Security APIs for My Private Cloud
Granting access to anyone, from anywhere at any time

David W Chadwick
School of Computing
University of Kent
Canterbury, UK
d.w.chadwick@kent.ac.uk

Matteo Casenove
School of Computing
University of Kent
Canterbury, UK
m.casenove@gmail.com

Abstract—We describe a set of security APIs that grant federated access to a user’s cloud resources, and that also allow the user to grant access to his resources to anyone from anywhere at any time. The APIs implement federated access to clouds, fine grained access controls and delegation of authority. We have integrated these APIs into two cloud applications in order to validate their utility. This paper describes the conceptual model and architecture of the APIs, as well as their integration into the Eucalyptus S3 service. The paper concludes by specifying the current limitations.

Keywords—Cloud Computing; Federated Identity Management; Delegation Of Authority; Levels Of Assurance; Attributes; Access Rights, Authorisation

I. INTRODUCTION

Cloud computing provides some excellent features [1] such as resources on demand, scalability, optimization of resources etc. Some threat models assume that the cloud provider can never be trusted, and therefore propose storing only encrypted data in the cloud. Others assume that the cloud provider can be trusted (to a certain extent), and that the threats come primarily from outside attackers and other cloud users. Given that most public cloud services are currently being run by large relatively trustworthy organizations such as Amazon, IBM, Microsoft and Google, we believe that the latter threat model is reasonable for many users, especially when they are given the tools and services to provide with the assurance they need about the provider’s trustworthiness. Furthermore, organizations are now able to run their own private clouds, using open source software such as Eucalyptus and Open Stack, hence the trusted cloud service provider model is the most appropriate one for this scenario.

Unfortunately, the security and privacy features of existing cloud services and software are currently not as well designed as they should be. In the case of storage services, having all the data in a single place without having a strong fine grained privacy policy does not encourage users to choose cloud computing as a good solution to meet their needs. However, once we acknowledge that cloud providers can be trusted (to a certain extent), we can start to build trust, security and privacy preserving infrastructures that rely in part on trusted cloud providers to operate parts of the infrastructure for the users. We can then turn our attention to the problems of designing and building tools and services for a large scale trusted federated privacy preserving cloud infrastructure with fine grained access control and delegation of authority. In this paper we present the design and implementation of a set of three security APIs that provide these features: federated access to cloud resources, fine grained access controls and delegation of access rights. Specifically, by building extensively on our previous research, these APIs:

- Allow cloud users to set their own fine grained controls for access to their data whilst it is in the cloud [2];
- Allow cloud users to delegate access to their resources to any other users of their choosing [3];
- Provide federated access [4] to cloud services and users’ resources, thereby not constraining the set of users to those only known to the cloud service provider;
- Include the level of assurance (LoA) [5] in the authentication assertion so that access decisions can be based on this [2];
- Allow requestors to aggregate their roles and attributes from multiple sources of authority so that finer grained controls can be created [6];

The rest of this paper is structured as follows. Section II presents related research and supporting technologies. Section III presents the conceptual design of the APIs whilst section IV describes the system architecture. Section V describes the security APIs, whilst Section VI describes how the Security APIs have been integrated with the Eucalyptus S3 service. Finally, Section VII discusses the current limitations of the Security APIs.

II. BACKGROUND TECHNOLOGIES AND RELATED RESEARCH

Federated identity management, for example, as typified in the Shibboleth implementation [7], comprises an Identity Provider (IdP), Service Provider (SP) and user agent (usually a web browser). The user attempts to access the SP via his user agent, but the SP, not knowing who the user is, redirects the user to its IdP for authentication. If the SP supports multiple IdPs, then it may first ask the user to choose which IdP he wants to use for authentication. When the user’s agent contacts the IdP, the user is asked to authenticate, and if successful, the IdP redirects the user back to the SP, providing the agent with digitally signed assertions to present to the SP. These assertions say that the user has been authenticated by the IdP, and that he has the attached identity attributes. These attributes may include a unique persistent identifier (PId) for the user, so that the SP can provide a personal service for the user on repeated visits, and so that
each user can be uniquely identified for audit purposes. When we apply this model to cloud services, the cloud service provider (CSP) becomes the SP, and the IDP is any IDP from any existing federation to which the CSP belongs.

