A Credential Store for Multi-tenant Science Gateways

Thejaka Amila Kanewala
Research Technologies, UITS
Indiana University, USA
thejkane@iu.edu

Suresh Marru
Research Technologies, UITS
Indiana University
smarru@iu.edu

Jim Basney
NCSA
University of Illinois, USA
jbasney@illinois.edu

Marlon Pierce
Research Technologies, UITS
Indiana University, USA
marpierc@iu.edu

***Abstract*—Science Gateways bridge multiple computational grids and clouds, acting as overlay cyberinfrastructure. Gateways have three logical tiers: a user interfacing tier, a resource tier and a bridging middleware tier. Different groups may operate these tiers. This introduces three security challenges. First, the gateway middleware must manage multiple types of credentials associated with different resource providers. Second, the separation of the user interface and middleware layers means that security credentials must be securely delegated from the user interface to the middleware. Third, the same middleware may serve multiple gateways, so the middleware must correctly isolate user credentials associated with different gateways. We examine each of these three scenarios, concentrating on the requirements and implementation of the middleware layer. We propose and investigate the use of a Credential Store to solve the three security challenges.**

***Index Terms*—Science Gateways, Security, OA4MP, Apache Airavata, Credential Store**

I. Introduction

Science Gateways [1] hide complexities in accessing and using cyberinfrastructure resources while providing application-centric or science-specific user interfaces to scientists. Gateways typically support communities of scientists, enabling them to run computational experiments on remote computing resources and to manage and share data and metadata in a secured and controlled fashion. Gateways are typically developed as a 3-tiered architecture with a portal layer handling the user interface as one tier, a middle tier that is responsible for the data and execution management, and a resource tier that provides the computing and storage resources. These logical tiers may be further subdivided or merged in actual implementations. As illustrated in Figure 1, we refer to the middle tier as the Gateway Middleware Tier. The Gateway Middleware Tier is increasingly hosted as a separate set of services that is distinct from the user interfaces and resource layers. Further, each of these tiers may be operated by a distinct group. Ultrascan [2], Paramchem [3], DARE [4], and iPlant [5] are examples that follow this trend. The user interface layer interacts with the middleware layer through an Application Programmer Interface (API). The API cleanly abstracts the presentation layer from the heterogeneity in computational resource interactions. These include resource-specific authentication, authorization mechanisms, data encodings, communication protocols and execution management. Users with security credentials must be able to authorize the middleware to act on their behalf. Science Gateways

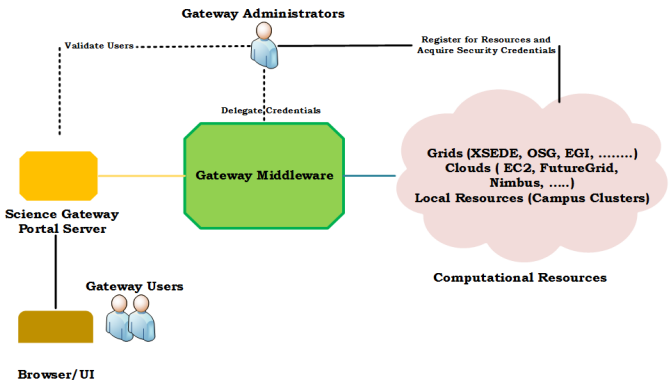

Fig. 1: The overall structure of a Science Gateway consists of three logical tiers. Separation of the gateway middleware from the portal server tiers and the proliferation of different types of resource tiers introduce security challenges.

serve communities of users who do not necessarily have accounts on computational resources but are brokered through community accounts allocations [6]. Gateway administrators own and manage these community accounts. This scenario is quite common on national grid computing infrastructures like XSEDE [7]. This is also a good model for university coursework, where the instructor acquires the allocation on the grid and allows students who are taking the course to use the allocation through a gateway. The gateway interface enables the students to concentrate on the science problems instead of details associated with using complicated computing resources.

In this paper we discuss the challenges faced by the science gateway infrastructure and the need for a secured Credential Store to manage diverse resource credentials, to map end user identities with community accounts and to enable credential delegation without human interaction while creating a trusted multi-hop authorization. Section II discusses the related work and complementary security components within the cyberinfrastructure ecosystem. Section III describes the science gateway security requirements in greater detail. Section IV introduces an abstract view and design of the Credential Store, and Sections V and VI present a software

implementation of such a component and discuss its role in science gateway infrastructure. Section VII concludes the paper with a discussion of future work.

II. Related Work

MyProxy [8], an X.509 credential management system, is primarily used in grid computing. For example, XSEDE [7] uses MyProxy as a primary credentialing mechanism for users with an approved computational allocation. The MyProxy server generates short-lived X.509 credentials based on the users request, authenticated by a username and password. These credentials are authorized for use on XSEDE resources by the XSEDE accounting system. Since MyProxy manages only X.509 credentials, an alternate approach is required to support heterogeneous resources and associated authentication mechanisms like SSH (Secure Shell) keys. Also, MyProxy does not assist gateways with mapping end users who lack grid credentials to shared community accounts.

For a simplified gateway scenario where a portal is directly interacting with resources, the OAuth for MyProxy (OA4MP) service [9] provides a two-hop solution built over the OAuth Version 1 [10] protocol. To fully enable a complex three-tiered gateway, this solution has to be extended to three hops where an administrator/end user can delegate credentials to an intermediary middleware service that acts on its behalf for an extended duration, facilitating long running computational experiments. A comprehensive list of science gateway security requirements and usage scenarios is discussed in [6] and [11].

The Java Keystore [12] is a well-known mechanism for storing keys. There are also a number of repositories (such as LDAP) capable of storing keys. But science gateways must deal with heterogeneous credential types. For example, a gateway will use X.509 credentials to communicate with grid resources and SSH credentials to communicate with Amazon EC2 services [13]. We conclude that what is needed is a Credential Store to persistently store any type of credential so those resources can be easily integrated with the gateway middleware.

