Authentication and Authorization Considerations for a Multi-tenant Service

Randy Heiland
Center for Applied Cybersecurity Research
Indiana University
Bloomington, Indiana USA
heiland@iu.edu

Scott Koranda
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin USA
skoranda@gmail.com

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

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

Von Welch
Center for Applied Cybersecurity Research
Indiana University
Bloomington, Indiana USA
vwelch@iu.edu

ABSTRACT

Distributed cyberinfrastructure requires users (and machines) to perform some sort of authentication and authorization (together simply known as *auth*). In the early days of computing, authentication was performed with just a username and password combination, and this is still prevalent today. But during the past several years, we have seen an evolution of approaches and protocols for auth: Kerberos, SSH keys, X.509, OpenID, API keys, OAuth, and more. Not surprisingly, there are trade-offs, both technical and social, for each approach.

The NSF Science Gateway communities have had to deal with a variety of auth issues. However, most of the early gateways were rather restrictive in their model of access and development. The practice of using community credentials (certificates), a well-intentioned idea to alleviate restrictive access, still posed a barrier to researchers and challenges for security and auditing. And while the web portal-based gateway clients offered users easy access from a browser, both the interface and the back-end functionality were constrained in the flexibility and extensibility they could provide. Designing a well-defined application programming interface (API) to fine-grained, generic gateway services (on secure, hosted cyberinfrastructure), together with an auth approach that has a lower barrier to entry, will hopefully present a more welcoming environment for both users and developers.

This paper provides a review and some thoughts on these topics, with a focus on the role of auth between a Science Gateway and a service provider.

SCREAM '15, June 16 2015, Portland, OR, USA

ACM 978-1-4503-3566-9/15/06...$15.00
DOI:http://dx.doi.org/10.1145/2753524.2753534.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and protection

General Terms

Security

Keywords

Authentication, X.509, API keys, OAuth, usability

1. INTRODUCTION

Our interconnected world depends on distributed cyberinfrastructure (DCI). Whether it is email between family members, searching the web for information, doing online banking, or performing computation and analysis for science and engineering, we rely on DCI. An implicit assumption (or hope) by users is that the DCI we are using is trustworthy. The concept of trustworthiness deserves considerable attention and we will only be able to touch on it here. How do we know that an email we received from a friend or colleague was actually sent by them? Is it possible their email account was compromised and someone else was pretending to be them? Or perhaps the true sender did indeed originate an email, but the message was intercepted and altered before it arrived. Trustworthiness for online banking is, arguably, a more serious matter. And what about science and engineering? Should the lay public, policymakers, and funding agencies trust scientists' claims of some recent discovery? Claims should always be questioned and verified. Those of us helping to build and support DCI, especially those in information security, can help insure that the underlying technology provides at least some degree of trustworthiness. Auth (authentication and authorization) plays a critical role.

The NSF has funded the evolution of high-end DCI for many years[1]. This led to the formation of the TeraGrid program and, more recently, the XSEDE program. The

[1] http://www.nsf.gov/news/special_reports/cyber/fromsctotg.jsp

Science Gateways program was part of this evolution. Its primary purpose was to make it easier for researchers and educators to use DCI by providing web portals that offered easier access to resources (data, computational, experimental) and tools associated with those resources, e.g., workflow creation, job monitoring, and data analysis/visualization. Security considerations were part of the foundation of Grid technologies[5]. Security considerations for gateways, which act on a user's behalf, have been discussed by Welch et al.[15].

Of course the evolution of DCI has not stopped. During the last few years, we have seen a tremendous growth in social networking and the accompanying infrastructure associated with it, including mobile devices, apps, and numerous online services. Associated with these services are application programming interfaces (APIs) that expose the available functionality. Users can invoke the APIs, typically using one of several software development kits (SDKs) that the developers of the service make freely available. APIs have become commonplace both for the business world[12] and also for science and engineering.

