
16. SA.Shutdown_UUID ← (SA.clearinghouse).shutdown
17. SA.Shutdown ← (SA.clearinghouse).shutdown

**Aggregate Managers**

1. AM.slice_authority ← (SA.slice_authority).slice_authority
2. AM.slice_authority ← SA
3. AM.ListResources ← (AM.slice_authority).DiscoverResources
4. AM.CreateSliver ← (AM.slice_authority).CreateSliver
5. AM.DeleteSliver_UUID ← P
6. AM.SliverStatus_UUID ← P
7. AM.RenewSliver_UUID ← P
8. AM.Shutdown_UUID ← P
9. AM.Shutdown_UUID ← (AM.slice_authority).shutdown

Alternatively, the previously established GENI Types, Rights, and Operations system can be translated into ABAC; however, this is not necessarily the best way to utilize ABAC logic due to the hierarchical implications of the established system. Below is an example of the slice type encoded in ABAC statements. Note that a slice credential is for a particular slice, therefore some of the attributes are parametrized by the GID, afterword referred to as `target`, of the slice.

First, encoding each right -

Refresh:
1. SA.Remove_target ← SA.refresh_target
2. SA.RenewSlice_target ← SA.refresh_target

Embed :
3. SA.CreateSliver ← SA.embed_target
4. SA.RenewSlice_target ← SA.embed_target
5. SA.Remove_target ← SA.embed_target

Bind :
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 principal holding the credential:

1. SA.slice_target ← User