DAuth: Fine-grained Authorization Delegation for Distributed Web Application Consumers

Joshua Schiffman
Systems and Internet Infrastructure Security Laboratory
The Pennsylvania State University
Email: jschiffm@cse.psu.edu

Xinwen Zhang, Simon Gibbs
Computer Science Lab
Samsung Information Systems America
Email: {xinwen.z, s.gibbs}@samsung.com

***Abstract*—Web applications are becoming the predominant means by which users interact with online content. However, current authentication approaches use a single authentication credential to manage access permissions, which is too inflexible for distributed programs with unique security and privacy requirements for each component. In this paper, we introduce DAuth, an authorization mechanism that allows fine-grained and flexible control of access permissions derived from a single authentication credential for distributed consumers of web applications. We implement DAuth as a proxy for a Twitter social networking application within our distributed Elastic Application framework and find it introduces negligible overhead and requires only minor modification of existing applications. Through our evaluation, we demonstrate DAuth improves on existing web authentication mechanisms to support distributed web application consumers and can be implemented as a proxy to web applications that do not wish to develop their own implementation.**

I. INTRODUCTION

Application functionality is increasingly moving online as evidenced by the popularity of web applications such as Google Docs and Facebook applications [4]. These applications are no longer isolated. They now provide extensible APIs allowing social networks and other web application *consumers* to access user resources shared on those sites [23]. As web services become more intricate and highly interrelated, developers have begun to create distributed applications. Such applications are comprised of various services that enhance user experience and offer new and interesting content by mixing data from various sources. Examples include the so-called "mashups," which combine a user's social networking sites into a single aggregated feed.

While this integration inherently increases the value of individual services, the interaction is rarely sufficiently mediated, placing the applications and end users at significant risk. Developers frequently design programs that exercise a wide variety of web application functionality and users often authorize these third parties with little understanding of how much access they are giving up. What is worse, the granularity at which applications are authorized is typically far too broad, necessarily enabling full access to user accounts.

Traditionally, users have provided authorization to third party applications by entering their username and password, effectively giving complete control to the applications. With this approach being obviously dangerous, protocols have been developed to authorize third party applications without revealing such secrets [13], [5], [2], [12], [11]. However, these approaches assume that the consumer accessing the web service is a single identity. An application authorized to access user content will be given a single credential associated with the permissions delegated to it. As a result, all components in a distributed applications will be authorized with the full permissions of the application, regardless of whether they need them or not.

As developers leverage the cheap and easy to procure computational resources of public hosting environments like cloud computing platforms [1], [8], such overly authorized components may be run on public infrastructures in different administrative domains. Past studies have shown that these virtualized hosting platforms are vulnerable to attack [29], [26], and may lead to theft of access credentials. As a result, malicious entities can compromise user accounts by gaining access through systems to which they are indirectly authorized by the user.

To address this problem, we introduce DAuth, a mechanism for fine-grain sub-delegation of access permissions for distributed consumers of web applications. DAuth enables developers and users to specify the exact set of permissions that application components may have by generating new credentials from existing authorization protocols. Using a permission policy, DAuth can enforce least privilege by limiting application component permission according to the functionality required by the component and the level of trust associated with the location the component is hosted. The central component of our design is the DAuth agent, which creates, registers, and revokes access Sub-tokens for application components based on the application's permission policy. The service provider also exposes an interface for these Sub-tokens, which can be implemented with only small changes. Alternatively, a DAuth proxy can be used for web services that do not implement DAuth.

We developed and deployed DAuth on a framework for distributed programs called Elastic Applications [30]. This framework was designed to seamlessly shift application components between resource constrained devices and cloud computing platforms to conserve power and reduce costs. We evaluated our design by creating a Twitter application, whose functionality is split between a mobile device and a remote hosting platform. We demonstrate in our evaluation that DAuth adds minimal overhead of less than 6 ms when using a proxy and requires only minor modification of existing applications.

In this paper, we make the following contributions:

- We have created an approach for specifying the exact set of web-service-specific permissions that distributed application components may access based on the required functionality of the component and the safety of the hosting location.
- DAuth enables the application policy author to define permissions regardless of how broadly service providers currently authorize applications, which is typically on the order of read or write.
- Our approach is fully transparent to the user and will manage the delegation and revocation of application components without additional user input beyond the initial authorization of the application.
- DAuth introduces only minimal overhead to existing applications and complements popular authorization protocols like OAuth [11].

Outline: We start by providing background on current authorization approaches in web applications and define our problem and security requirements in Section II. Next, we give an overview of our DAuth design in Section III. We provide a brief overview of our Elastic Application framework and DAuth implementation in Section IV and evaluate our design in Section V with an example Twitter application. Section VI covers related work in authorization and we conclude with Section VII.