Here, we focus on one particular service, the Science Gateway Platform as a service (SciGaP), and explore the evolving security and usability issues associated with online services. This is ongoing work from a collaborative engagement between the CTSC project (`trustedci.org`) and the SciGaP project. Anderson[1] provides a theme for this project with the following quote: "Managing the evolution of an API is one of the toughest jobs in security".

The target audience for this paper includes not only science gateway administrators, but also DCI administrators and software developers who want a better understanding of auth options that are relevant for Science Gateways and service providers.

2. SCIGAP: A MULTI-TENANT SERVICE

The Science Gateway Platform as a service (`SciGaP.org`) project is an NSF funded effort to provide consolidated, scaleable, elastic services for common gateway operations. Examples of common operations include identity management, application and resource metadata management, experiment provenance for online application executions, and data management (Figure 1). The SciGaP project is based on a multi-tenanted version of the Apache Airavata software system[11]. Airavata's credential store component[9] manages user identities within the gateway.

Establishing authentication and authorization between Airavata and client gateways and desktop client applications is an open problem that this paper examines.

SciGaP is developing an Application Programming Interface (API) in order to meet its goal of scalable support for multiple gateways[13]. The API defines a uniform contract for interactions between SciGaP and its client gateways, eliminating custom, per-gateway integrations that are difficult to sustainably support. SciGaP uses Apache Thrift to define its API and associated data models and releases Software Development Kits (SDKs) in multiple programming languages. Gateway clients integrate these SDKs into their code bases to make over the wire calls to SciGaP services.

SciGaP must address the security challenges introduced by the API approach. SciGaP use cases broadly fall into two categories: 1) SciGaP has direct or indirect (reference) access to the gateway user store, and 2) SciGaP cannot access gateway end user information. In the second case, SciGaP and its client gateways must establish mutual trust (which can be a one-time operation) as well as over the wire security for each session. During a particular session, a client gateway and SciGaP must mutually authenticate to each other. Both sides must also ensure that the integrity of requests and responses is not compromised; privacy via encryption of the message exchanges may or may not be required, depending on the gateway.

These are well-established security considerations. Note also that the gateway establishes trust with its users and can make assertions to SciGaP about a user's roles and permissions. It is the gateway therefore, rather than the user, that is the center of the trust relationships.

Science gateways come in at least two major varieties, which must be considered when developing a security model for SciGaP. The first is the classic Web-based gateway, which has been implicit in the above discussion. Here, users interact via a Web browser with the gateway, which in turn invokes SciGaP services. The gateway is a centralized service that is only accessible to a well-defined operations team. In contrast, a gateway may distribute desktop or mobile application clients to its users. These clients may directly contact SciGaP services. Consequently, gateway sessions do not come from a single, centralized service, making the establishment and maintenance of trust more difficult.

3. AUTH: BACKGROUND

In this section, we offer a basic review of the ideas and technologies behind authentication and authorization.

One of the earliest and still one of the most common ways to authenticate oneself to a computer, web site, etc., is to provide a username and password. The subject of passwords has received and continues to receive considerable attention, in both the daily news and in research. In one sense, passwords, or at least password complexity, can be traced back to the 1940s and the beginning of information theory with the seminal work of Claude Shannon[14]. The concept of entropy, which in a somewhat convoluted fashion was borrowed from statistical mechanics[2], has been used to quantify "good" passwords. In spite of the decades that have since passed, passwords still remain vulnerable to being guessed[3] or obtained through phishing attacks or other means.

One alternative to username/password authentication is public key infrastructure (PKI). PKI arose from the concept of a cryptographic key which is simply a string of bits used to encrypt plain text. Cryptographic key algorithms can be *symmetric* or *asymmetric*: symmetric having a single (private) key and asymmetric having two keys (one public, one private). PKI uses the latter. X.509 is a standard model and protocol (`https://tools.ietf.org/html/rfc5280`) for PKI. It is fairly complex and includes multiple components, including the critical concept of a certificate. An X.509 certificate includes multiple entries, including the time range for the certificate's validity, the subject's public key and the type of algorithm used to generate it, a signature, and the issuer, a.k.a. the certificate authority (CA).

