Web-based single sign-on: an examination of security and usability

Samuel Waters

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Web-based Single Sign-On: An examination of security and usability

By

Samuel R. Waters

Committee:
Charles Border, Chair
Thomas J. Borrelli
Harris Weisman

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Networking and System Administration

Approved November 16, 2012

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences
# Table of Contents

Abstract ........................................................................................................................................... 3

Overview

  Research motivation .................................................................................................................... 5
  Single Sign On basics .................................................................................................................. 9

SSO Components

  OAuth .......................................................................................................................................... 13
  SAML ........................................................................................................................................ 23
  OpenID ....................................................................................................................................... 27

SSO Configurations

  Shibboleth ................................................................................................................................... 33
  Amazon ...................................................................................................................................... 36
  Social Network Providers ........................................................................................................... 39

Discussion

  Security ....................................................................................................................................... 42
  Usability ..................................................................................................................................... 47
  Benefits ...................................................................................................................................... 52
  Future Needs ............................................................................................................................... 53

Conclusion ..................................................................................................................................... 55
Abstract

Among system administrators, users are generally considered the weakest link in the security chain. One of the biggest concerns is how users create and remember passwords. To try and ensure the security of their systems, administrators will generally create policies regarding how complex a user’s password must be, and will exhort the user to never write down, share, or reuse that password. On the surface, this seems like good security sense, and from an administrator’s perspective, it is. However, to the user, who may have dozens of unique accounts, stringent policies create a significant cognitive burden. As such, users are generally guilty of creating ad hoc solutions to remember passwords, such as making them as simple as possible, writing them down, or reusing them. Administrators react by changing policies to make passwords even longer and more complex, and a negative circle is created. An alternative is to make use of Identity Federation (IF) systems. These systems allow end users to authenticate using a single password, thereby reducing the overall cognitive burden. This work will discuss the different technologies currently in use, and examine if these systems can provide adequate security while improving overall usability.
OVERVIEW

Research Motivations

User generated passwords are generally considered to be the weakest link in a secured system. It is well documented that users will choose the weakest password allowed. (Adams & Sasse, 1999; Florencio & Herley, 2007; Hayashi & Hong, 2011) Adams and Sasse (1999) found that most users were unaware of what constituted a good password. This finding was reinforced by Florencio and Herley’s 2007 large-scale study, which found that the vast majority of passwords are comprised solely of lower case letters. This tendency causes administrators to create complex policies regarding password strength, complexity, age, and history.

It’s not that users aren’t concerned with security, just that they have different priorities than administrators. (Adams & Sasse, 1999) They wish to accomplish a particular task, and authentication is a necessary barrier to completing that task. Policies that require complex, lengthy passwords and frequent password changes may cause users to create stronger passwords (Florencio & Herley, 2007), they also create a distinct burden on users.

Human short-term memory is only capable of remembering about seven distinct items. (Miller, 1956) Self-selected passwords are easier to remember than machine generated strong passwords, but all users have difficulty remembering passwords to some degree. (Yan, Blackwell, Anderson, & Grant, 2004) Inglesant and Sasse (2010) found that users found it problematic to create new passwords that meet with the system rules, and
that frequent password changes made remembering passwords extremely difficult. This leads to a “genuine fear of forgetting”, because there is a time delay in resetting a forgotten password, and therefore a loss of productivity.

This fear is compounded by the sheer number of accounts users have. Florencio and Herley (2007) estimate that each user has approximately 25 online accounts, and that the longer an individual has had Internet access, the more accounts that user owns. Hayashi and Hong (2011) found in their own research that only about 75% of logins are web based, which suggests that users in total have on average at least 32 accounts-and none of this research includes logins from mobile devices such as smart phones. Finally, users are exhorted to never write down or share their passwords, to not reuse passwords on different sites, and to change their passwords frequently.

This puts users in an awkward position, trying to remember a large number of unique passwords with varying degrees of complexity. This leads to a situation where users must rely on an external method for remembering passwords, such as writing down passwords, reusing the same password, or making slight variations on the same password. Shay et al (2010) found that users were more likely to reuse their passwords than write them down, and that they were more likely to modify an old password than create a new one. To the users, these ad hoc approaches are a completely rational response. When faced with policies that reduce productivity, users will do what they feel is necessary to get their task done. In most cases, they see a very small risk associated with the unauthorized access of their accounts. Generally, the user loses very little, even when financial accounts are involved. Therefore, as there is little incentive to work at protecting accounts, users do not see any value in putting much effort into making their passwords secure. (Herley, 2010) To
those charged with securing systems and assuring the security of information, this reduces the effectiveness and overall security of the systems in question.

