Security Infrastructure for Hybrid Clouds and Cloud Federation


Center for Cloud Computing and Big Data, Department of Computer Science PES Institute of Technology Bangalore, India

Abstract

Hybrid clouds are increasingly becoming important in cloud computing. We see a rapid raise in the demand for a secure infrastructure that would enable sharing of computing resources between multiple hybrid cloud deployments to facilitate accommodation of situations where the demand outstrips supply, load balancing, and other such infrastructure constraints. From the end user perspective, this would also mean that the end users can host applications with their choice of federated cloud provider, as opposed to choosing from a host of global cloud providers on the market. The following paper describes a federated infrastructure for hybrid clouds, in particular, federation between Openstack and Openstack and also between Openstack and Amazon.

1. Introduction

Cloud computing, has become widespread recently due to its ability to create a datacenter consisting of a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management and time. Though cloud computing was first popularized by companies such as Amazon, security and other concerns have led to wide deployment of private clouds owned by a single enterprise. Such clouds may need to acquire resources from public clouds to accommodate sudden spikes in demand, for experimental access to specialized software, or for outsourcing non-mission critical functions. This has given rise to hybrid clouds, which federate private and public clouds [1].

While our work is general and applicable to all types of clouds, in hybrid clouds there generally is a primary cloud owned by the enterprise, and multiple Remote Clouds owned by service providers or partners [2]. In such cases, it is preferable to extend the capabilities of the primary cloud to allow federation, instead of inventing a completely new infrastructure. Additionally, it is desirable that the solution be general enough to allow federation with any Remote Cloud. The aim of this paper is to address the problem of providing secure architecture for federation between Openstack and other Openstack clouds and also between Openstack and Amazon. The mechanism for federation presented in this paper is to be submitted to Openstack so that it may be incorporated in their subsequent releases.

The federation blueprint that we have used in implementing the project has been presented in the Openstack Summit-2013 held in Portland (This work was supported in part by a grant from EMC), Oregon. In addition to this, we have extended the architecture to allow federation with Amazon EC2, and this work is underway.

The rest of this paper is structured as follows. Section II presents related works in this domain. Section III describes the federated security for hybrid cloud architecture whilst section IV presents the way in which this architecture is implemented in openstack. Finally, Section V discusses extending this architecture to federate Amazon.

2. Related Work

Hybrid cloud support is an active area of research. In this section we survey some of the important work in this area. Many approaches to federated security focus on the use of plug-ins for managing access to different clouds. Examples of this include the built-in EC2 support in Openstack [3]. However, these are low-level APIs that require the user to explicitly manage their credentials on different clouds, unlike our approach. Eucalyptus has a security infrastructure that would facilitate the building of private and hybrid clouds that are compatible with Amazon Web Services [8]. However, this is not generalizable. An alternative model for Openstack federation is in [5]. However, this model, unlike ours, requires significant re-writing of the current Openstack security infrastructure. It also requires the creation of transitory user-ids in the Remote Cloud and is specific to Openstack. Finally, the mechanism described in relies upon Single-Sign-On, which is not required in our system [1].

A number of papers have examined different aspects of the security federation problem. Different models for implementing resource-sharing between tenants have been proposed. We have leveraged the resource sharing model in [4], which specifies the use of a 5-tuple for an authorization database. As described later, we have simplified this to the use of a 4-tuple. The importance of trust in the cloud security paradigm is discussed in [6].

The authors use a risk analysis approach to measure the trust levels. With an increase in risks, there is a reduction in trust.
In our work, we assume that trust has already been established between the clouds to be federated, since trust establishment involves legal and operational issues beyond the scope of the paper. A Semantic Access Control Policy Language (SACPL) is described in [7]. This is basically a language that would outline access control policies, unlike a model for access control proposed in our paper. However, we could extend our access control with such a mechanism.

3. Federated Security for Hybrid Clouds

In the following section, we describe our solution for federated security. Following a description of the requirements, we provide a brief outline of the solution. We first describe the general solution, and then its implementation in Openstack.

3.1. Requirements

At a high level, the requirements for the solution are that it be extensible to arbitrary Remote Clouds, and that it extend the capabilities of the primary cloud. After describing these requirements in more detail in this subsection, the next subsection provides a brief outline of our solution.

Conceptually, the federation design has to ensure that controlling access to the remote cloud should be easy and secure i.e., it should not be too tedious for the user and at the same time, it should be secure enough to prevent any kind of threats to the applications/data in the clouds. For achieving this, our design uses the well-known concept of Role Based Access Control or RBAC. We extend the notion of role to provide remote access as below.