The SSL protocol and, more recently, the TLS protocol (`https://tools.ietf.org/html/rfc5246/`), on which web browsers rely for security, require X.509. A user can in-

[2]Gleick's book, *The Information*[6], offers a lucid discussion of entropy in this context for the lay reader.

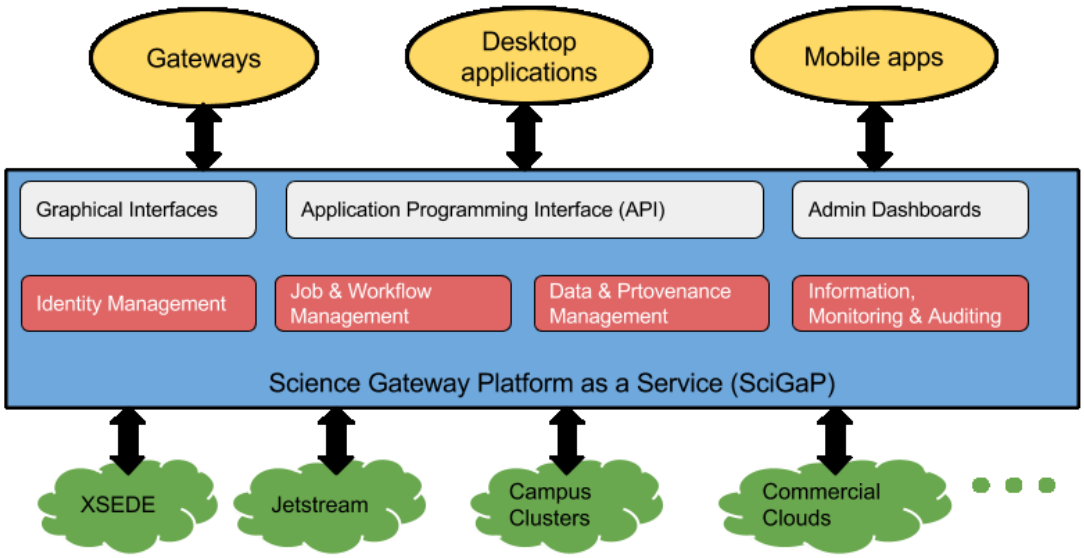

Figure 1: SciGaP as a service architecture overview

spect their web browser's security settings and see that it has many built-in CAs. This infrastructure has been built up over the past few decades and is how we have at least some degree of security on the Internet. However, X.509 has its share of criticism[7][8], primarily due to its complexity.

In the context of Science Gateways during the NSF TeraGrid program, an auth architecture was implemented, based on the X.509 model, that allowed for community accounts[2]. This worked, in part, due to the trust relationship between a science gateway web portal and the resource providers, as described in the previous section.

Another type of auth, with less complexity than X.509, is to use an API key, a.k.a. bearer token or developer token[3]. An API key is a cryptographic key, i.e., a unique, random string of alphanumeric characters, that serves to authenticate and authorize the holder of the key to invoke an API. One could imagine using the widely known Universally Unique Identifier (UUID) as an API key. For example, running the BSD command *uuidgen* will generate a unique 128-bit (hyphenated) ID, such as 92E5E0DB-7A49-4E42-A001-C5B343DB3F06. By removing the hyphens, one might consider using it as an API key. Note, however, that the IETF RFC cautions against doing so (`https://tools.ietf.org/html/rfc4122#section-6`).

In the era of social networking and apps running on mobile devices, many online services have adopted a relatively new auth protocol called OAuth[10] (OAuth 2; `https://tools.ietf.org/html/rfc6749`). OAuth makes it possible for a user to authorize one service to access their information on another service. A simple social networking example would be to allow Twitter to post your tweets on your Facebook page. In spite of this trivial-sounding example, the authorization *flow* that takes place for OAuth is far from trivial, as depicted in section 1.2 of the IETF RFC. One key idea of OAuth is that the user's credentials are not shared between the services; rather, an *access token* is generated and shared. This access token is a string containing a scope of the authorization, its lifetime, and other attributes.