II. Authorization in Web Applications

Web applications revolve around user-generated content. A key concern for both users and application providers is how to manage access control of data stored in the application. In this section, we discuss current approaches used by web applications for authorizing access to user data. Next, we will demonstrate how distributed applications violate the basic assumptions of current protocols. Finally, we detail our security requirements for a solution to authorizing distributed applications.

A. Authorization Background

Increasingly, Web 2.0 applications expose functionality through a set of APIs that enable both installed applications (i.e., desktop programs or mobile phone apps) and other web applications such as mashups to access user generated content on behalf of a user. For example, Twitter [15] provides an API that allows applications to manage a user's status updates or integrate user feeds with other social networking services. While some users may publish their content publicly, it is often desirable to protect content from unauthorized clients. Current approaches for managing access to user data can be classified into two broad categories: 1) Authentication and 2) Access Authorization.

Web applications use authentication techniques to validate a client's identity, which gives that client access permissions

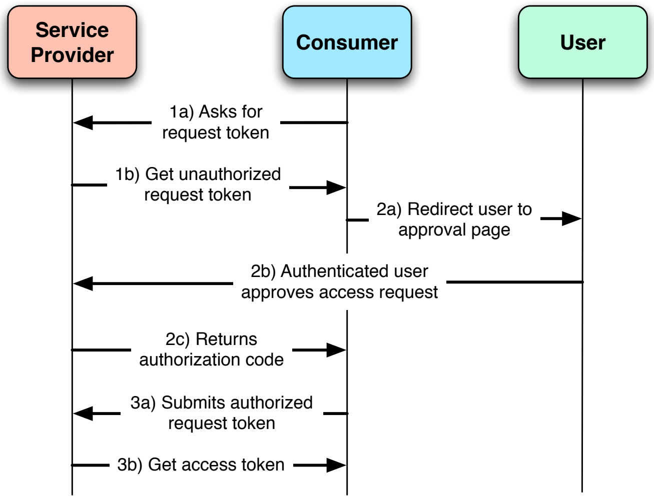
Service Provider
Consumer
User
1a) Asks for request token
1b) Get unauthorized request token
2a) Redirect user to approval page
2b) Authenticated user approves access request
2c) Returns authorization code
3a) Submits authorized request token
3b) Get access token

Fig. 1. **OAuth [11] protocol:** The consumer requests permission to access a user's protected resources stored at the service provider. The provider gives the consumer a request token, which the user approves and is exchanged for an access token. This gives the consumer the necessary permissions to access content on behalf of the user.

associated with that account. The most basic form of authentication is using a combination of username and password. Unfortunately, users must create and manage often-redundant account information across the numerous web applications they frequent. An alternative is to use an identity provider to manage the process of authenticating users to web applications. Federated authentication approaches such as Shibboleth [14] centralize authentication for multiple applications to a single point, but are restricted to services within a particular administrative domain. Single sign-on services like Microsoft Passport [10] and Google Accounts offer a single identity management point while OpenID [12] offers an open and decentralized standard for authenticating users. In fact, several large sites such as Google, Yahoo, and MySpace already act as OpenID providers.

While authentication approaches are effective for identifying users, they are less useful for applications that act on their behalf. For example, a mobile phone application should not necessarily have all the access permissions of the user it represents. Moreover, providing third party applications with login credentials raises the risk of identity theft through leaked secrets. In order to avoid giving login credentials to third parties, authorization protocols, most notably OAuth [11] and Google's AuthSub [2], can be used to delegate limited permissions to third parties without giving away user credentials.

We will focus on OAuth as it is an open standard that is largely adopted by popular web applications such as Google, Facebook, and Twitter. The OAuth standard defines a *consumer* as a website or application that accesses content at a *service provider* (i.e., the web application) on behalf of a user. Consumers first obtain a consumer key and consumer secret from the service provider. The key identifies the consumer making a request and the secret is used to authenticate the consumer. Once registered, a consumer can request access to a user's account by following the OAuth protocol shown in Figure 1.

The protocol involves three key steps:

1) The consumer obtains an unauthorized request token that describes the resources the consumer wants to access.
2) The consumer directs the user to a service provider authorization page where an authenticated user can approve the request. Upon approval, the consumer is given a verification code to authorize the request token.
3) The token is exchanged for an access token, which grants a set of access permissions for protected resources on the service provider.

When the consumer performs API calls to the service provider, it signs each HTTP request and the parameters with the consumer secret and a secret included with the Access token using one of several signature methods such as HMAC-SHA1 or RSA-SHA1. This binds the request to a specific consumer and set of access permissions. This authorization information can be encoded in the HTTP Authorization header, the body of a HTTP POST request, or as parameters in the URL. The user may later make a request to revoke the access token or the provider can optionally provide some time limit.