The current situation as it stands is untenable, and authentication and access schemes need to be rethought with an emphasis on usability as well as security. One option to reduce the burden on users is to make use of an Identity Federation (IF) system. Using an IF system would significantly reduce the burden users currently carry in creating and remembering passwords. Initial research has indicated that users are enthusiastic about the opportunity, provided they understand the concept. (Heckle, Lutters, & Gurzick, 2008)

Of course, there is an argument to be made against using IF systems. Password reuse is a troublesome problem, simply because password theft and accidental password exposure happens with alarming frequency. Once a username/password combination is “in the wild”, they can become part of a database of known good authentications, and all the accounts where the user has reused that password-and possibly the systems themselves-are suddenly vulnerable. With the increase and evolving sophistication of phishing attacks, this is a serious problem. (Ives, Walsh & Schneider, 2004)

To rethink security design with an eye on usability, we need to understand how users think about security. We’ve already discussed the burdens users feel with regards to standard password policies, the need for ad-hoc solutions, and the lack of risk felt in most cases. One avenue to explore is how exactly users reuse their passwords. Notoatmodjo and Thomborson (2009) found that most users categorize their passwords based on context. Type of service was the most dominant type of categorization. For example, one password might be used for banking, another for online commerce, and third for social
networking sites. Additionally, users tended to use better passwords for sites they deemed of high importance, and tended to reuse these passwords less. Conversely, sites considered to be of lower importance used simpler passwords, and those passwords were reused significantly more often. The researchers go so far as to state “reusing passwords on the less important accounts should be encouraged rather than discouraged”, explaining that this reduces the burden of creating many strong passwords, and that instead emphasis should be placed on teaching users how to correctly identify sensitive accounts.

Therefore, the use of an IF system for low-risk sites makes logical sense. Users are already reusing passwords along these lines as much as possible. Switching to an IF system in a low-risk context reduces the password burden without exposing the user (or the systems) to additional risk. In fact, it can be argued that a centralized authentication might be beneficial. Should a user have reason to believe that their account has been compromised, a single password change will affect all associated systems. This will remove the problem of an infrequently used account being forgotten and remaining vulnerable. Identity Federation systems have the potential to maintain security while easing the burden on end-users.

One aspect of Identity Federation is Single Sign-On (SSO), which effectively allows users to leverage a single username and password combination across multiple sites. Currently, the vast majority of logins are web-based. In recent years, SaaS has made significant gains in usage, and cloud computing (often with web-interfaces for authentication) has skyrocketed in popularity. Given this, it makes sense to focus on how web-based SSO authentication will benefit or hinder users. Specifically, we need to
examine if the use of web SSOs will create any additional security risks, or complicate usability.

The rest of the paper will examine Single Sign-On, and will be organized as follows: first, there will be a discussion of general principles; this will be followed by close examination of the most popular component protocols. Next will be an in-depth look at the most widely used complete systems, followed by a discussion of their overall security and usability strengths and weakness. Finally, there will be a discussion about the future work needed in these areas.

**Single Sign-On Overview**

The ultimate purpose of web-based SSOs is to allow users to use a single account at multiple content websites. At least theoretically, this is a winning prospect for all parties. Users have fewer accounts to maintain, and need to enter their credentials less frequently. Identity providers can leverage this service to create a portal. Content providers are freed from the need to implement, maintain, and secure accounts locally, allowing them to focus on more relevant design and content related tasks.

**Elements**

In general, there are four elements that make up a single-sign on system:
**User:** an individual represented by digital identity, who uses unique credentials to authenticate him or herself.

**User-agent:** an application directed by a user to access resources. In the case of web-based SSO, this is almost always a browser.

**Resource Provider (RP):** an application that provides information and tools to a user. Sometimes known as a Service Provider (SP).

**Identity Provider (IdP):** A server that authenticates a user's digital identity based on unique credentials.

**Data Flow in SSO:**

There are two major SSO design structures, based on what entity initiated the authentication process. In most cases, a user visits a resource provider with the intent of accessing information or tools through his or her user-agent. In some cases, the RP wishes the user to identify his- or herself before accessing particular resources. The user is redirected from the RP to the IdP. At the IdP, the user enters his or her credentials, and is authenticated. The IdP then redirects the user back to the RP, along with a token or message indicating if the authentication succeeded or failed and possibly a set of assertions. Depending on the authentication status, then user is then granted or denied access to protected resources on the RP. This is an RP-initiated design, and is the most common implementation. Alternately, in an IdP-initiated design, the user visits the IdP first, authenticates, and then uses the IdP as a portal to access RPs.