Of course OAuth is used in more than just social networking applications and we will show some of these in the next section. The Agave API[4] is another service for science gateways that has also adopted OAuth[4].

4. RELATED WORK

In this section, we examine a few popular services and see how they deal with authentication and authorization. While these examples may seem repetitive, we wish to demonstrate the variety of options for auth, how credentials are managed, some best practices, and highlight a common approach: OAuth.

4.1 Google

Google provides numerous APIs to its services (`https://developers.google.com/apis-explorer`) and offers the following advice related to auth:

> Every request that your application sends to the Google+ API needs to identify your application to Google. You can use the API key you get when defining your project, or you can use an OAuth 2.0 access token. You should use an access token when you are making calls on behalf of a given user.

The general approach that Google uses to assist application developers is to provide a web-based (HTTPS) console (`https://console.developers.google.com`) to create a project and obtain either an OAuth token or an API key, depending on the nature of the application (Figure 2). Many other online services/APIs provide similar web interfaces.

[3]Note that the terminology surrounding tokens can be a bit confusing and overlapping.

[4]http://agaveapi.co/authentication-token-management

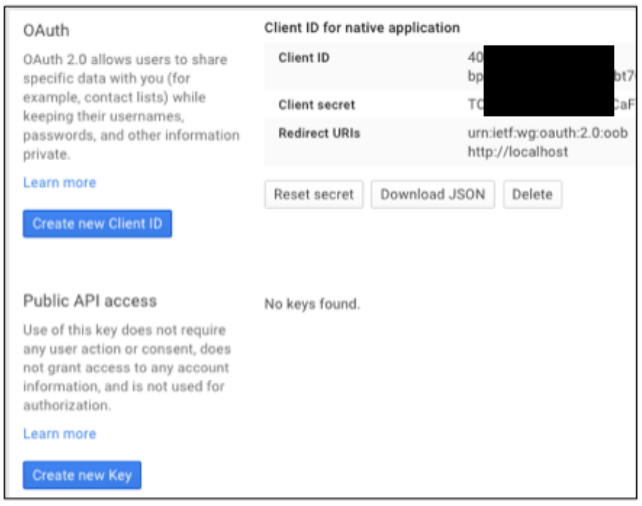

Figure 2: Google Console to create an application that will use OAuth

4.2 AWS

Amazon Web Services (AWS) is an extremely successful commercial cloud computing and data storage, delivery, and analytics suite of services. AWS powers several well-known online services, e.g., Netflix, Adobe, reddit, Docker, Globus, and more. For auth, AWS offers multiple options:

- email and password
- Identity and Access Management (IAM) username and password
- Access keys (access key ID and secret access key)
- Key pairs (public and private), only for a limited number of AWS services

In addition, AWS offers multi-factor authentication as an option.

AWS also provides a web-based interface (Figure 3) for managing one's keys and offers the following explanatory text for these keys:

> Access keys are also used with command line interfaces (CLIs). When you use a CLI, the commands that you issue are signed by your access keys, which you can either pass with the command or store as configuration settings on your computer.

AWS provides SDKs for multiple platforms and programming languages, including Python, Ruby, PHP, Java, and others (`http://aws.amazon.com/tools/`).