B. Problem Definition

We expand on our threats with an example scenario and then describe the challenges distributed consumers cause for authorization protocols such as OAuth. In previous work, we developed a framework for designing applications that move heavy computation off resource constrained handheld devices onto cloud computing architectures such as Amazon's EC2 [1] and Microsoft's Azure [8]. We call applications that run in this framework, Elastic Applications [31] or EAs. These EAs are comprised of small programs, named *weblets*, which can be spawned either locally on a device or remotely on a cloud environment depending on resource constraints.

Consider the social translator EA in Figure 2. The application subscribes to a user specified social network and translates status updates from a foreign language to another language. The user can then repost the translated updates back to the social networking site. The application divides the translation and posting logic into two separate weblets so that code for polling the site for updates and translating them can be run on the cloud instead of the device.

While we trust both the application and service provider to behave correctly and not leak protected resources, weblets running in environments outside of the user's control pose a potential risk. Previous work has shown that hosting environments such as clouds [26] are susceptible to compromise [29], which may intercept network traffic, snoop memory [24], or modify file contents on disk. This can lead to theft of the weblet's access token and secrets. Hence, it is desirable for applications to limit the permissions available to components running in untrusted environments.

Unfortunately, authorization approaches such as OAuth assume the consumer is always a single entity, which gives all components using the consumer's access credentials the same

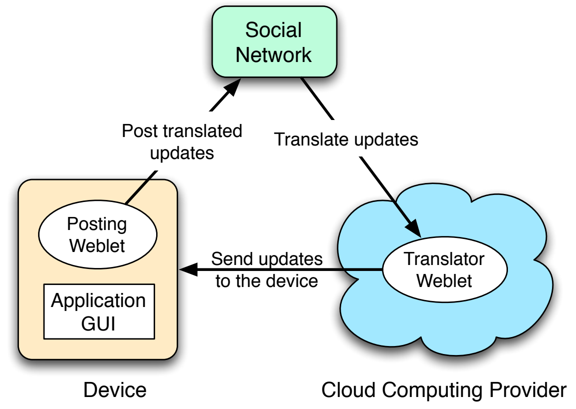

Fig. 2. An example distributed social network consumer. The application is divided into a translation weblet and a posting weblet. Status updates are read by the translator and converted to the user's language. The user then chooses which translated updates to post back on the social network for his friends.

permissions to access user data. In the case of our translation EA, both weblets will be given permission to read status updates and post new ones because the application requires both permissions to function properly. This is despite the fact each weblet needs only a subset of those permissions. Another issue is that OAuth relies on the service provider to define the permissions the consumer can request. Unfortunately, most providers only define a broad set of access rights such as complete access to data or read only. This makes it impossible for an application developer to adhere to a least-privilege policy when deploying application components. Another issue is revocation of delegated permissions. Most service providers either require the user to manually invalidate the access token or have it expire after some time. This widens the damage a stolen access credential may cause since it will persist if the user is unaware of the theft.

C. Solution Requirements

In order to address these issues, we outline a series of requirements that a solution for delegating permissions should meet for distributed web application consumers.

Fine-grain assignment of permissions: An authorization mechanism must allow an application to sub-delegate the specific permissions that its components require from its full set of permissions. This means being able to select the application-specific API calls without granting additional unnecessary rights (i.e., read updates instead of read everything).

Automated sub-delegation and revocation: While it is expected for the user to manually authorize the consumer application, it is unreasonable to require the user to authorize each consumer component individually. A solution must be able to issue and revoke sub-delegated permissions in an automated fashion.

Policy driven delegation: An application policy should describe the amount of access granted to components. This enables application developers and users, not service providers, to specify a level of trust for each portion of the application and the environment in which they run. For example, a user may give components on a mobile device full permissions, but give a restricted set to code running on a public cloud.

Minimal and incremental changes to existing architectures: A new authorization approach should require only minimal changes to existing service providers.

III. DAuth Policy

We now introduce DAuth, an approach for sub-delegating permissions in a distributed consumer application. DAuth is comprised of three components, 1) a *DAuth agent*, 2) a *permission policy*, and 3) the service provider *sub-token interface*. The DAuth agent runs in the consumer and transparently manages the issuing and revocation of access sub-tokens to application components. These tokens authorize the access of protected resources at the service provider. The permission policy defines the subset of rights that may be grant to components. When consumer components are executed, the DAuth agent sends a request for a sub-token to the service provider's sub- token interface with the permissions defined in the policy. Later, the agent can send a request to revoke that sub-token when the component is no longer in use.

In this section, we first explain our assumptions and threat model. Next, we describe the DAuth components are used in an authorization protocol. Finally, we discuss the permission policy and how it is defined and used.