Delegation of Authority allows a user (the delegator) to delegate any of his attributes to another user or application of his choice (the delegate) [3]. This form of delegation is superior to existing grid based delegation that relies on proxy certificates [8], since i) the delegate must authenticate as himself, and not as a child of the delegator as in proxy certificates (which is a form of masquerade) and ii) fine-grained delegation is automatically supported by the delegator delegating a subset of his attributes. Our system permits a resource owner to delegate access to anyone who is not an existing cloud user, by assigning them an attribute (e.g. MyFriend, for which the resource owner is the sole authority.

Current cloud platforms (e.g. Amazon, Google, Eucalyptus etc.) do not support federated access, delegation of authority, or fine-grained access control. They primarily use a simple Access Control List (ACL) to provide access to other cloud users. In general, these are coarse-grained. Controlling who might invoke a virtual machine is often restricted to everyone or no-one. Amazon's S3's ACLs only allows users to specify the following access levels: anonymous access (everyone, including non-Amazon Web Services (AWS) users), all AWS users, or named individual AWS users. All existing ACL based systems are limited in only being able to grant controlled access to other registered users. Amazon has recently extended its Access Control system with Bucket Policies [9]. These policies allow users to manage access to their S3 resources at the bucket level for both the buckets and the objects, providing a more fine-grained access control for those resources, but still limited to other registered Amazon users.

Different approaches to the cloud access control problem have been published in the literature. V. Echevarria et al. [10] have developed a novel approach called Permission as a Service (PaaS) which provides a separate access control service from the other cloud services. In PaaS, user data is encrypted in the cloud, using attribute based encryption (ABE), to maintain confidentiality. Permissions are managed via decryption keys based on the attributes of the users being granted access. Our approach is similar to PaaS, in that access is granted based on the attributes of the user, but there the similarity ends. We assume the cloud service provider can be trusted to keep the information confidential, so encryption is not mandatory (though the cloud provider or user can encrypt the data if they wish to). Each cloud resource owner determines which other users can access their resources, based on their IDP asserted or resource owner delegated attributes.

III. CONCEPTUAL MODEL

Conceptually the design should ensure that controlling access to cloud services is as easy as possible for cloud application developers to integrate into their cloud applications, otherwise they are likely to “roll their own security” which is known to be fraught with difficulties. Ease of use drives us towards providing the developers with a set of APIs that they can be directly integrated into their cloud services and applications, rather than giving them a set of standardised protocols that allow the various functional components to talk to each other. By providing APIs, the application developers are shielded from knowledge of the underlying protocols, and even the functional components that are used, as these can be standardized and modified as necessary without unduly affecting the actual APIs. A disadvantage of the API approach is that we have to either choose one programming language and require all developers to use this, or produce multiple implementations of the APIs in different languages, which is resource intensive. Given the resource limitations of the current project, we were obliged to choose one implementation language as a proof of concept, in the knowledge that if we were successful, other language implementations of the same APIs could subsequently be developed. We chose PHP.

Since the design is based on federated identity management, we should define what we mean by a digital identity. We base our definition on that in X.1250 [11] viz: “a digital identity is a set of attributes which allow the entity(s) to be uniquely recognized within a context to the extent that is necessary”. Thus in our model, a user will be identified by a set of identity attributes, issued either by an IDP from the federation, or a delegating resource owner, both of whom are trusted by the cloud application/service to assert those attributes. This set of attributes will then grant the user access to one or more cloud resources. The user may subsequently grant fine-grained access to his cloud resources, to other users in the federation, based solely on their identity attributes issued by either one or more trusted federation IDPs or himself.

The identity attributes are classified in a hierarchy, with the most generic attributes being subordinate to more specific ones, as in the X.500/LDAP model. An example of an attribute hierarchy might be: name>personal name(formal name. This makes it easier for cloud services to request attributes of certain types if they are not concerned which specific type is actually returned e.g. any type of credit card attribute or any type of employer attribute.

Similarly identity providers are classified by type, as in the X.500/LDAP object class hierarchy. An example class hierarchy might be: UK-AMF>UK University>Russell Group. Again this makes it easier for cloud services to request attributes from certain classes of IDP rather than naming each IDP specifically e.g. return a degree attribute from a UK university. The security API might then return a biochemistry degree attribute from the University of Kent.