Conventionally, a role represents a number of privileges or rights a user has or actions they are allowed to perform in their primary cloud. For example, a user who has an 'Admin' role can take administrative actions such as viewing all tenants in their primary cloud. We extend this notion to say that two clouds that have already established a trust relationship with each other will grant certain privileges to certain roles; i.e., if cloud A and cloud B have established a trust relationship, then cloud B may recognize and grant certain privileges to certain roles in cloud A. This is a light-weight mechanism for ensuring remote access, since role definition is an inherent part of the security mechanism in most clouds. Also, since access in cloud B is defined by roles, adding or deleting users in cloud A does not require any synchronization or overhead in cloud B.

Another major requirement is that it is desirable that remote access should not imply that a cloud surrender control over part of its resources, as this will complicate security management. For example, if cloud A wants to access resources in cloud B, security policy management and total control over the resources of B should remain in B. In our design, this is achieved by requiring that although B recognizes roles from A, the privileges that a particular role in A has is defined by B.

The other requirement is that the implementation of the solution extends the capabilities of the primary cloud, which is Openstack in our case. This would make the federated infrastructure appear as a natural extension of Openstack.

3.2. Implementation

In the following, we assume that, as stated earlier, there are two clouds – a primary cloud, and a Remote Cloud, and that users of the primary cloud want to access resources in the Remote Cloud (see Figure 1). It is assumed that there are a number of tenants in both clouds. We also assume that trust relationships have already been set up between the primary cloud and the Remote Cloud, since the establishment of a trust relationship involves legal and operational issues that are beyond the scope of the paper.

Our implementation relies on gateways that are proxy processors that (i) relay requests from the primary cloud to the Remote Cloud and (ii) translate requests for resources to commands that are understandable by the Remote Cloud. Since a trust relationship has already been set up between the clouds, the gateways in the primary cloud and the Remote Cloud have already established a secure communication channel and are ready to receive requests from each other. It can be seen from the description below that the inter-gateway communication can consist of SAML calls.

To acquire resources, users in the primary cloud execute the following protocol. The exact implementation of these steps would depend upon the cloud in question and is discussed in the next subsection.

1. Remote Gateway acquisition: the user obtains a list of remote gateways (clouds) that they can access and then obtains access to any one of them.
2. Remote Tenant acquisition: for each remote cloud, the user obtains a list of tenants that they can access and obtains a scoped token to one of the tenants.

3. Remote Resource acquisition: the user sends a request for obtaining a particular resource belonging to a particular tenant in the remote cloud that he is trying to access.

4. Openstack Implementation

The existing Openstack implementation does not support Federation between two Openstack installations. In Openstack, access control is via tokens that encode the access rights of the users. These tokens and access rights are validated by an access control process called Keystone. In order to acquire resources, users pass these tokens to the appropriate control process. Before granting access to the requested resources, the control process invokes Keystone to validate that the user has access. This mechanism is used to solve the problem of revoking tokens.

Since Keystone is the central access control mechanism, we have implemented our federation mechanism as extensions to Keystone, i.e., as separate APIs. This allows our extensions to be compatible with versions of Openstack that do not support Federation. In the following, we describe the Openstack components and our extensions in more detail.

4.1. Keystone

The policy service in Keystone provides a rule-based authorization engine and the associated rule management interface. The credentials of the person making the access to the resource, the current state of the system and the resource being modified is identified and sent as parameters to the policy engine of Keystone. All the checks to validate the policy against the given rules are made here.

The rules in the current Keystone architecture can be expressed in two different ways:

1. List of lists representation: every rule is represented by a list and the lists are concatenated to form a list of lists.

   For rules_dict:{abc:{role:[netadmin],
tenant_id:[mytenant]}
def: {role:[computeadmin],
tenant_id:[mytenant1]}

   This says that for the rule abc to be satisfied, the user has to be a “netadmin” in the tenant “mytenant”.

   Similarly, for the rule def to be satisfied, the user has to be a "computeadmin" in the tenant “mytenant1”.

2. Policy language representation: In this, every rule is separated by conjunction operators.

   For e.g.: {abc:{role: admin} or {(project_id :%{project_id}: s and role: projectadmin)}}

   In this form of implementation, concatenation operators are used to join the rules instead of implementing it using a list of lists. In the above rule, either the user should be an admin or he should be a “projectadmin” in the projectids that are mentioned in the rule.