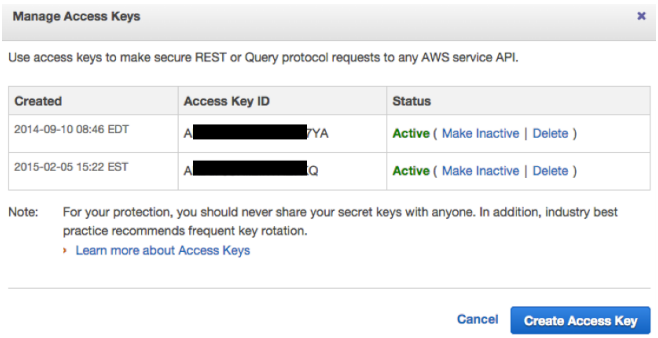

Figure 3: Web (HTTPS) interface for managing AWS access keys

And AWS also provides an OAuth option for applications:

> After users log in, they are returned to your website or mobile app. At this point, your client can obtain an access token by calling the Login with Amazon authorization service. That token allows clients to access the customer's name and email address from their customer profile.
>
> When you are granted an access token, you may also receive a refresh token. A refresh token is valid for longer than an access token, and allows you to trade in the refresh token for a new access token and a new refresh token.
>
> To access customer data, you must provide an access token to the Login with Amazon authorization service. An access token is an alphanumeric code 350 characters or more in length. Access tokens begin with the characters Atza|. Access tokens are only valid for sixty minutes and are specific to the user logging in and the data the app requested when it triggered the login. These access tokens are bearer tokens...
>
> Access tokens are returned in both the Implicit and Authorization Code grants.

4.3 GitHub

GitHub, as most software developers know, is a very successful hosting service for software repositories. It provides an API to its services and offers multiple auth approaches. The simplest auth is the *personal access token*, described in a GitHub help article[5]:

> When it comes to dealing with the API, personal access tokens work the same as OAuth tokens, and can easily be generated on GitHub.com.
>
> Personal access tokens are useful when it's too cumbersome to provide a client/secret pair for a full application, such as when authenticating to GitHub from Git using HTTPS, or within a command line utility or script.

GitHub also provides a web-based interface (Figure 4) for managing personal access tokens.

[5]https://help.github.com/articles/creating-an-access-token-for-command-line-use/

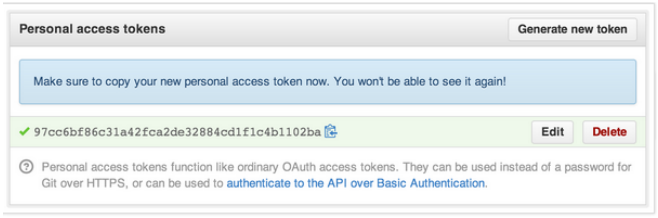

Figure 4: Web (HTTPS) interface for GitHub's Personal Access Token

But for production applications, GitHub recommends authentication via OAuth. From their developer documentation:

> While the API provides multiple methods for authentication, we strongly recommend using OAuth for production applications. The other methods provided are intended to be used for scripts or testing (i.e., cases where full OAuth would be overkill). Third party applications that rely on GitHub for authentication should not ask for or collect GitHub credentials. Instead, they should use the OAuth web flow.

4.4 Evernote

Evernote (evernote.com) is a cloud-based note taking, archiving, syncing, and sharing service. It is of interest to our work for multiple reasons:

- it is a very successful service and offers web, desktop, and mobile clients,
- it uses the Apache Thrift (RESTless) framework for its API,
- it provides both OAuth and developer tokens for authentication,
- it provides SDKs (for multiple languages/platforms) to assist in developing clients

In the following snippet of Python code, we see how easy it is to begin creating a Thrift client when an appropriate SDK has been provided.

```
import ssl
from thrift.transport import TSocket

class TSSLSocket(TSocket.TSocket):
   SSL_VERSION = ssl.PROTOCOL_TLSv1
```

When a developer is creating an application and needs to access only their personal account, they can use a developer token for authentication (Figure 5); otherwise, Evernote supports OAuth. Figure 6 depicts the OAuth flow for an Evernote application.