It is recognized that different IDPs operate with different procedures and to different levels of trustworthiness or assurance. NIST has codified the amount of assurance one can have in an IDP authenticating a user, using the level of assurance (LoA) concept and metric [5]. The NIST LoA ranges from 1 (lowest assurance) to 4 (highest assurance). We have extended this concept by applying the assurance level to each identity attribute that an IDP releases. Thus an IDP can assert not only its LoA in authenticating a user, but also its LoA in each attribute that it is returning to the SP.

Traditional access control systems are built using
identifiers rather than identity attributes. Identifiers are special identity attributes, in that on their own, they uniquely identify a user within a context. Identifiers thus make access controls easy to manage and update, since each user can be identified uniquely using his identifier. However, we have a severe problem if we try to use this approach in federated systems. This is because IdPs do not use federation wide identifiers for users, but rather use pairwise PIds between themselves and each SP. These PIds are generally unknown to the users themselves, and typically comprise 128 bit numbers or similar computer generated unique IDs. Thus one user is not able to grant access to his resources to another user by specifying the other user’s identifier, since a) he does not know the identifier of the other user b) the other user probably does not know it either (of even if he does, it is not easily usable and communicable) and c) the identifier changes for each IdP/SP combination. We therefore base our access control system on identity attributes rather than identifiers, since users find attributes easy to understand, use and communicate to each other. They also have the added advantage that they can apply to single users or groups of users. For example, one resource could be made accessible to, or owned by, any user with the attribute emailAddress= d.w.chadwick@kent.ac.uk whilst a second resource could be made available to any user with the attribute affiliation= University of Kent. In the first case, only one user in the world would be able to access the resource; in the second case over 20,000 users would be granted access.

We have specified three cloud security APIs based on our identity attribute approach, so that developers can choose which of the three they wish to implement.

- 1. The authentication/identification API (Authn API). This is responsible for authenticating the cloud user and returning the user’s set of identity attributes.

- 2. The authorization API (Authz API). This is responsible for granting rights to cloud users, revoking rights from cloud users, and determining what rights cloud users have, based on their identity attributes.

- 3. The delegation by invitation API (Delegation API). This is responsible for assigning a freshly minted (by the delegator) identity attribute to any user (the delegate) who can successfully authenticate to the FIM infrastructure. The delegator subsequently determines which of the identity attributes are accessible to which freshly minted identity attributes by using the authorization API above.

The delegation by invitation service solves the problem that traditional delegation services, such as that described in [3] suffer from, namely, that the delegator identifies the delegate by his unique identifier. Since this is not possible in FIM systems, instead, the delegator assigns one of his own freshly minted attributes e.g. MyFriend, to his delegate, and then refers to the delegate by this identity attribute. The delegation process is described in more detail below.

IV. ARCHITECTURE

Behind each of the 3 APIs lie infrastructure components that provide the appropriate services to the API. Note that these infrastructure components can be changed, upgraded, replaced etc. without affecting the functional service that the API provides to the application (although the quality of the service may vary with different backend components).

The Authz API has a database that stores the mappings between the identity attributes and access rights, and an optional policy decision point (PDP) that can provide even finer grained control. It is currently not possible to dispense with the database because PDPs don’t have the ability to list either the access rights a user has, or the users who can access a resource.

The Delegation API has a Delegation Issuing Web Service that issues invitation tokens to delegates, and stores the identity attributes that have been delegated to them. New (previously unknown) delegates are automatically enrolled in this service without administrator involvement.

The Authn API acts as a federation SP which contacts the configured federation IdP asking it to authenticate the user and return his identity attributes. In our architecture we replace the (single) federation IdP with a proxyIdP that:

i) is capable of talking multiple IdP protocols such as SAML, OpenID, Twitter etc ;

ii) is configured with the level of assurance (LoA) of each trusted IdP;

iii) allows the user to choose which IdP he wishes to use for authentication;

iv) allows the user to link together his accounts at various IdPs in order to increase the LoA;

v) allows the cloud service provider to integrate its own corporate LDAP service in order to retrieve the identity attributes of its employees for finer grained access control;

vi) has a configurable Credential Validation Service (CVS) which contains policy rules saying which IdPs are trusted to issue which attributes to whom [12].