   The service that the user is trying to access determines which rules are to be satisfied.e.g. Consider a service A that has the following match list: match_list : [rule:abc].This means that any user who is trying to access the service A has to satisfy the rule abc that is mentioned in the rules_dict. The service that has to grant access to the user calls stores the rules that are to be satisfied by him in the form of a dictionary called the target dictionary. This is passed to the policy engine as a parameter along with the user credentials/role in RBAC implementation. The policy engine then checks the validity of the users credentials against the rules specified by the service. A typical call to Keystone by the service looks like this:

   ```
   target_dict={create_vm:[“role:computeadmin”],
   [“tenant_id :%( tenantid)s”, “role: foreignAdmin”]},
   add_network=[“role:nova_netadmin”]}
   match_list=create_vm
   ```

   Call to Keystone will be of the form: authenticate(match_list,target_dict, user_credentials)

   This tells Keystone to check the credentials of the user against the rule create_vm in the target dictionary. The rules dictionary is stored in JSON format in the Keystone database.

   In the list of lists representation, each check inside the innermost lists id combined with an AND conjunction and all the outermost lists are combined with an OR conjunction. This means that any one of the outermost lists should satisfy the rules and that all the innermost checks in that list have to satisfy the checks.

4.2. Keystone Extensions

In our implementation, the policy engine is RBAC (Role Based Access Control) enforced. i.e., the user’s role is looked into when he is trying to access a service and not his attributes. The rules dictionary is modified accordingly to support RBAC.

An example policy in our implementation is as follows:

   ```
   {create_vm: [[“role: computeadmin”], [“tenant_id :%( tenantid)s”, “role: foreignAdmin”]]}
   ```

   This means that if the user wants to create a VM, then he has to satisfy the rule “create_vm” i.e. He either has to be a computeadmin or he should be a foreignAdmin and has to belong to the tenants listed in the rule.
The policy engine in our implementation is extended to support RBAC. Every service is extended to support remote access i.e. every rule in the rules dictionary is modified to support a role of “foreignAdmin” and the filtering is done based on the tenantids in the remote cloud. We have a method called add_policy() which enables the admin user to add a policy to the existing database. We make use of this particular method to enable the remote user authentication. For example to enable the user to create a Virtual Machine in the remote cloud, a rule “create_vm” (as shown above) is added. For every service the user tries to access, his role in the tenant he belongs to is extracted from his Tenant Acquisition Token (acquired before requesting a resource and described below). This is passed to the policy engine (Keystone), which then authenticates it by checking it against the rules that are to be satisfied for that particular service.

The foreignAdmin role is one of the roles that is framed to be issued to a user once he chooses to login remotely into a Remote Cloud. The gateway has the right to issue a role to the user based on his requirement in the remote cloud and also based on the trust relationship between the Primary and the Remote cloud. The user uses the role issued by his gateway while accessing the Remote Cloud (in this case foreignAdmin) and the policy engines of the services that he is trying to access in the remote cloud authenticate the user based on the foreign role.

Instead of passing the user credentials to the policy engine as in the existing implementation, we extract the role of the user and pass it to the policy engine for authentication, which on successful authentication grants the service access to the user.

The gateway needs to maintain a list of resources accessible by users. Since the gateway in this case is integrated with Keystone, we have modified the access control list maintained by Keystone to include the rights of remote users. Specifically, Keystone maintains a 3-tuple (Subject, Privilege, Object) which specifies the Privilege a Subject has on an Object. Subject can be a user or a role. However, for a RBAC implementation, it is likely that Subject+ will be the name of a role. We have extended this to a 4-tuple (Cloud, Subject, Privilege, Object) where the Cloud field is non-blank for remote users and specifies the cloud that the user belongs to. The 4-tuple mechanism enables the user to have a remote access to a Remote Cloud since the authentication system uses the cloud field in the credentials to authorize the user access to the remote cloud. Any unique attribute of the cloud can be used; in our case, the URL for the gateway is used.

For example, ((Cloud A,admin),Read,.) is interpreted as follows: Anybody who is an admin in cloud A has read access to the root folder of cloud B assuming that cloud B is the remote cloud and there exists a trust relationship between the two clouds.

Figure 2. Existing Openstack Access Control Architecture

4.3. Existing Openstack Access Control

Existing Keystone architecture provides authorization by means of tokens. All tokens are dictionaries within Python.

1. At login, the user provides his credentials (username and password) which are sent as a GET request to the Keystone URL (Current IP:5000/v2.0/tokens/). This request returns an unscoped token in JSON format. This token is not associated with any tenant and is temporary. This is represented by the step 1 and 2 of Figure 2.

2. Using this token, the user may request for a list of tenants that he has access to. This can be accomplished by issuing a GET request to another Keystone URI (Current IP:5000/v2.0/tenants/). This request returns a list of services that a user in that tenant may avail of. This is represented by the steps 3 and 4 of Figure 2.