III. Problem Statement

We can summarize the challenges for science gateway computational resource credential management as a) credential delegation for gateway systems that separate the user interface components and gateway middleware services into separate network services; b) management of multiple credentials associated with a heterogeneous collection of grids, clouds, and local resources; and c) gateway middleware services potentially serving multiple gateway user interfaces, requiring the gateway middleware to support multi-tenant credential management.

First we consider the delegation problem, for example, when gateway user interface components are separated from gateway middleware services (which manage job executions, file management and similar headless tasks). These interfaces and services may be operated by separate groups a desirable separation for many reasons, as gateways can outsource their generic tasks to centralized services and concentrate on user-facing capabilities. But this brings up a design challenge; since the credentials are associated with the end user, we need a mechanism to securely delegate credentials to middleware layer.

The traditional MyProxy [8] approach is for the user to enter the MyProxy username and password via the UI so the middleware can get short-lived certificates from MyProxy. In this case, no hard-coded authentication data is stored on the file system. The user provides the authentication data (username/password) via the UI when launching jobs. But for many grid resources, particularly XSEDE, community credentials [6] are widely used to share cost (that is, charge to the same account) across multiple end users who do not have personal allocations on the Grid. Community credentials are usually managed by Gateway Administrators or other privileged users. We need a secure way of delegating resource credentials to end users. Since community credentials are needed to make secure invocations on the resource layer, those credentials need to be managed at the gateway middleware layer, not at the gateway user-interface layer. Therefore we need a mechanism to directly provision credentials to gateway middleware without embedding credential information in each request sent from the portal.

Straightforward solution this problem is to provide resource authentication data (that is, the MyProxy username and password) to the operators of the gateway middleware tier (Figure 2). But hard coding authentication data will forfeit the flexibility of the gateway middleware. Also, managing this authentication information in a file system will require sufficient security. If we use community credentials, we will lose the audit trails about the originating user, as many portal end users will be mapped to single community credentials (see Figure 3).

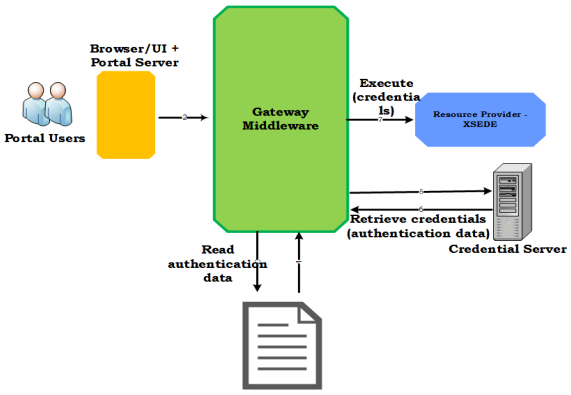

Fig. 2: Straightforward solution to credential transferring problem

Gateway middleware will use community credentials to authenticate to resources. Once the execution path passes gateway middleware, information about who invoked the operation will be lost unless the gateway adds per-user information to the community credential [6]. This means a loss of accountability at the resource level. If a resource is attacked through the gateway middleware we won't have a way to backtrack to the

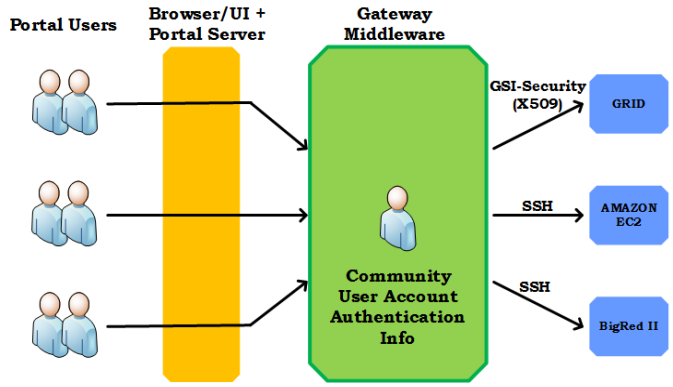

Fig. 3: Mapping multiple users to the same community account user

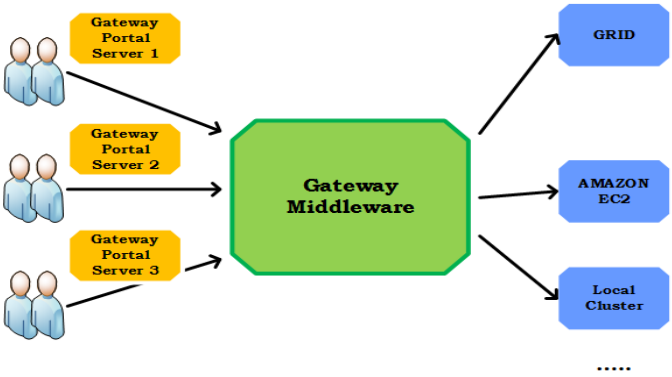

Fig. 4: Multiple gateway portal servers connecting to the same gateway middleware

responsible attacker without consulting the gateway middleware.

Managing authentication data for heterogeneous resources at the gateway middleware layer is a challenge since there are many different authentication mechanisms (see Figure 3). For example, Indiana University's Big Red II supercomputer uses SSH keys as an authentication mechanism, whereas XSEDE Trestles uses X.509 credentials that are part of the XSEDE single sign-on infrastructure. Cloud services such as Amazon EC2 [13] use Public Key Infrastructure as the primary authentication mechanism and SSH as an access mechanism. The Globus Toolkit [14], UNICORE [15], HTCondor [16] and gLite [17] are also middleware systems that can be used to run operational cyberinfrastructure. The gateway middleware should have a mechanism to efficiently manage credentials from these heterogeneous sets of resources.

Gateway providers may rely on third-party services to provide general-purpose applications and data management services in the gateway middleware tier. This relieves providers from operating these services themselves. From the gateway middleware providers point of view, it is preferable to concentrate gateways into a single, multi-tenanted service to obtain better operational scalability. In this multi-tenant scenario, we need to manage credentials for each portal server separately. Each gateway will be concerned about their credential usage. The gateway middleware needs to guarantee that credentials owned by a particular gateway are used only for jobs submitted through that gateway. The audit logs need to be managed separately for each gateway. We need to make sure each portal server is handled individually when it comes to credential management at the gateway middleware layer. This behavior is depicted in Figure 4.