The design, implementation and user trials of the proxyIdP are described in [13].

V. API DESCRIPTIONS

A. Authn API

The Authn API has three methods, namely:

getIdentity – given a URL and a set of identity
requirements, return the authenticated user to this URL along
with: i) a set of qualified identity attributes that match the
requirements and ii) the user’s PId and name of the IdP
authenticating the user. If no identity requirements are
specified then obtain as many identity attributes as possible
from as many IdPs as possible.

logout – given a URL to return to, log the user out of his
session with this CSP and return the user to this URL. Note
that this is only logout of the cloud application and is not
logout from the FIM infrastructure, i.e. SSO with the
authenticating IdP is still active.

setCVS – given the URL of the CVS, this method
enables the use of the CVS by the proxyIdP. When this
method is called all the attributes returned from the IDP are
validated by the CVS according to the policy rules
configured into it. This method should be called before the
ggetIdentity method is used in an operational system.
However, during installation and testing it can be omitted.
but in this case gdentity will accept every attribute that it
is given without validating if they came from the correct (i.e.
trusted) IdPs.

The cloud service provider’s identity requirements are
specified in gdentity as tuples, each tuple comprising:
- An attribute type – taken from an attribute class
  hierarchy as described in Section III.
- An attribute issuer – which specifies who the issuer
  of the attribute type should be. This can either be a specific
  issuer instance (URL), or a class of issuer (URN) taken from
  an issuer class hierarchy as described in Section III.
- The minimum required Level of Assurance (in the
  range 1 (lowest) to 4 (highest)) for this identity attribute.

Each qualified identity attribute that is returned by
ggetIdentity comprises:
- An identity attribute type/name
- An attribute value
- The issuer of this attribute
- The LoA of this attribute.

The attribute type hierarchy and issuer class hierarchy
must be shared knowledge between the cloud application
and the FIM infrastructure.

An example of the use of the gdentity API is:

UID, kent.ac.uk, 2, role, urn:federation:UK-AMF, 1;
affiliation, urn:federation:UKAMF, 1.
This specifies that the
user should be identified by 3 attributes, namely a UID
issued by kent.ac.uk at LoA 2, a role issued by a member of
the UK Access Management Federation, and the name of
the organization in the UK-AMF to which the user is affiliated,
both at LoA 1. An example of a return value might be
(UID=dwc8, kent.ac.uk, 2), (role=professor, kent.ac.uk, 2),
(affiliation=University of Kent, kent.ac.uk, 2)

B. Authz API

The Authz API comprises 5 methods:

getRights – given a set of user identity attributes (types
and values), return the resources (identified by a set of
attribute types and values) and access rights that are granted
to users possessing this identity.

listAccess – given a resource (identified by a set of
attribute types and values), return the sets of users with
access rights to this resource, each set comprising a user
identity (a set of attribute types and values) and its associated
access rights.

addRights – given a set of user identity attributes (types
and optionally values), a resource (identified by a set of
attribute types and values) and a set of access rights, grants
these rights to users possessing this set of identity attributes
(in addition to any existing rights).

removeRights – given a set of user identity attributes
(types and optionally values), a resource (identified by a set
of attribute types and values), and a set of access rights,
revoke these rights from users possessing this set of identity
attributes. If the rights have already been revoked this is not
treated as an error.

authzDecision – given a set of user identity attributes
(types and values), a requested resource (identified by a set
of attribute types and values) and a requested access right,
return a Response object indicating whether access is granted
(GRANT) or not (DENY). The Response object can be
checked by using the method isGrant, which returns the
value True if access is granted.

When the set of user identity attributes is null, this
indicates that the user is anyone i.e. a member of the public.
In this way resources can be given public access rights.
When a user identity attribute type with no value is
presented, this means the right is granted (or revoked) to
users possessing any value of this attribute.

Whilst it is an application dependent issue what the set of
user identity and resource attributes are, normally one would
expect each resource to have a resourceID or name attribute
that uniquely identifies it.

The API does not restrict which access rights the cloud
application can use, provided they are encoded as a bitstring
(as determined by the cloud application). The use of
bitstrings makes manipulation of the rights by the Authz API
much quicker.