3. Once the user obtains a scoped token, he may use it to request creation of a virtual machine at a specific endpoint. Keystone then verifies the validity of the token and whether or not the tenant has access to the specified service. It may also return some additional information to the user. The request is validated against the service’s policy framework. If all requirements are satisfied, a virtual machine is created and status is returned to the user. This is represented by the steps 5 and 6 of Figure 2.
4.4. Openstack Federation Extensions

We now describe how the above protocol is executed in Openstack as represented by Figure 4:

1. Remote Gateway Acquisition: as stated above, Openstack access control is via tokens that encode the resources that users are authorized to access. Therefore, it is necessary for users to get a token (referred to as a GAT or Gateway Acquisition Token) that allows them to access the gateway in the remote cloud. Since in Openstack, the access tokens are obtained at login, the user obtains the GAT via a background call to the gateway in the primary cloud which in turn contacts the remote gateway so as to verify the user credentials from the NEWROLES table present in its database. Upon successful authentication, it returns a GAT (Gateway Acquisition Token) to the user and at the same time updates the GATS table so that the GAT can be validated at a later time.

The authentication and validation process has been organized into different functional components like assignment, auth, catalog, common, contrib, credential, identity, locale, middleware, openstack, policy, tests, token, trust. Each of these except a few like the middleware, locale have routers.py and controllers.py files. Each of these according to the url will match to the actions in its corresponding controllers.

For example, if ‘../user/tokens/’ is the specified url, then routers.py residing in the ‘token’ directory will be called and controllers.py inside the token directory will have corresponding matching action.

In this step of gateway acquisition, identity controller comes to play and also the token controller needs to be invoked for authenticating user credentials and returning an unscoped token. This is accomplished by first calling the auth controller which in turn invokes the identity controller and token controller.

The Auth controller is used to match corresponding controller and action from pattern in url. Some of the typical examples of how the url would be are .../auth/tokens, .../auth/credentials. The auth controller would in turn call the token controller for the former and credentials controller for the latter case.

Functionalities involving identity related information such users, groups, and group membership are provided in the Identity controller.

Token controller performs functions like authenticating user credentials and returning an unscoped token, returning of scoped token for a particular tenant, validate an existing token, authenticate against an identity backend.
2. Remote Tenant Acquisition: once the user has the GAT, they need to obtain a token that specifies the list of tenants they can access in each remote cloud. This could also be accomplished via background calls at login. However, for efficiency and security, in our design, these tokens are acquired when the user wishes to access a remote tenant. At that point, the user sends the GAT to the foreign gateway. The foreign gateway validates the GAT from the GATS database, then generates a list of tenants that the user can access. Once, the user has chosen a particular tenant, the gateway returns the TAT or Tenant Acquisition Token for that particular tenant after making the background calls to Keystone of the remote cloud.

This phase involves validation of existing token, to be performed by the token controller. As mentioned previously, this is achieved by a call to the auth controller which then calls the token controller. The function of listing all the tenants that a user could belong to, associating user with a particular tenant , assigning a particular role to user is done through a call to the assignment controller via the auth controller.

Assignment controller deals with getting valid tenants for a token, granting a role to user or a group on either domain or project, revocation of role granted to user/group on domain/project, adding a user to tenant, removing a user from a tenant or deleting the contents of a domain.

3. Remote Resource Acquisition: the user then sends the TAT with a request for resources to the Remote Cloud. Before granting the request, as before, the appropriate Openstack process invokes Keystone to validate that the user has access. The modified Keystone recognizes that this is a remote request, and invokes the gateway to validate the request. The Gateway validates the user credentials obtained from the TAT and uses RBAC to authenticate the same for the resource requested. Upon successful validation, the gateway grants the resource access to the user. Once the access is granted, the user can access the resources directly without having to go through the authentication processes.

This requires a call to token controller to validate an existing token, and a call to policy controller.

Policy controller is the Controller for enforcing policy engine, also an interface for getting, updating and deleting policy. The rules are defined here.

4.5. Gateway and Gateway API

We have the Gateway as the main component of our model, since all the communication between the clouds as well as the authentication is done by the gateway.
the user by verifying his credentials. It also updates the GATS table with the appropriate values so that the GAT verification can be done as and when required.

5. TATcreate: This has two functions namely TATlist and TAT_create. TATlist returns the list of accessible tenants for the given user after verifying the GAT and TAT_Create generates the Tenant Acquisition Token (henceforth known as the TAT) for the chosen Tenant.