A. Assumptions

In our design, we assume the application and service provider is implemented correctly and is not malicious. We also trust the user's client such as a browser for web application consumers or for installed programs. We assume the authorization protocol is correct and do not consider attacks on the cryptography it uses. Furthermore, we trust that the hosting environment for remote components will not intentionally leak secrets or modify code. However, our threat model assumes an attacker that can intercept and modify network traffic and compromise public hosting environments like clouds to steal secrets, snoop memory, or modify disk contents of the host.

B. Protocol

Recall from Section II-A, the OAuth protocol gives the consumer an access token to interact with protected resources on the service provider on behalf of the user. The DAuth protocol augments this process so the application can request sub-tokens with a subset of access token's permissions. Figure 3 describes the DAuth protocol among the user, service provider and the application components.

In the first phase, the consumer obtains an unauthorized request token for the consumer application as normal. The user is then redirected by the consumer to the service provider's approval page. If the user approves the request, the consumer is issued an access token, which we refer to a master token. The master token authorizes the consumer with a set of permissions, $M = \langle p_0, p_1, \ldots, p_n \rangle$, where p is an authorized operation.

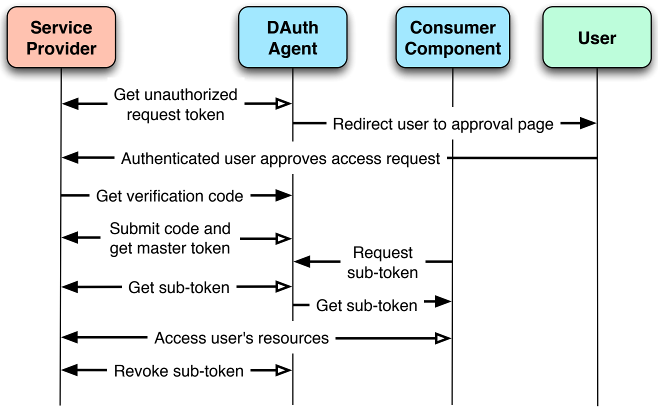

Fig. 3. **DAuth Protocol:** The DAuth protocol begins the same as OAuth with the DAuth agent obtaining the access token from the service provider. Instead of giving this token to the consumer components, the agent registers Sub-tokens with specific access permissions for each component. The agent later revokes the Sub-token when it is no longer needed. Here, white arrowheads represent a response by the recipient of the message.

In phase two, the application initiates its components. For each component c running in location l, the DAuth agent uses the policy P to obtain the subset of permissions the component may have, s_c, where $s_c \subseteq M$. The policy is described further in Section III-C. Next, the agent sends a request to the service provider for a sub-token with permissions s_c. The service provider generates a new sub-token, stores it in association with the consumer's account, and returns the token to the agent. The sub-token is then passed to c, which uses it in place of an access token to access user resources on the service provider. This phase is repeated for each newly created c during the lifetime of the consumer or until the master token is revoked.

The protocol enters phase three when a component c finishes and is no longer needed. First, the DAuth agent sends a revocation request to the service provider for c's sub-token. The provider then deletes the sub-token from the consumer's account and responds with an acknowledgment. From then on, c can no longer access user data with permissions s_c. Note that if the master token is revoked by the user, by extension all sub-tokens are also revoked.

C. Permission Policy

The application permission policy defines the rights each application component may be delegated. The policy enables the application developer and user to express the required permissions for each component and limit permissions that can be delegated for specific environments. The granularity of permissions the policy can express is limited to the operations the components can perform. Since many web services expose APIs through a RESTful interface [25], access operations can be identified by a URL. Policies can be more broadly defined by the type of HTTP request performed (i.e., GET, PUT, POST, DELETE), which many web services already use to classify write (POST) versus read (GET) operations.

Our policy design is motivated by the need to: (1) define the

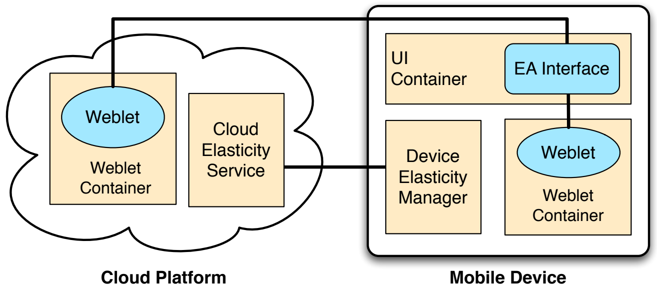

Fig. 4. **Elastic Application [30] infrastructure:** The Cloud Elasticity Service (CES) and Device Elasticity Manager (DEM) cooperate to start, stop, and manage EA components on the device and cloud. The EA is made up of a UI running in the device, which accesses weblets running in weblet containers.