IV. Design

To solve problems described in Section III, we propose to create a Credential Store, a secure database for storing authentication data with added utilities for performing delegation and key generation. The Credential Store is developed as a pluggable module for gateway middleware and provides a component interface for such operations as persistently storing and renewing credentials. Any gateway middleware can use the Credential Store as a library.

The gateway middleware can decide whether to use a separate or shared database (with Middleware) for the Credential Store. In deployment, we need to make sure database files are properly secured using file access control mechanisms provided by the operating system, and database access is secured using proper authentication and authorization mechanisms.

We also require that the gateway portal and gateway middleware trust each other. That is, the gateway portal authenticates end users and passes user information to the gateway middleware. We must trust the gateway portal not to give a malicious or incorrect user id and the Credential Store to generate correct audit data (for more on this, see Section IV-C). Trust between the gateway portal and middleware can be achieved by a deployment model such as the one shown in Figure 5. This is based on TLS (Transport Layer Security) [18] mutual authentication. The Credential Store operation can be

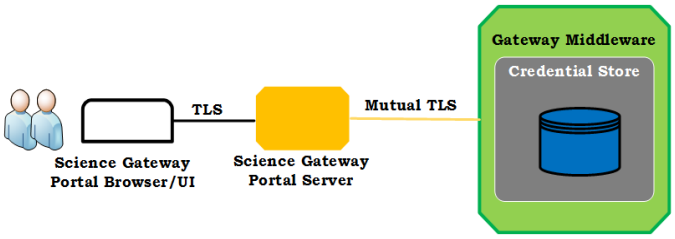

Fig. 5: A deployment model that trusts gateway portal server and gateway middleware

divided into three steps.

1) Initialize a Credential Store for each gateway portal layer when it registers with the gateway middleware layer.
2) Place credentials in the Credential Store. This can be done using a delegation mechanism such as OAuth [10], generating a key pair (explained below), or manually calling a service interface method in the gateway middleware.
3) Query appropriate credentials from the Credential Store during a job submission request.

The following sections discuss each step in detail.

A. Store Initialization during Gateway Registration

The gateway middleware hides complexities in resource communication. The gateway portal needs to go through a registration process with the gateway middleware. During registration we establish trust between the gateway middleware and the gateway portal. The main step in registration is setting up mutual TLS authentication through a public key exchange. Vetting between the gateway and the middleware is done through human interaction by administrative users, and humans are responsible for depositing keys. During the registration phase, the gateway middleware needs to call the Credential Store with certain parameters. In general we expect the following parameters to be passed to the Credential Store:

1) Gateway Id: A unique id the middleware assigns to the gateway that is used to uniquely distinguish gateways;
2) Gateway portal administrator "portal user id"; and
3) Gateway administrator email: We need gateway portal administrator information for auditing purposes and also for notifications. (Section IV-E discusses notifications.)

Figure 6 shows Credential Store operation during gateway registration.

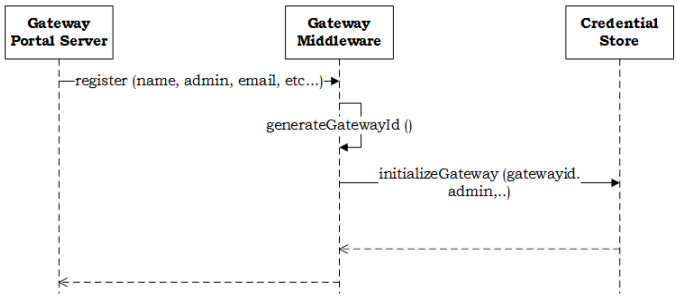

Fig. 6: Store initialization for a newly registered gateway

B. Persisting Credentials in the Credential Store

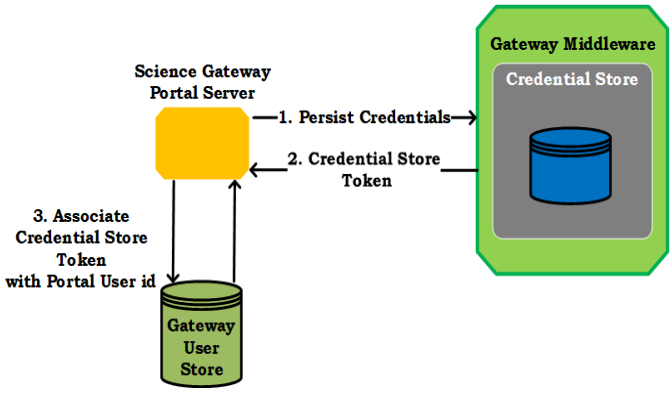

Fig. 7: Credential persistence overview

This section discusses various ways of provisioning credentials in the Credential Store (Figure 7 provides an overview.) Each time the store receives new credentials, it generates a token that is returned to the gateway portal. At the portal server this token is associated with users in the user store. Later, when invoking a job, the associated token is sent in the request.

Credentials can be deposited into the Credential Store in three ways:

1) Delegation-based credential persistence;
2) Generating SSH keys (based on credential persistence); and
3) Invoking a gateway middleware service API method (raw credential persistence)

Discussions of each mechanism follow.