C. Delegation API

The Delegation API comprises eight methods:

encodeDelAtt – given the identity of the delegator (as a
set of attribute types and values) and the attribute to be
degradated (e.g. delegationAttribute=MyFriend), it returns the
(usually) encoded delegation attribute.

decodeDelAtt – given an encoded delegation attribute,
return the identity of the delegator (as a set of attribute types
and values) and the attribute that is delegated.

getSecret - given the identity of the delegator (as a set of
attribute types and values), the nickname of the delegate
(string), and the encoded delegation attribute, return a secret
to be given to the delegate.

useSecret - given a secret, the identity of the delegate (as
an IdP/PId pair), and the delegate’s nickname for the
delegator, return the encoded delegation attribute.

getDelegationAttributes - given the delegate's identity
(as an IdP/PId pair) return the set of encoded delegation
attributes, each set comprising: an encoded delegation
attribute and the delegator (as a set of attribute types and
values).

revokeDelegate – given the identity of the delegator (as
The overall approach was to use the APIs provided by the Eucalyptus S3 service and the security APIs described above, in order to build a new S3 application that provides federated access, fine grained access control and delegation of authority. We have called this new service the proxyS3 cloud service. The security APIs have been written in PHP.

Each S3 user is given his own KeyID and QueryID by the S3 service, in order to gain owner access to his resources (files and buckets). However, when providing federated access to S3, we do not want the user to be burdened with these additional credentials, since he already has his own IdP issued credentials for gaining access to all federated resources. Therefore the proxyS3 service stores all the users’ KeyID/QueryID pairs in an Owner table in its own secure storage and then accesses each user’s resources on the user’s behalf using the user’s own S3 credentials. The ProxyS3 service enforces the fine grained access controls by utilizing the security APIs, and only allows users to access what the Authz API says they are entitled to. For example, if an S3 user, nickname Fred, delegates access to his resource (say Bucket X) to his friend, nickname Bob, then Bob will be given a new attribute of Fred’s choosing (say MyFriends) by the Delegation API. Fred will grant MyFriends access to Bucket X. (Note that Fred may invite many people to be his friends, apart from Bob, and they will all get access to Bucket X.) Bob will attempt to access the proxyS3 service, and will be redirected to the proxyIdP to authenticate. There Bob chooses his preferred IdP, authenticates to it, and is redirected back to the proxyS3 with a unique PId/IdP tuple. The proxyS3 service contacts the Delegation API (getDelegationAttributes) and is returned the name of the delegator, Fred, and his delegation attribute MyFriends. It then contacts the Authz API (getRights) to see which resources Fred’s MyFriends have access to. Armed with this information, the proxyS3 service can now use Fred’s KeyID/QueryID pair to access Bucket X on behalf of Bob. Fred can remove Bob from his list of friends at any time, and add new people to the list at any time as he sees fit, by simply delegating and revoking the MyFriends attribute from them. Fred can also change which resources he wishes his friends to access at any time, by using the Authz API via the proxyS3 service.

B. Configuring the ProxyS3 Service

The proxyS3 service must hold an Owner table of KeyID, QueryID pairs, along with each owner’s nickname (for display purposes), and the identity attributes of each owner. This table should be held encrypted with a symmetric key that is only known to the proxyS3 administrator. The proxyS3 service must also be configured with information about the S3 service (url, port, etc), and the S3 admin key pair (admin QueryId, admin IdKey) which needs to be suitably protected.

The proxyS3 service also needs configuring with an Account Attributes table which lists the set of identity attributes that are to be used for creating S3 accounts. For example, suppose the administrator wants users from the University of Kent to be able to create personal accounts (identified by the uid attribute), shared accounts for different roles (identified by the role attribute), a permanent account for himself (identified by his email address), and shared accounts for roles at different universities (identified by the Role and university attributes), then the configuration file would contain the following 4 sets of attributes:

- university=kent.ac.uk+uid
- role
- emailAddress=fred.smith@kent.ac.uk
- role+university

C. Use of Authn API

Whenever a user tries to access the proxyS3 service this calls the getIdentity API which deals with the user authentication and identity attribute retrieval as described above. When getIdentity returns the user’s identity attributes, these are matched against the attribute sets in the Account Attributes table. The user is regarded as the owner of all the S3 accounts that were matched in the table. Next the proxyS3 service looks in the Owner table to see if any of the matched accounts currently exist or not. If they already exist, the user is asked if he would like to open the existing account(s), but if they do not exist, the user is asked if he would like to create them. For example, Table 1 shows the Access Accounts Welcome Page for a user for which getIdentity returned three identity attributes: role=staff, university= kent.ac.uk and uid=dwc8. The user is the owner of 3 accounts. One has already been created, one is not yet created, and one is pending creation. (Pending creation is a temporary glitch due to the current design of the Eucalyptus API that requires a human administrator to intervene in the
The attributes of the resource and the chosen access rights.