set of necessary and desired functions a component can perform and (2) limit what functions are performed in untrusted environments. To do this, we created our policy to specify for each component c, the set of permissions, f_c, needed for complete component functionality. Optionally, the policy author may also specify, r_c, the set of required permissions c needs at a minimum in order to run, where $r_c \subseteq f_c$. Conceptually, one can imagine a program that requires read access to user data to function at all, but additional features would be enabled if write access is give as well. Each location, l, has a set of permissions a_l that are allowed to be delegated while running in that location. When the DAuth agent needs to assign a sub-token to a component, it obtains $s_c = f_c \cap a_l$. If $r_c \subseteq s_c$ and $s_c \subseteq M$, the agent can request a sub-token for that subset of permissions. For components whose $r_c \not\subseteq s_c$, either a new location must be selected for the component or it is denied a sub-token and will not have access to the user's resources. Users can also configure the DAuth agent to attempt to launch components at locations that allow all or portion of the optional permission in f_c beyond those in r_c.

Since web application consumers come in two forms (installed application and web application), we envision two use cases for designing and verifying the policy. For installed application consumers, the user can inspect the application's permission policy file locally for approval and make any changes they wish like restricting the permission set for specific locations. For web application consumers, the consumer should present the policy to the user for inspection and modification before the consumer is authorized to access the user's resources.

IV. DAuth Architecture

We evaluated our DAuth design by implementing it within an experimental application SDK for building distributed applications, called Elastic Application (EA). These applications are comprised of various autonomous components called weblets, which are intelligently deployed on the device and cloud hosting platforms based on objectives such as minimizing power usage or maximizing performance. We chose to implement DAuth within the service that manages the deployment of weblets because it acts as a reference monitor for authorizing weblet permissions. In this section, we first give an overview of the Elastic Application infrastructure and then describe our DAuth implementation.

A. Elastic Application Overview

In order to better understand our implementation, we give a brief description of the Elastic Application framework. While the full details are unnecessary for this discussion, they can be found in our previous work [30]. An EA is a UI component and one or more weblets packaged along with a manifest, which includes signed hashes of the package contents and other configuration information. Weblets are self-contained functional components that communicate to each other using HTTP(S). Weblets are executed in an application VM (such as Java or a Python interpreter), called a weblet container. At launch-time, the UI is instantiated for the user. As functionality is requested, weblets are spawned by the EA infrastructure to perform processing. As process demand increases or resources become limited, the weblets can be transitioned to a remote cloud platform.

Figure 4 shows an overview of the EA framework. An EA is supported by two key components, the *Device Elasticity Manager* (DEM) and the *Cloud Elasticity Service* (CES), both of which manage the EA component execution and migration between device and cloud platforms. The DEM consists of a set of services that runs locally on a consumer device, such as a mobile phone, and is responsible for configuring applications at launch time, making configuration changes during run-time, and launching or transitioning weblets to a cloud platform. The CES resides on a cloud and is used by the DEM to manage weblets and resources on the underlying platform.

We have developed an SDK to help developer write programs that leverage the infrastructure. We also developed a set of example applications for smart phone systems (Android and Windows Mobile) and Amazon EC2 for the cloud component. For example, an EA could leverage cloud resources to perform image processing, which is too computationally demanding for typical phones. Another possible application could send snapshots from a phone's camera to weblets on a cloud to perform image recognition. The resulting pixel locations of the object are returned to the device's tracking weblet, which performs motion tracking of those points and renders an informative message on top of the object.

B. DAuth for Elastic Applications

Our DAuth implementation consists of three parts as illustrated in Figure 5. The first is the *Sub-token Delegation Service* (SDS) that is integrated into the DEM on the client device. The SDS acts as the DAuth agent and exposes an interface for EAs to register OAuth access tokens and for creating, registering, and revoking sub- tokens at a target service provider on behalf of the application. Adding the SDS to the DEM required about 50 extra lines of code. At install time, the EA registers their manifest containing the service provider's REST API and the EAs permission policy. We implemented the permission policy as an XML document and give an example of one in Section V-A.

Next, we extended the standard OAuth protocol with two API calls. The first, `/register_subtoken`, is an authorized call that takes a freshly generated sub-token (OAuth access token) and set of access permissions (list of service provider URLs) the token is authorized to use. Upon receiving the call, the service provider associates the sub-token with the consumer's access token authorized by a particular user and returns an identifier to the sub-token. The second call, `/revoke_subtoken`, takes the sub-token identifier and deletes it from the set of sub-tokens for that consumer's access token.