Delegation-based credential mechanisms: These can be easily plugged into the Credential Store. A credential delegation mechanism for grids is "OAuth for My Proxy" (OA4MP) [9]. Figure 8 illustrates how OA4MP can be integrated with the Credential Store. In this scenario resource credentials are directly retrieved into the Credential Store on behalf of the credential owner (usually the gateway administrator). Figure 8 depicts how credentials are persisted

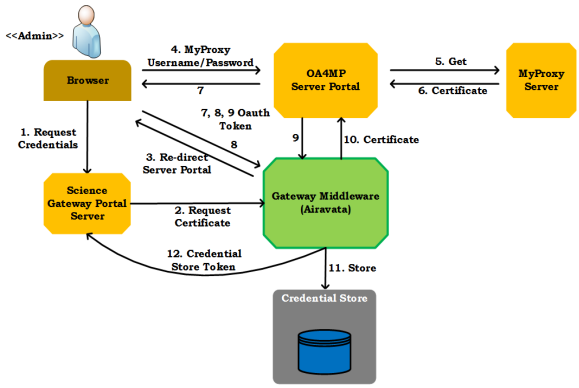

Fig. 8: Persisting community credentials using OA4MP delegation

using OA4MP delegation. It also shows OAuth protocol steps. The gateway administrator, who knows the MyProxy username and password for the community credential, requests credentials from the Credential Store. The OAuth protocol retrieves certificates without passing the MyProxy password to the gateway middleware. The retrieved certificates are stored in the Credential Store in an encrypted form along with their private keys and a token is generated, which is a random value generated to encrypt data in the Credential Store (using AES encryption). The token is then passed to the science gateway portal through the callback URI registered during the gateway initialization.

Figure 8 shows how we persist community credentials. Individual credentials are also persisted in the same manner, but the process is performed by the actual user, not by an administrator. In the case of individual credentials, the token received by the portal server is associated with a single user, whereas in a community account, the token is associated with multiple users.

OA4MP presumes that the resources are associated with a grid security domain: the credential that is returned can be used to access one or more resources by an appropriate

client program. This is the case, for example, with the X.509-based security systems used by XSEDE. Gateways, however, are not tied to a specific grid infrastructure operator and can also provide access to applications running on campus clusters and supercomputers. In such cases, we also need to support SSH-based access through public-private key pairs.

SSH key generation-based credential persistence: Cloud resources such as Amazon EC2 [13] and campus computing resources such as Indiana University's Big Red II support SSH-based authentication. Those resources do not provide delegation mechanisms, and every user must have an account on the resource authentication system (LDAP or database) to communicate with the resource.

One approach is to directly get a user's SSH keys (private and public) and store them in the Credential Store. But end users should not be required to give their keys to a third party. Instead we generate a key pair within the Credential Store and deposit that key pair in the store. The public key is returned back to the portal server along with the token. The gateway portal server is responsible for displaying public keys to the user. The sequence of actions involved in key generation and key persistence is depicted in Figure 9. The user is required to deploy SSH public keys to the actual resource by manually logging into the resource. This is a one-time step.

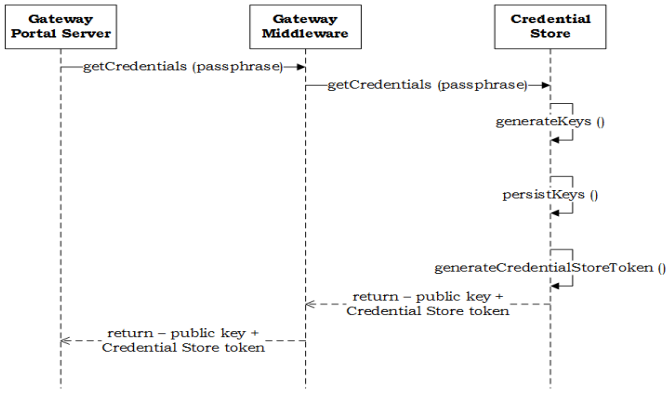

Fig. 9: Sequence of actions taking place when persisting generated SSH keys

Raw credential persistence: If we cannot retrieve credentials using a delegation mechanism or generate credentials (like SSH keys), we can use a direct method to deposit credentials into the Credential Store. Here, the user contacts the resource service/server directly, gets credentials, and deposits them in the Credential Store by calling an API method in the gateway middleware. Once the user calls the gateway middleware with the credentials, the gateway middleware invokes the Credential Store component interface to deposit credentials. As Figure 10 shows, first the user requests and receives credentials for a particular resource, and the Credential Store component interface provides methods to persist and retrieve them. The user can safely transfer credentials over the network to the gateway middleware because we have established mutual trust between the gateway portal server and gateway middleware server.

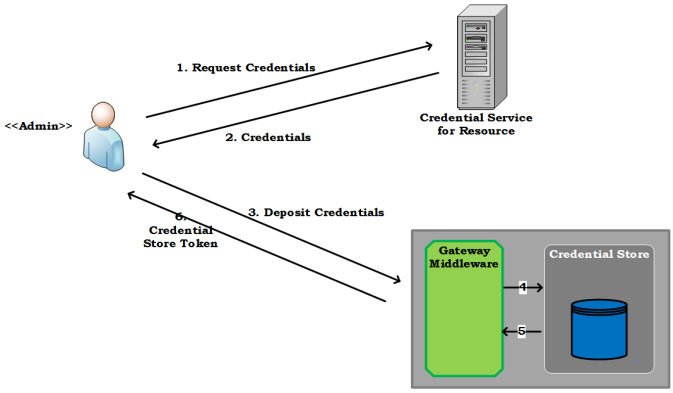

Fig. 10: Raw credential deposit

C. Job Invocation with Credentials

When a credential is persisted into the Credential Store, the Credential Store returns a token to the gateway portal. The portal server associates the retrieved token with portal users, and can decide the strategy for associating tokens with the user store. For example, if credentials are retrieved for a community user in a given project, the portal can associate a token with all the users in the project. If a credential is retrieved for an individual user, the portal server can decide to associate that token with the individual user alone. Once the token is associated with the user, the portal can access remote resources securely through the gateway middleware layer.