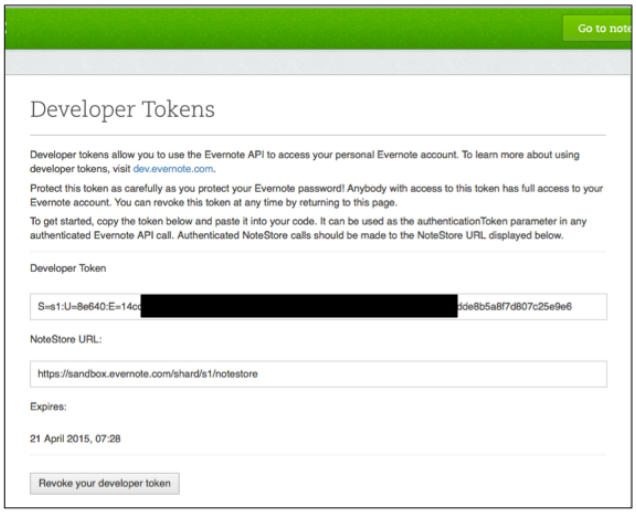

Figure 5: Web (HTTPS) interface for obtaining (or revoking) an Evernote token

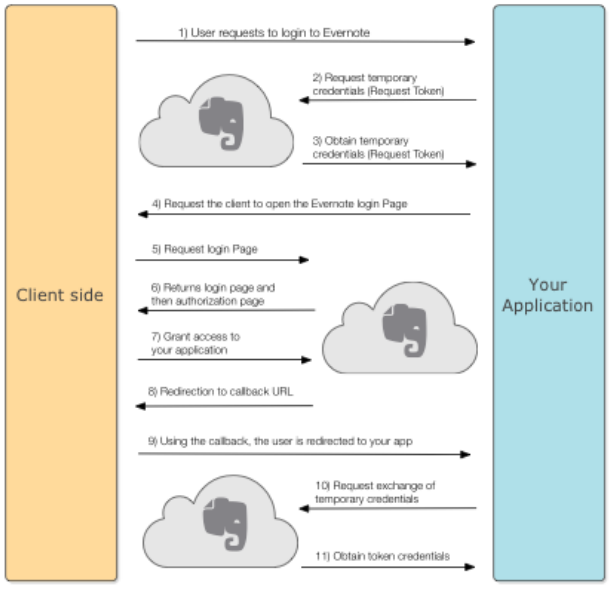

Figure 6: The OAuth flow for Evernote

5. RISKS

There are multiple risks associated with a project such as this. When one considers risks, it is usually in the context of security and certainly those are present and taken seriously here. There are, for example, risks associated with API keys being compromised, for various reasons. OAuth also has risks. The OAuth specification itself (OAuth2) has received some criticism for being overly complex. This may lead to implementations that are non-interoperable or, worse yet, insecure.

However, there are also risks associated with the usability of the API. If the API is difficult to use or lacks sufficient functionality, the project risks alienating users. Designing

and implementing an acceptable compromise between security and usability is a challenge for any service.

6. ARCHITECTURE AND BEST PRACTICES FOR AUTH

The SciGaP team will be concluding a design and initial implementation of an architecture for their service in the next few months.

We began this project considering three primary contenders for an auth solution: X.509, API keys, and OAuth. X.509 was ruled out primarily due to its complexity and because of the targeted clients. In addition to the traditional web portal science gateways, SciGaP would also be targeting native clients running on desktops and mobile devices. Many of these users would find X.509 to be overwhelming.

Using API keys was our preferred choice for awhile because they seemed to offer an acceptable compromise between security and usability. However, upon further consideration, the functionalities of API keys can be achieved as an authorization grant option within the OAuth specification. With increasing adoption of the OAuth2 standard, combined with SciGaP's wish to use an existing implementation over writing an API key implementation from scratch, we eventually selected OAuth as our choice for auth. SciGaP's eventual preference to incorporate the WSO2 Identity Server[6] into its architecture also affected our decision. This open source product has broad support for OAuth. The developers of both projects have a close working relationship which suggests that WSO2 may be willing to tailor some of their product's OAuth functionality to help satisfy SciGaP's needs.