The last component we implemented is a DAuth proxy. The proxy is a small HTTP server (about 500 lines of Python) in the CES that sits between the service provider and the consumer, in order to implement the DAuth sub-token interface. This component is necessary for service providers that do not support DAuth. Instead of interacting with the service provider, the SDS communicates with the proxy via HTTPS for all Sub-token management and stores a copy of the master token on the proxy. We modified each weblet container to access the proxy instead of the service provider's domain. The proxy intercepts all requests and inspects the authorization header of the request. Specifically, the sub-token and API call are compared to the set of permissions for that token in its list of registered tokens. If the sub-token is authorized, the proxy replaces the token in the header with the master token (OAuth access token) and resigns the request. Otherwise, the request is passed along as normal, which will be rejected by the service provider. This allows the provider to return an error message regarding a failed authorization attempt directly to the weblet.

C. SDS Protocol

We now describe each step in our DAuth protocol implemented in Figure 5. 1) The user first authorizes the consumer application using the standard OAuth protocol described earlier in Section II-A. As a result, the application is returned an OAuth access token. 2) The application UI then registers the master token with the SDS, which resides within the DEM's set of services. In our implementation, we created the SDS as a set of XMLRPC calls the EA can perform on the DEM. In addition to the access token, the DEM possesses a definition of the service provider's API calls and the application's consumer key and secret. Since the EA framework requires applications to be installed before use, we designed the DEM to store this information at EA install time. Next, 3) the SDS registers the access token with the DAuth proxy.

During the course of the application's use, the DEM will be instructed to launch a weblet. In order to authorize the weblet, the SDS references the policy to determine the set of API calls the weblet may perform and then generates a Sub-token by taking the SHA-1 sum of a random nonce. It then 4) registers the Sub-token and the allowed API URLs at the proxy.

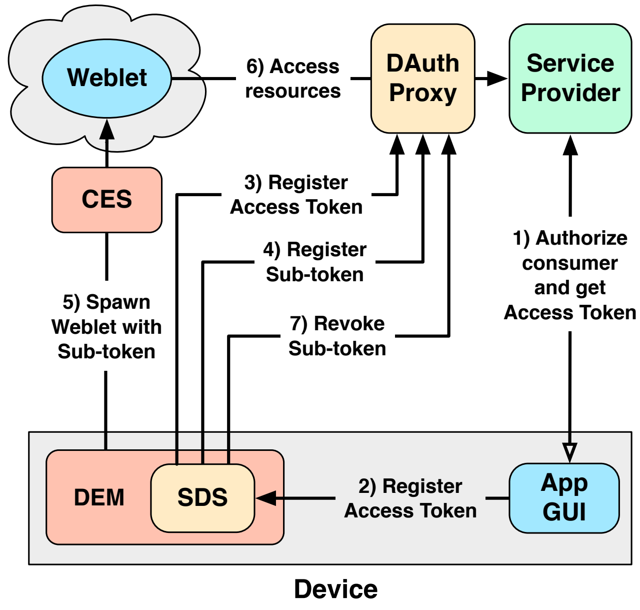

Fig. 5. Implementation and authorization protocol for DAuth in an Elastic Application framework. After obtaining the OAuth access token, the SDS registers it with the DAuth proxy. Later, the SDS generates and registers Sub-tokens for new weblets, which allow the weblets to access resources on the provider. The SDS revokes the Sub-token when access is no longer necessary.

For weblets that run on a remote host, the 5) DEM contacts the CES to spawn a weblet with the newly generated Sub-token. The Sub-token is included with this call to the CES, which is encrypted using HTTPS. For weblets running on the cloud, 6) the weblet accesses protected resources at the service provider via the DAuth proxy. When the proxy receives the request, it first examines the Sub-token used and the associated API calls registered for it. If the Sub-token grants the weblet access, the proxy replaces the Sub-token used in the Authorization headers of the HTTP request with valid credentials. This means using the access token and resigning the request with it. For weblets that are running on the device instead of the cloud, the DEM acts as the proxy.

Finally, when the SDS is required to revoke a Sub-token, 7) the proxy is contacted and told to delete the Sub-token. This revocation can occur for several reasons. The first, is when the application is shutdown. The second is when a specific weblet is shutdown. This may occur when the functionality of the weblet is no longer needed.

V. Evaluation

We evaluated our system by developing a Twitter monitor EA. The application monitors a user's Twitter account and sends status updates to the user's device when new updates are posted by people they are following. We divided the EA into a monitoring weblet that runs on a cloud platform and a posting weblet on the device that lets the user reply or repost updates retrieved by the monitor weblet. First, we will describe the EA and examine the overhead on performance and additional coding that was required to support DAuth. Then we analyze how effective DAuth is at handling sub-delegation of permissions.

For ease of development, we created both the DEM and CES as python XML-RPC servers, which communicate via HTTPS. Our DEM was deployed on a netbook with 1GB RAM and a 1.6GHz Intel Atom CPU and the CES was on an Amazon EC2 AMI. Both systems ran Ubuntu Server 9.04.