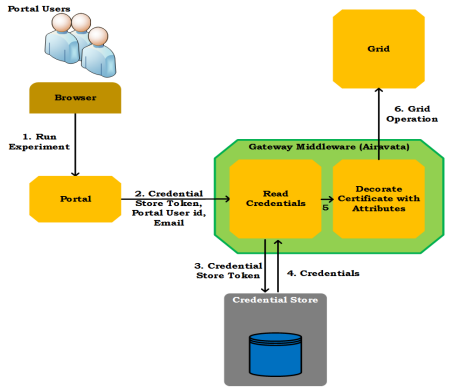

Fig. 11: Component interaction while executing a job using persisted credentials

During job execution, the portal server needs to submit the token associated with the user and metadata needed to execute the job. Middleware will extract the token from the job execution request, confirm that the token is owned by the portal (i.e., matches the portal identity from TLS [18] mutual authentication as per Figure 5), and invoke the Credential Store to get the appropriate credentials. Figure 11 shows how credentials are retrieved from the Credential Store while executing a job. Figure 12 depicts the sequence of actions performed at each entity when retrieving credentials from the Credential Store. If credentials are X.509 certificates, we decorate the retrieved credentials (certificates) with end-user

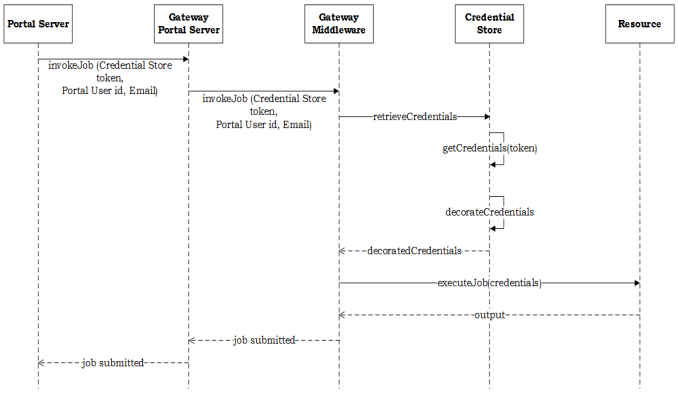

Fig. 12: Operations performed at each entity during credentials retrieval

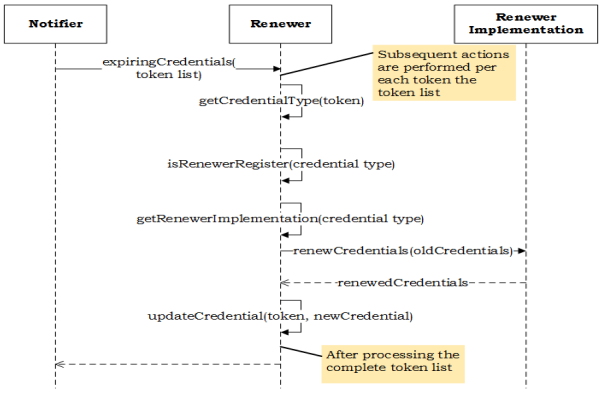

Fig. 13: How credential "renewer" is executed during a job submission

information, and the portal server sends the end users portal username and email. Since we have mutual trust between the portal server and gateway middleware, we rely on the portal server to send correct information about the user who is executing the job. For X.509 credentials (MyProxy credentials) we embed portal user information along with other metadata in the certificate in the form of an SAML [19] attribute token. To do this we use GridShib [20] tools.

For SSH credentials, we write an audit record about the portal user who accessed the credentials (along with other access data, such as time accessed, from which IP address the request came, etc.).

D. Credential Renewal

By default, we honor the policy defined by the credential-issuing entity on the credential's lifespan and expiration time. It is up to the manager of the credential to renew it. In the Credential Store we calculate and record the credential's expiration time when it is deposited. Before expiration, we send notifications to the user who deposited those credentials (the lead time for these notifications is a configurable parameter in our implementation). Currently, the Credential Store can send notifications in the form of email. The notification methodology is configurable and new delivery methods can easily be plugged into the Credential Store. Further, we can configure notifications to be sent after a defined period.

For non-interruptive operations, we may need to renew credentials programmatically, overriding the default behavior described above. We can implement renewal by registering a "renewer" for a credential type.

When a credential renewer is registered for a particular type of resource, the execution order is depicted in Figure 13. Credentials must be renewed before they expire. The Credential Store consists of a component called a Notifier (explained in Section IV-E), which notifies when a credential nears its expiration. The renewer registers itself with the Notifier. When the Notifier indicates that a particular credential is about to expire, the renewer is notified. The renewer checks whether there is a renewer implementation registered for the type of credential. If so, the appropriate renewer implementation is invoked to renew credentials. If the renewer is able to successfully renew credentials, the Credential Store will update stored credentials against its indexed token.

Our implementation includes the MyProxy-based credential renewer, discussed in the following paragraphs. SSH keys do not expire, but stay in the Credential Store until they are deleted. They will be reused when submitting jobs.

MyProxy based credential renewer: The MyProxy-based credential renewer requires following a deployment function pattern. To renew proxy credentials, we follow an approach similar to the way the WMS [21] system renews its credentials. We register the gateway middleware as a trusted renewer. The gateway middleware registers with MyProxy server and can retrieve credentials for some time (a year, for example). This process may need additional permissions. The long-term credentials are deposited into a Credential Store under a special token, accessible only through gateway middleware. The renewer then uses the special token to renew the portal-persisted credentials.

For example, a user is assigned to the middleware during deployment and this user is registered as a trusted renewer in the MyProxy server and gets credentials that are valid for one year. Now, the middleware gets an 11-hour certificate from the portal in a request. To renew the 11-hour certificate, the middleware sends a renewal request to the MyProxy server with the assigned user's credentials. (depicted in Figure 14).

E. Credential Store Object Model

The basic structure of the Credential Store is depicted in Figure 15. We explain each sub-component below.