The proxyS3 service also calls getDelegatedAttributes, passing the IdP/PID returned from getIdentity. If a delegated attribute is returned, it extracts the delegator’s (owner’s) identity from the result value and displays this account to the user as well in Table 1, with the action Open.

The user can now choose which account he wants to open or create, including opening a delegated account.

D. Use of Authz API

The proxyS3 service is configured with the set of access rights that are to be supported, and their associated bit values e.g. Read Bit 1, Write Bit 2, Modify Bit 3 etc.

Once a user has opened an account, the proxyS3 service calls getRights passing it the user’s account attributes and any delegated attributes, in order to determine if any other cloud user has granted or delegated the current user access to any of their resources. The proxyS3 service can then display these to the user along with all the resources in the account for which he is the owner.

If the user (resource owner) wishes to grant access to any of his resources to other users, he selects the resource, then selects new permission, and is given the choice between granting public access rights to the resource, granting access to existing S3 account holders, or delegating access to his contacts. If the owner chooses either of the latter two, he is shown a picking list of the attributes that either characterize the other account holders or that he has personally created, such as MyFriends, Family, WorkMates etc. He picks the identity attribute(s) that is(are) needed to access his resource, then selects which access rights are to be given to this set of accessors. of contacts. The proxyS3 service then calls addRights, passing it the identity attributes of the chosen accessors (null in the case of public access), the attributes of the resources, and the chosen access rights.

If the user wishes to remove access rights from others, he selects the resource properties button, and the proxyS3 service calls listAccess. The returned list of accessors (identified by their identity attributes) along with their various access rights, is displayed. This list includes other cloud resource owners, the public, and the delegates, since the Authz API does not differentiate between these three sets of users. They are all simply users who have different sets of identity attributes. The owner picks the accessors whose rights are to be removed, including which rights should be removed from them (it may not be all of the rights), and the proxyS3 service calls removeRights passing it the identity attributes of the accessors, the attributes of the resource and the access rights to be removed.

E. Use of Delegation API

The proxyS3 is configured with details of the delegation issuing web service to be used, and the name of the attribute that is to be used to delegate access rights to anyone e.g. DelegationAttribute. This attribute name must be unique within the federation and not assigned by any other IdP, since users of this proxyS3 service will be responsible for assigning unique values to it.

If the resource owner (delegator) wishes to grant access to users (delegates) who are not S3 account owners, and who do not have attributes that can be asserted by federation recognized IdPs, then he may delegate them his own freshly minted attributes via the delegation API. The only requirement is that the delegate can be authenticated by a trusted federation IdP and obtain a valid PId/IdP pair, which the delegation service will use to register the delegate. Many federations may include “public” IdPs, which will allow anyone to enroll with them. Whilst these public IdPs may provide a single sign on service (SSO), they cannot provide trustworthy identity attributes. For example, Facebook and Twitter provide an SSO service, and anyone may enroll with them, but all the identity attributes they assert are user provided and therefore have zero level of assurance. However, these IdPs will always return the same persistent identifier (PId) for the same user, and therefore the CSP can be reasonably sure it is the same user each time, even if the user cannot be reliably identified.