4.6. Description of APIs

The following provides a description of the APIs that were introduced in the previous section:

1. Gateway: This API is called in the phase 1 which is Remote Gateway Acquisition phase where the user acquires the GAT through a call to the gateway in the primary cloud which in turn contacts the remote gateway and upon successful authentication returns GAT.

2. Prq: This API is called whenever the client makes requests to the gateway in order to obtain the list of accessible gateways, acquire GAT from a particular choice of gateway, procure the list of tenants, and obtain TAT for the chosen tenant. Prq then forwards the request by calling the appropriate APIs. Hence this API is called in all the three phases, Remote Gateway Acquisition, Remote Tenant Acquisition, and Remote Resource Acquisition.

3. GATList: This API is called during Remote Gateway Acquisition phase by the Prq API, when the user wishes to get a list of clouds that are accessible from the primary cloud.

4. Remotetauth: This API is invoked in the Remote Gateway Acquisition phase, in order to grant the Gateway Acquisition Token (GAT)

5. TATcreate: The call to this API is during the Remote Tenant Acquisition phase. When the gateway receives a request to either return the list of accessible tenants for the given user or request for the Tenant Acquisition Token for the chosen Tenant, the Prq API invokes the TATcreate API.

In addition to the APIs and the existing Keystone database illustrated in Figure 6, we have a separate database which contains the following tables that enables us to validate the user access in the remote cloud as shown in Figure 7.

1. ABLECLDS: This stores the list of the gateways that have a trust relationship with the primary cloud. It contains of the fields, Tenant_ID and GatewayIP. TenantID is used so as to provide a tenant filtered access control to the remote clouds. GatewayIP contains the IPs of the remote clouds that are accessible for the given tenant.

2. NEWROLES: This database is present in the remote cloud. This contains Issuer, Tenantid, role, NewTenantID and ForeignRole. The issuer, TenantID and role fields are used to validate that the user belongs to a particular tenant in the primary cloud and has some pre-defined role. Based on this, he is given a list of tenants that he can access in the remote cloud and is provided with a new role (Foreign Role).

3. GATS: This database, present in the remote cloud is used to verify the GAT that has been issued to the remote user. This contains the fields GATID and STOKEN. The GATID is unique for every GAT and hence it acts as the primary key. This field is used for authenticating the GAT.

Amazon Web Services has seen a large number of customers that is constantly on the rise, typically Amazon is estimated to have 80% of the cloud market, and hence is a major player. In the current scenario, with support for OpenStack gaining more impetus, it would be useful to design a system where OpenStack and Amazon coexist. A customer may have an account in Amazon and another with an OpenStack deployment and may find it worthwhile to link the two. In this section, we propose a system that supports federation between Openstack and Amazon.

Amazon federation can be integrated into our federation model using the Amazon Web Services Security Token Service (AWS STS). It is a web service that enables a cloud service to request temporary, limited-privilege credentials for federated users. Federation with Amazon requires the extension of the current implementation of the gateway. The credentials of the user in the primary cloud are authenticated. When the user requests access to Amazon, he is directed to the gateway. The gateway then authenticates with Keystone and returns an SAML assertion. The user contacts STS with this SAML assertion. STS returns a set of temporary credentials (similar to GAT).

Figure 6. Amazon federation
the Openstack-Openstack federation model. AWS STS returning temporary credentials is similar to obtaining Resource Acquisition Token (RAT).

The access rules supported by Amazon are very similar to but different from Keystone policies (described in IV.A). Access rules here refer to a JSON document that specifies the access rights for federating users. It contains the following fields – Action, Resource, Effect and Principal [13]. Effect specifies whether to allow or deny an Action on a Resource by a Principal. The Principal is generally a user. These access rules are used along with the API calls to enforce access control. This is similar to the model used for Openstack - Openstack federation. Action can be mapped on to a Privilege, Resource is analogous to Object, Effect is analogous to Rule and Principal maps to Subject.

Since OpenStack is written in Python, we are using the Python SDK for Amazon called boto. This has modules for various services provided by Amazon like EC2 and S3. There is similarly also a module for AWS STS which provides the necessary APIs namely get_federation_token, assume_role and assume_role_with_saml.4 Similar to the policy engine of Openstack.

6. Conclusion

We have proposed a secure infrastructure which allows secure sharing of resources between Openstack and Openstack and also between Openstack and Amazon. This showcases the generality of the proposed scheme. In addition, our architecture also has the desired property of having the control of resources in the remote cloud. The federation design also ensures that controlling access to the remote cloud is easy and secure.

7. References