1) Store - This is the secure database that stores credentials. As the store should be able to store any type of credentials, it is designed to ignore the structure of actual credentials (e.g.: an X.509 credential or an SSH key). Credentials are secured in 3 layers. First: The data store is deployed in a host with restricted access to its file system. Second: The database is secured using an available authentication mechanism (username/password). Third: The credential is encrypted using a strong symmetric key algorithm (AES [22]). The key for the symmetric key

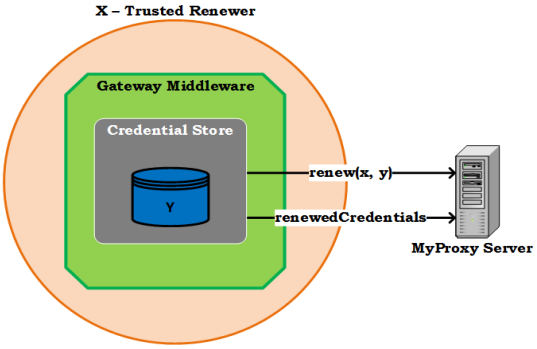

Fig. 14: MyProxy credential (X. 509 Certificate) renewing example, where x is a trusted renewer

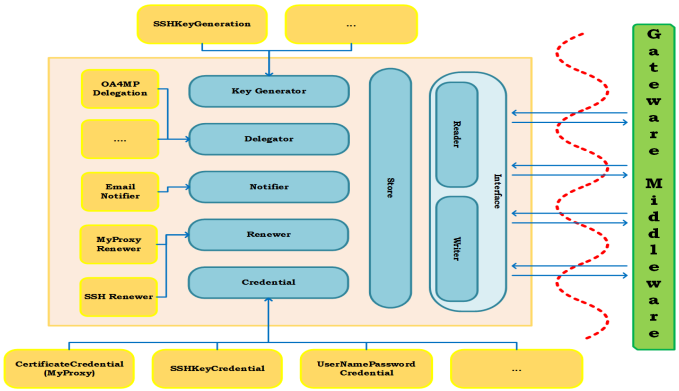

Fig. 15: The component view of the Credential Store and how it interacts with gateway middleware

algorithm is generated based on a passphrase provided by the administrator. The passphrase can reside in a configuration file or be input when the gateway middleware initializes. We recommend supplying a passphrase at the start of the gateway middleware.

2) Credential - This is an abstract representation of the credentials that the Credential Store handles. The core of the Credential Store is aware of this abstract implementation only. All specific implementations can be plugged into the Credential Store with minor configuration changes and a jar containing the implementation. So far, we have Credential implementations of X.509 certificates and SSH Keys. If we want to add a new Credential implementation (such as a SAML authentication token) we add it as a plugin to the Credential Store.
3) Delegator - A simple approach of using the Credential Store is to deposit a username/password pair for access to a particular resource. Since we do not wish to send a username/password (such as a MyProxy username/-password) to a third party, we need a way to persist credentials on behalf of the user. In other words we need a mechanism to delegate credentials to a trusted third party to perform operations. For a particular type of resource, if a delegation mechanism is available, we can interface that with the Credential Store. Currently we have a delegation mechanism only for MyProxy credentials OAuth for MyProxy (OA4MP) [9]. We discussed OA4MP integration with the Credential Store in Section IV-B.
4) Key Generator - Many resources, such as clouds and campus clusters, use SSH keys as a primary authentication mechanism. Transmitting private keys over the network is not a good practice. We need a mechanism at the middleware layer to generate keys and deploy them in the appropriate resource. Key generation is handled by the Key Generator component.
5) Interface - The Store Interface hides details about the Credential Store operations and provides an easy API-like interface to the gateway middleware. This Interface is divided into two sub-components: CredentialReader and CredentialWriter. CredentialReader is responsible for reading credentials for a given token, while CredentialWriter deposits credentials into the Credential Store and provides interface methods to delete and decorate credentials.
6) Notifier - The gateway administrators will want to know a whether particular credential will expire in order to renew credentials and continue gateway operations without interruption. We have configured email notifiers to send events, based on certain rules. If a particular credential is expiring within a configured time period, the Notifier sends email to the gateway administrators. The Notifier can be configured to receive other types of events based on certain actions. For example, we can send notification messages whenever credentials are queried. As with other components, the implementation of notifier is pluggable. Internally the Renewer (explained in item 7) uses the Notifier to get notifications about credentials that are about to expire.
7) Renewer - This component handles renewing credentials. Like other sub- components, concrete renewer implementations can be plugged into the Credential Store. In this case we use the MyProxy credential renewer, which renews X.509 credentials. If the Notifier indicates expiring credentials, the Renewer is invoked. The Credential Store searches the plugged-in concrete renewer implementation needed to initiate the process. MyProxy renewer implementations must satisfy the deployment requirements discussed in section IV-A or the credential renewal will fail. In such cases the renewer will throw an exception to the caller with the proper error message.

V. Implementation

We implemented the Credential Store as a module in Apache Airavata [23]. In our implementation the Credential Store is a relational database. We tested it with both MySQL [24] and Derby [25]. Implementations of credentials are serialized and the serialized stream is encrypted using the AES [22] algorithm. The encrypted stream is stored in the databases as a BLOB [26] type, along with the associated token and metadata. When retrieving credentials by token, the

Credential Store will decrypt the stream and de-serialize it to get the actual object implementation. For example, in the case of OA4MP, we convert X.509 certificate and its private key into a serialized form, encrypt it, and store in the database. We used standard Java security packages. BouncyCastle [27] handles certificates and other security keys; and JCraft [28] and JGlobus [29] are used for SSH and MyProxy operations, respectively. The key for the AES algorithm is generated based on passphrase input by the administrator. The passphrase can reside in a configuration file, or it can be input when the Credential Store initializes. The recommended approach is to provide the passphrase as an input when the middleware starts and pass it to the Credential Store during initialization. For attribute decoration of credentials we used the GridShib tools [20].

The Credential Store database lies in the file system on the server side and should be secured using standard Unix file access control mechanisms. Database connections are also secured using a provided authentication (username / password) mechanism. The damage done by an attacker getting into the system is minimized by enforcing Unix system security and database security. Even a person who has access to the database will not be able to see credentials as they are in encrypted format.

Each sub-component in the Credential Store is implemented as a pluggable module. We use Java class loading [30] mechanisms with configurations to achieve pluggability. For example if we want to add a new credential type we can implement the Credential interface (Java) provided by the Credential Store, package it as a jar, and place it in the class path. In the Credential Store configuration we then specify which implementations should be loaded to the memory.

Operations related to handling authentication information are logged in a separate audit log. The log file should be secured using Unix access control mechanisms and streamed to a dedicated, secure log server for an independent audit log. For each request, we record information on the machine originating the request, the gateway id, the portal user invoking the request, the DNs (Domain Names) and serial numbers of credentials (in the case of MyProxy), SSH public keys, and SSH user names.