A. Twitter EA

As described in Section IV-A, an EA contains a UI, weblets, and a manifest. Our Twitter EA is comprised of two weblets and a GUI written in Python and using wxWidgets for the interface. The first weblet is the monitor, which constantly polls a user's Twitter account for new content and sends the results to the application's UI. The monitor is intended to run on the cloud where network access and power consumption is not an issue compared to a mobile device. The second weblet is the poster, which allows the user to post new updates or responses to updates the monitor detects. For all weblets, we hard code the EA's consumer key and secret.

For the EA package's manifest, we created an XML document. Figure 6 is a portion of the manifest showing the relevant permission information. We note the DAuth policy could be implemented within a more sophisticated policy framework like XACML [3]. However, we chose to build a more simple and tailored approach that avoids the overhead of a large enforcement engine. For each weblet, a location and permissions tag is present. The location tag specifies the desired or required (indicated by the required attribute) location where the weblet should run. The permissions set contains application specific permission tags. The required attribute defines what permissions must be given to the weblet if it is to function at all. Twitter defines API calls similarly using POST requests as writes and GET requests as reads. We took a similar approach and divided the permissions into two broad categories: 1) `READ` and 2) `WRITE`. Since our monitor weblet only requires permission to read status updates, we only assign read permissions with the `<twitter:permission type="READ"/>` tag. By contrast, the poster weblet needs only write permissions, but may optionally have read as well.

We first designed the application to use OAuth as normal. We found the average time it took the monitor to read the user's Twitter timeline to be approximately 439 ms. We then converted the EA to support DAuth, which took roughly 25 lines of code. This added about 5.23 ms (1.19%) to the request time with only 1.76 ms due to processing at the proxy. Thus, DAuth proxy only added only minimal overhead to the EA, mostly due to network delay.

B. Security Analysis

We now analyze the how access to user resources is protected by DAuth in our example Twitter EA. As stated in Section III-A, we do not consider attacks on the device itself or preventing malicious applications from intentionally leaking credentials. We also trust the EA framework, the DAuth proxy, and the Twitter to be bug free and correct. Finally, we focus