In order to use the Delegation API, the resource owner chooses the My Contacts tab, and selects Add Contact. The proxyS3 interface then invites the delegator to enter two strings, the name for the (possibly new) group, and the name of the new group member e.g. MyFriends and Bob. The proxyS3 service calls encodeDelAtt, passing it the identity of the delegator and the delegated attribute e.g. DelegatedAttribute=MyFriends, and is returned the encoded delegation attribute containing both input parameters. The proxyS3 service then calls getSecret, passing it the encoded delegation attribute, the identity of the delegator and the nickname of the delegate (Bob). The Delegation API returns a secret to the proxyS3 service which it converts into a URL to point to its web page that is ready to process new delegates. The proxyS3 gives the secret URL to the delegator, asking him to give it to his contact (Bob). The delegator passes this URL to his friend (Bob) by some private means e.g. by email or on a memory stick etc. We do not dictate this. (Indeed if we were to dictate normal email, this would be an open invitation for new types of phishing attack.) Once the delegate receives the URL, he clicks on it, whereupon the proxyS3 service calls getIdentity which asks the delegate to authenticate via his preferred IdP. Once authenticated, getIdentity returns the identity of the delegate as an IdP/PId pair (assuming the delegate has no other trusted attributes) and the proxyS3 service calls useSecret

<table>
<thead>
<tr>
<th>Account Type</th>
<th>Account Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kent Personal Account</td>
<td>University=kent.ac.uk, ID=dwc8</td>
<td>Pending creation (check status)</td>
</tr>
<tr>
<td>Role at University</td>
<td>University=kent.ac.uk, Role=staff</td>
<td>Create Account</td>
</tr>
<tr>
<td>Role</td>
<td>Role=staff</td>
<td>Open</td>
</tr>
</tbody>
</table>

**TABLE I. ACCESS ACCOUNTS WELCOME PAGE**
passing it the secret from the URL, the identity of the delegate (IdP/PId) and the latter’s nickname for the delegator. The delegation service assigns (internally) the encoded delegation attribute to the delegate, and stores this information, ready for it to be used again in subsequent accesses by the delegate to the proxyS3 service. The Delegation API returns the encoded delegation attribute to the proxyS3 service, which can now determine which resource(s) the delegate has access to, by calling the Authz API (getRights).

The next time the delegate contacts the proxyS3 service it asks the Authn API (getIdentity) to authenticate the delegate via his chosen IdP. The proxyS3 service calls getDelegationAttributes, passing it the delegate’s IdP/PId and is returned the encoded delegation attribute(s). The proxyS3 service now calls getRights and is returned the resource(s) and access rights granted to the delegate.

VII. CURRENT LIMITATIONS

The current system is only a first proof of concept and has a number of limitations. Firstly none of the security infrastructure components are themselves cloud services, and therefore they will be limited in capacity and throughput. For example, the Authz API has been implemented using a MySQL database rather than a cloud based database.

The Authn API anticipates that each federated identity management system will develop its own class hierarchies for attribute types and issuers (IdPs) in order to simplify the process of requesting trusted attributes, but this has not yet been implemented. The current implementation has simply used a set of LDAP attribute types (with no relationship between them) and a set of trusted IdPs to issue them. Further work on specifying attribute and IDP class hierarchies is needed in order to cater for scalability and inter-federation interworking.

The security APIs have currently only been written in PHP, and therefore are not available in other programming languages. This reduces their utility to a subset of cloud developers, and excludes Open Stack users for example, which use Python.

The Authz API currently does not use a PDP. Thus sophisticated policy based authorization rules are not possible, such as grant access between 9am and 5pm but only if the user is located on campus.

The Owner table is a vulnerable component since it stores all the account KeyID, QueryID pairs. It therefore needs to be strongly protected in any operational system.

The proxyS3 service itself is an attack point, since it has unrestricted access to all users’ accounts. Thus if it is compromised the attacker can potentially gain access to any S3 resource it is protecting. Thus the proxyS3 service needs to be as secure as the native S3 service.

The security APIs themselves are vulnerable to attack. Modification of the Authz API’s backend database would give an attacker who can assert any trusted identity attribute access to any S3 resource by modifying the database. Even an attacker who cannot assert an identity attribute, but can authenticate via a public IdP, could insert a suitably encoded delegation attribute to gain access to a resource. Consequently the authz database needs to strongly protected.

Tampering with any of the PHP code could allow invalid attribute assertions to be treated as valid or unauthenticated users to gain access.

In order to gain wide acceptance, the security APIs need to be standardized by a body such as the OGF, so that all cloud application developers can use them. In this way, many different language variants would be created.

ACKNOWLEDGMENTS

This research has received funding from the EC FP7 under grand agreement no. 216287 (Trusted Architecture for Securely Shared Services) and the UK EPSRC under grant ref. no. EP/1034181/1 (My Private Cloud)

REFERENCES