In a multi-tenant deployment, the middleware layer enforces isolation across the portal instances, i.e., each portal instance authenticates to the middleware layer, so the middleware layer can ensure credentials are not improperly shared across portal instances. Each gateway needs to go through a registration process and establish trust as described in Section IV-A. During the registration each gateway is assigned a unique ID which is later used to distinguish gateways from one another. For each request processed by the Credential Store we create a context indexed by gateway ID. Each context will carry the state information needed for a single gateway. During persistence and credential retrieval we use the gateway ID to isolate credentials related to a gateway. The audit logs are separately managed for each gateway and the Credential Store component interface takes the gateway id as a parameter for each method of invocation.

VI. Use Cases - Integration with Apache Airavata Science Gateways

The credential store is integrated with the Apache Airavata [23] science gateway framework. Gateways using Airavata are transitioning to take advantage of the capabilities discussed in Section IV. In this section we discuss two complementary use cases in detail: 1) the integration of MyProxy with the ParamChem [3] gateway and 2) the use of the SSH Key-based approach in the Indiana University Cyber Gateway.

A. ParamChem Science Gateway using Apache Airavata

ParamChem users use the desktop tool Paramberoo to perform dihedral angle parameter optimization. A significant group of users may use XSEDE HPC resources to execute workflows that involve computationally intensive reference data generation, while non-XSEDE users can provide their own reference data and run short-lived workflows on the computing resources provided by ParamChem. Paramberoo is integrated with Apache Airavata for workflow management and XSEDE job handling capabilities.

The ParamChem gateway administrator initializes the gateway with Apache Airavata during the bootstrap step. In this process, the XSEDE X.509 short-lived credentials issued by the MyProxy server are stored. In the traditional approach the ParamChem gateway is used to store the MyProxy username and password in a properties file trusted by the file system security. With the integration of the Credential Store, the ParamChem gateway administrator never needs to store authentication data in configuration files.

The gateway administrator logs into the XSEDE OAuth [10] portal and initiates the transaction. The transaction is completed when the handle is passed to the Credential Store, which interacts with the OAuth server, retrieves the certificate, and deposits it. The Credential Store-generated token is then passed to the Science Gateway portal through a callback URI. These steps are illustrated in Figure 8 and discussed in detail in Section IV-B.

Once credentials are deposited, end users can execute parametrization workflows using the molecular editor client. Users use the client to navigate through the wizard interface to select a dihedral parameter, configure parameterization options, and launch the parametrization workflow. During this, Paramberoo passes the Credential Store token retrieved during credential persistence. This token is propagated to the Airavata job management framework, which consults the Credential Store to retrieve credentials. Those credentials are then used to execute jobs and move data. Users retrieve their results, and data is archived on mass storage devices for persistence. These steps are the same among all gateways that use XSEDE resources through Apache Airavata.

B. IU Cybergateway SSH Credentials

The Indiana University (IU) Cyberinfrastructure Gateway provides a consolidated, web-based gateway to the IU research

computing infrastructure: Big Red II, Quarry, and Mason [31]. Users can get summary views of resource loads, information about their submitted and running jobs, searchable lists of available software on these research computing resources, and interactive plots of the usage statistics. Users can also transfer data from their laptops and desktop machines to storage systems, and between IU and other computing resources such as XSEDE. Here, we focus on the gateway's ability to launch applications directly through the gateway web interface and to build custom, science-centric web gateways from the gateway services. This use case illustrates the use of auto-generated SSH keys, as illustrated in Figure 9 and described in Section IV-B.

During the user registration step, the cybergateway invokes the Credential Store to persist credentials through the gateway middleware. Once SSH keys are deposited into the Credential Store, the cybergateway gets a token and a public key. The user is then required to add the generated public key to the authorized_keys file on the target compute resource. For security purposes, the resource authentication is locked down, allowing only the gateway middleware server to access the resource (i.e. the host access list in authorized_keys file inside resource has only gateway middleware server address and no outside client machine is allowed to access the resource). After this, users can submit jobs to the gateway. The gateway, in turn, makes a request to the Apache Airavata middleware and Airavata invokes the Credential Store with the token. Airavata will retrieve the SSH private key from the Credential Store and use it to communicate with the resource.

VII. Summary & Future Work

The owners of resource credentials are authorized, special end users. In the case of community credentials, these special users are the gateway administrators. There are also stand-alone users who own resource credentials. Between the end user and resource, the gateway middleware hides resource complexities; the credentials are used by the gateway middleware. Transferring end-user resource credentials to gateway middleware is a delegation problem in standard security terms. But gateway middleware connects to heterogeneous resources, and some resources do not provide a usable delegation mechanism. Delegation is not always the best solution. If a delegation mechanism is available, the Credential Store uses it to transfer credentials. Where there is no such mechanism, the Credential Store uses other approaches such as SSH key generation and manual persistence of credentials.

In this section we discuss a few exceptional scenarios that are possible during this operation. One possibility is that credentials are compromised, and we must rely on audit records to sort out which credentials are compromised. In this case we go through audit records to figure out responsible credentials (DNs, serial numbers, public keys) and delete them from the Credential Store. The next time, the legitimate user needs to persist new credentials before executing jobs.

As of now we do not have a mechanism to determine audit log integrity, but are researching an efficient way to preserve it as described in [32]. There could be cases where credentials are compromised but audit logs do not show which ones. We would need to remove all credentials in the Credential Store to make sure the system is in a safe state. Every user would then need to deposit credentials back to the Credential Store and perhaps manually store them in the authorized keys file (for SSH keys). Though this is cumbersome process, it is the safest action given the situation. We do not currently have a way to automate credential restoration, and this is an area we would like to further investigate.

As a preventive mechanism we recommend deploying the Credential Store database in a separate server with strong restrictions (i.e. only the database storage is kept in a separate host). We can implement strong file system security, make the database accessible only to the gateway middleware, modify firewalls (IPtables) so that only the gateway middleware can communicate with the database, and take other standard security measures.

We expect the gateway middleware tier to implement standard fault-tolerant mechanisms to avoid single points of failure. Further, the Credential Store database should also be replicated on two servers to guard against the service being unavailable, and should be secured using the same security mechanisms discussed in Section IV.