Further, the OAuth2 specification naturally maps to SciGaP use cases. They will cover three grant flows discussed in the specification:

1. Gateways integrating with SciGaP and using the SciGaP provided identity management will follow the "Authorization Code Grant flow".
2. Gateways with native apps and mobile apps will use the "Implicit Grant flow".
3. Gateways managing their own users (and SciGaP unable to access this information) will need to employ the "Client Credentials Grant flow".

For completeness and to offer some advice to other projects that may choose to use API keys, we provide a list of best practices (some of which apply to OAuth also):

- have the service generate the API key
- deliver the API key in a web interface using HTTPS
- allow multiple API keys per user/client and allow deactivating them when one is potentially compromised or no longer needed
- regenerate API keys periodically (annually, at a minimum)
- do not embed API keys directly in code
- store API keys on the server hashed and salted
- provide easy to use SDKs in multiple languages
- provide example clients (using the SDKs) that demonstrate using TLS for the transport protocol

[6] `http://wso2.com/products/identity-server`

7. CONCLUSIONS

Traditional science gateways from ten years ago enjoyed a unique trust relationship between a web portal and resource providers and piggybacked on the Grid Security Infrastructure. The emergence of Cloud Computing resource providers requires integration with infrastructure APIs. In addition, science gateways are evolving into services and are themselves providing public APIs. These require us to revisit the trust relationships. For many online services, it is now commonplace to use either API keys or OAuth for authentication and authorization. API keys are, conceptually, easier to understand and use: a user obtains a (cryptographically secure) key and passes it into the API. However, there are significant burdens on both the user and the service to securely manage these keys. OAuth, although more complex, is widely supported and recommended by many popular online services. In addition, there has been a proliferation of libraries and frameworks, for both clients and servers, that implement the OAuth specification and thereby encourage its adoption. In this paper, we discussed these transitions and shared the experiences of the SciGaP project in arriving at a security solution. With the expert consulting and validation by CTSC, SciGaP is planning to support OAuth for science gateways. Its support on the server side will, at least initially, be provided by the WSO2 Identity Server, an open source product with a broad industrial and academic customer base. For client side support, SciGaP plans to bundle OAuth libraries into its various language binding SDKs.

8. ACKNOWLEDGMENTS

This work was funded by the National Science Foundation through the awards 1339774 and 1234408.

9. REFERENCES

[1] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2nd edition, 2008.

[2] J. Basney, V. Welch, and N. Wilkins-Diehr. Teragrid science gateway aaaa model: Implementation and lessons learned. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[3] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[4] R. Dooley, M. Vaughn, D. Stanzione, S. Terry, and E. Skidmore. Software-as-a-service: The iplant foundation api. In *5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, Nov. 2012.

[5] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, Aug. 2001.

[6] J. Gleick. *The Information: A History, a Theory, a Flood*. Vintage Series. Vintage Books, 2012.

[7] P. Gutmann. Pki: It's not dead, just resting. *IEEE Computer*, 35(8):41–49, 2002.

[8] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The ssl landscape: A thorough analysis of the x.509 pki using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM Conference*

on Internet Measurement Conference, IMC '11, pages 427–444, New York, NY, USA, 2011. ACM.

[9] T. A. Kanewala, S. Marru, J. Basney, and M. Pierce. A credential store for multi-tenant science gateways. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 445–454. IEEE, 2014.

[10] B. Leiba. Oauth web authorization protocol. *IEEE Internet Computing*, 16(1):74–77, 2012.

[11] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, and S. Weerawarana. Apache airavata: A framework for distributed applications and computational workflows. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 21–28, New York, NY, USA, 2011. ACM.

[12] R. Medrano. Welcome to the api economy. *Forbes*, Aug 2012.

[13] M. Pierce, S. Marru, B. Demeler, R. Singh, and G. Gorbet. The apache airavata application programming interface: Overview and evaluation with the ultrascan science gateway. In *Proceedings of the 9th Gateway Computing Environments Workshop*, GCE '14, pages 25–29, Piscataway, NJ, USA, 2014. IEEE Press.

[14] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal, The*, 27(3):379–423, July 1948.

[15] 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*, 19(6):893–904, 2007.