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:ea="http://example.com/elasticApp"
          xmlns:twitter="http://ea.twitter.com/">
    <ea:application name="elasticTwitter">
        ...
        <ea:weblet name="Monitor">
            <ea:location required="false">
                Cloud
            </ea:location>
            <ea:permissions required="true">
                <twitter:permission type="READ"/>
            </ea:permissions>
        </ea:weblet>
        <ea:weblet name="Poster">
            <ea:permissions required="true">
                <twitter:permission type="WRITE"/>
            </ea:permissions>
            <ea:permissions required="false">
                <twitter:permission type="READ"/>
            </ea:permissions>
        </ea:weblet>
        ...
    </ea:application>
</manifest>
```

Fig. 6. Manifest for Twitter EA

on an attacker that can intercept and modify network traffic and compromise the public cloud that hosts remote weblets.

In the normal EA framework, both the monitor and poster weblets would be required to use the EA's access token for all API calls to Twitter. As a result, the monitor weblet would be given permission to use both read and write calls. Given our threat model, an attacker can compromise the monitor's hosting environment and steal the access token and Consumer Secret. This would let the attacker gain both read and write access to the user's account until the user revokes the access token. Revoking the access token would require the user to reauthorize the entire application.

In our DAuth enabled system, the monitor would only have access to the Sub-token associated with read permissions. Theft of the Sub-token would only allow the attacker to perform read operations through the DAuth proxy assuming Twitter does not implement the DAuth interface directly. To reduce the window of access the Sub-token provides the attacker, the DEM can shutdown the monitor weblet and restart it periodically with a fresh Sub-token after revoking the old one. This invalidates the potentially stolen Sub-token without requiring the user to reauthorize the application. While DAuth cannot prevent the theft under our attack model, it does limit the permission of weblets running in vulnerable environments to the set defined in the EA's manifest. This enables policy authors to create more conservative permission policies that minimize the number of sensitive operations performed on untrusted hosts.

VI. Related Work

Today's web applications divide code into server-side and client-side portions. Typically, the server-side portion of the code contains the most security critical parts because the client-side runs in an entirely untrusted environment. Malware or malicious users can modify the code and subvert the code's behavior. Previous work on preventing unauthorized data leakage in web applications have focused primarily on securing this untrusted portion of web applications and on managing authentication credentials.

In the domain of securing client side code, effort has been made to ensure browser-run portions behaves as expected. Since client-side code is typically written in JavaScript, several tools have become popular in generating browser- run code from other languages. These frameworks perform "tier-splitting" in order to take fully designed application code and produce the server and client portions. Popular frameworks include GWT [6], Links [19], Hop [27], and Volta [9]. These frameworks aid in creating less buggy code, but provide little guarantee that the client side is unmodified at runtime.

Recent research has adapted these frameworks to achieve additional security guarantees. The Swift [18] compiler tier-splits a Java application so that the server never depends on the client for security critical information. To do this, the programmers must annotate their code with information-flow labels, which is then compiled in Jif [7]. The resultant program, if compiled, is guaranteed to adhere to an information flow policy. From there, the program is tier-split using GWT. Ripley [28] uses the Volta compiler to tier-split a .Net application similarly to Swift, but without the use of annotation. In addition, Ripley allows the server to replicate the client side code concurrently with the browser in order to detect modified browser-side code. Unfortunately, these techniques require special compilers or code modification and fail to detect compromised clients that behave within the specification of the code.

However, the permissions defined in such authentication schemes are often too coarse, giving applications a large majority if not all permissions. Delegation Logic [22] defined a language for describing sub-delegation of permissions. Other research [17], [16], [20] investigated using X.509 certificates to delegate permissions using signed capabilities. However, public key cryptography and certificates are not frequently used in web applications environments. One proposed solution uses a trusted third party to issue plain text delegation permits [21] that users can inspect to know what permissions are being delegated to mashups. DAuth does not require such a third party and maps specific permissions to sub-delegation tokens.

VII. Conclusion

In this paper, we presented DAuth, a mechanism for authorizing fine-grain sub-delegation of web application access permissions in distributed applications. Where existing web authorization standards like OAuth fail to provide application-specific permission policies, DAuth extends them to enable developers and users to specify the exact set of API calls the each application component may use. For each component that uses a shared credential to access protected resources, DAuth manages the task of assigning and revoking capabilities that provide a policy-defined set of permissions to that component. DAuth is designed to function as both an extension at the service provider's endpoint and as a proxy that can be deployed within a private environment. Such a proxy allows applications to use DAuth without requiring the service provider to adopt it and within the confines of a protected installation. We implemented our design on our Elastic Application framework for resource constrained devices and evaluated its performance with a custom Twitter application. We found DAuth introduces only minor overhead to its performance and requires few code changes to make code DAuth aware.

References

[1] Amazon ec2, http://aws.amazon.com/ec2/.
[2] Authentication for Web Applications, http://code.google.com/apis/accounts/docs/authforwebapps.html.
[3] eXtensible Access Control Markup Language, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.
[4] Facebook developers, http://developers.facebook.com.
[5] Flickr authentication api, http://www.flickr.com/services/api/auth.spec.html.
[6] Google web toolkit, http://code.google.com/webtoolkit.
[7] Jif: Java information flow, http://www.cs.cornell.edu/jif/.
[8] Microsoft azure, http://www.microsoft.com/windowsazure.
[9] Microsoft live labs volta, http://labs.live.com/volta.
[10] Microsoft passport network, http://passport.net.
[11] Oauth, http://oauth.net.
[12] Openid, http://openid.net.
[13] Registration for web-based applications, http://code.google.com/apis/accounts/docs/registrationforwebappsauto.html.
[14] Shibboleth, http://shibboleth.internet2.edu.
[15] Twitter api, http://apiwiki.twitter.com.
[16] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust-management system, version 2, ietf rfc 2704, 1999.
[17] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of IEEE Symposium on Security and Privac*, 1996.
[18] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *In SOSP*, 2007.
[19] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects*, 2006.
[20] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory, ietf rfc 2693, 1999.
[21] R. Hasan, M. Winslett, R. Conlan, B. Slesinsky, and N. Ramani. Please permit me: Stateless delegated authorization in mashups. In *Proc. of ACSAC*, 2008.
[22] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, (1), 2003.
[23] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big web services: Making the right architectural decision. In *Proc. of 17th International World Wide Web Conference*, 2008.
[24] B. Payne. XenAccess. http://code.google.com/p/xenaccess/.
[25] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.
[26] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off my cloud! exploring information leakage in third-party compute clouds. In *ACM conference on Computer and Communications Security*, 2009.
[27] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Companion to the Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006.
[28] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing web 2.0 applications through replicated execution. In *Proc. of the Conference on Computer and Communications Security*, 2009.
[29] R. Wojtczuk. Subverting the Xen hypervisor. www.blackhat.com/presentations/bh-usa-08/Wojtczuk.
[30] X. Zhang, S. Jeong, A. Kunjithapatham, and S. Gibbs. Towards an Elastic Application Model for Augmenting Computing Capabilities of Mobile Platforms. In *Third International ICST Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, 2010.
[31] X. Zhang, J. Schiffman, S. Gibbs, A. Kunjithapatham, and S.Jeong. Securing elastic applications on mobile devices for cloud computing. In *Proc. of ACM Cloud Computing Security Workshop*, 2009.