An implementation of the Credential Store was tested using Apache Airavata [23] middleware. In this implementation the Credential Store was coupled to Apache Airavata middleware. We plan to separate out the Credential Store and make it a separate software component. Currently GFac (the resource communicating module within Airavata) [23] uses the Credential Store to retrieve credentials for both grid and SSH communication. Not all resource providers support OAuth-like interfaces to delegate credentials. We are planning to improve the Credential Store to work with better delegating protocols such as OAuth Version 2 [33]. Further, for some credential types, we do not have proper delegation mechanisms such as OA4MP. For example, we do not have such a mechanism for SSH. We plan to investigate how we can incorporate OAuth-like protocols with SSH credentials.

Acknowledgment

This work is supported by the National Science Foundation under grant number ACI-1127210.

References

[1] N. Wilkins-Diehr, D. Gannon, G. Klimeck, S. Oster, and S. Pamidighantam, "Teragrid science gateways and their impact on science," *Computer*, vol. 41, no. 11, pp. 32–41, 2008.

[2] B. Demeler, "Ultrascan: a comprehensive data analysis software package for analytical ultracentrifugation experiments," *Modern analytical ultracentrifugation: techniques and methods*, pp. 210–229, 2005.

[3] J. Ghosh, N. Singh, Y. Fan, S. Marru, K. Vanomesslaeghe, and S. Pamidighantam, "Molecular parameter optimization gateway (paramchem)," in *Proceedings of the 2011 TeraGrid Conference. ACM*, 2011.

[4] S. Maddineni, J. Kim, Y. El-Khamra, and S. Jha, "Distributed application runtime environment (dare): A standards-based middleware framework for science-gateways," *Journal of Grid Computing*, vol. 10, no. 4, pp. 647–664, 2012.

[5] E. Skidmore, S.-j. Kim, S. Kuchimanchi, S. Singaram, N. Merchant, and D. Stanzione, "iplant atmosphere: a gateway to cloud infrastructure for the plant sciences," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM, 2011, pp. 59–64.

[6] V. Welch, J. Barlow, J. Basney, D. Marcusiu, and N. Wilkins-Diehr, "A aaaa model to support science gateways with community accounts," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 893–904, 2007.

[7] J. Towns, "Evolving from teragrid to xsede," *Bulletin of the American Physical Society*, 2011.

[8] J. Basney, M. Humphrey, and V. Welch, "The myproxy online credential repository," *Software: Practice and Experience*, 2005.

[9] J. Basney and J. Gaynor, "An oauth service for issuing certificates to science gateways for teragrid users," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011, p. 32.

[10] B. Leiba, "Oauth web authorization protocol," *Internet Computing, IEEE*, vol. 16, no. 1, pp. 74–77, 2012.

[11] J. Basney, V. Welch, and N. Wilkins-Diehr, "Teragrid science gateway aaaa model: implementation and lessons learned," in *Proceedings of the 2010 TeraGrid Conference*. ACM, 2010, p. 2.

[12] Wikipedia. (2013) Keystore. [Online]. Available: http://en.wikipedia.org/wiki/Keystore

[13] E. Amazon, "Amazon elastic compute cloud (amazon ec2)," *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[14] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications*, 1997.

[15] D. W. Erwin and D. F. Snelling, "Unicore: A grid computing environment," in *Euro-Par 2001 Parallel Processing*. Springer, 2001.

[16] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *Distributed Computing Systems, 1988., 8th International Conference on*. IEEE, 1988.

[17] P. Andreetto, S. Andreozzi, G. Avellino, S. Beco, A. Cavallini, M. Cecchi, V. Ciaschini, A. Dorise, F. Giacomini, A. Gianelle *et al.*, "The glite workload management system," in *Journal of Physics: Conference Series*, vol. 119, no. 6. IOP Publishing, 2008, p. 062007.

[18] T. Dierks, "The transport layer security (tls) protocol version 1.2," 2008.

[19] J. Rosenberg and D. Remy, *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*. Pearson Higher Education, 2004.

[20] T. Barton, J. Basney, T. Freeman, T. Scavo, F. Siebenlist, V. Welch, R. Ananthakrishnan, B. Baker, M. Goode, and K. Keahey, "Identity federation and attribute-based authorization through the globus toolkit, shibboleth, gridshib, and myproxy," in *5th Annual PKI R&D Workshop*, 2006.

[21] D. Koufil and J. Basney, "A credential renewal service for long-running jobs," in *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*. IEEE, 2005, pp. 6–pp.

[22] E. Whaley and D. Miller, "Aes advanced encryption standard."

[23] S. Marru *et al.*, "Apache airavata: a framework for distributed applications and computational workflows," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM, 2011.

[24] A. MySQL, "Mysql," 2001.

[25] Apache. (2012) Apache derby. [Online]. Available: http://db.apache.org/derby/

[26] Wikipedia. (2013) Binary large object. [Online]. Available: http://en.wikipedia.org/wiki/Binary_large_object

[27] B. Castle, "Bouncy castle crypto apis," *U RL http://www. bouncycastle. org/.(Cited on page 82.)*, 2007.

[28] D. A. Yamanaka. (2012) Jcraft. [Online]. Available: http://www.jcraft.com/

[29] JGlobus. (2013) Jglobus github community. [Online]. Available: https://github.com/jglobus/JGlobus

[30] S. Liang and G. Bracha, "Dynamic class loading in the java virtual machine," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 36–44, 1998.

[31] I. University. (2013) Hpc systems. [Online]. Available: http://rt.uits.iu.edu/systems/hps/

[32] S. Haber, Y. Hatano, Y. Honda, W. Horne, K. Miyazaki, T. Sander, S. Tezoku, and D. Yao, "Efficient signature schemes supporting redaction, pseudonymization, and data deidentification," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008, pp. 353–362.

[33] D. Hammer-Lahav and D. Hardt, "The oauth2. 0 authorization protocol. 2011," IETF Internet Draft, Tech. Rep.