**Security Concerns**
There are two major areas of security concern with SSOs, transmission and authentication. The intent of SSO design means that sensitive traffic will be traveling between domains. Without proper precautions, SSO traffic is vulnerable to transmission vulnerabilities such as replay, session hijacking, and man-in-the-middle attacks. These attacks are well known, and there are a number of ways to mitigate these vulnerabilities, such as endpoint authentication and use of SSL/TLS to encrypt the connection. However, real-world deployments often do not implement these measures properly, or completely fail to implement these precautions at all.

User authentication is another major type of security concern, not just in the case of SSOs, but also in any remote authentication using username/password combinations. Users are vulnerable to phishing and other forms of social engineering. As of yet, attempts to solve the problem of social engineering have yielded only partial solutions. In the case of SSOs, the problem of social engineering is compounded. If a malicious party captures a user’s credentials, the scope of information to which that malicious user now has access is considerably broader.

Finally, storage of sensitive information is a concern. In a number of cases, user authorization tokens are stored long term, and refreshed periodically. There are a number of types of shared secrets that must be stored correctly as well. Access to any of this information can allow an attacker to impersonate users and access their protected information without their consent.

Privacy Concerns
There is a distinct concern with SSOs (and federated identity in general) about how much information is being shared between the involved parties. By design, these systems need to share information, but the question we need to answer is how much information is required? Some level of transparency is needed for the user to understand what information is being requested by the RP, what access is being granted by the IdP, and what other information is being exchanged between the two parties.

Furthermore, there are concerns about how IdPs specifically may be collecting information. By using an IdP to log in to multiple RPs, the user is unwittingly giving the IdP access to her browser history. Combined with browsing history, IdPs are in a perfect position to aggregate data on users.

Usability Concerns

A primary motivator in stressing the use of SSOs in web-based authentication is easing the burden on users who have multiple accounts strewn across the Internet. However, without user understanding of the process, SSOs do little to alleviate the problem. It must be apparent to a user how to sign in to a third-party. Available IdP options and redirects between the RP and IdP should be clear and straightforward. RPs and IdPs should be clear on information being shared, and how much access to information each party will have.

Below we will examine these issues, first by detailing the protocols used to build SSOs, then by exploring full SSO systems. The aim is to discover if the protocols and systems can provide a secure, reasonably private, and usable SSO implementation. If so,
SSOs can prove to be a practical alternative to site-centric authentication, providing easier implementation for resource providers and improved overall usability for end-users.

SINGLE SIGN-ON COMPONENTS

OAuth

Overview

OAuth is an authentication protocol developed by the Internet Engineering Task Force (IETF). The working group was first charted by IETF in May 2009, and RFC 5849, which documented version 1.0 was posted in April of 2010. The most recent draft version of 2.0 (version 25) was submitted on March 8, 2012. The main goals of OAuth are to create a layer between IdPs and SPs, allowing users to authorize access to IdPs via SPs, and ensuring that access to a user's information is both limited and revocable. (OAuth Authorization Protocol, 2012) This is done using tokens. Both 1.0a and 2.0 are currently in use by a number of high traffic sites for Web Browser Single Sign-On. Version 1.0a sites include Twitter (“Moving from Basic Auth to OAuth”, 2011), Yahoo (Hoffman, 2008), and LinkedIn (http://developer.linkedin.com/documents/oauth-overview). Version 2.0 sites include Facebook (Shah, 2011), Microsoft LiveConnect (http://msdn.microsoft.com/en-us/library/live/hh243647.aspx), and Google (Authentication and Authorization, 2012)

The OAuth standard defines four roles; Resource Owner, Resource Server, Client, and Authorization Server. A Resource Owner is the entity that is capable of authorizing access to a requested resource. A Resource Owner is most commonly an end-user. A
Resource Server, therefore, is the server hosting the protected resource. The Resource Server can accept access tokens and grant access based on the parameters contained within tokens. The Client makes the request for access to protected resources, on behalf of the Resource Owner. This is analogous to the Service Provider in other Identity Federation systems. Finally, the Authorization Server is the server that creates an access token after authenticating the Resource Owner, similar to an Identity Provider in other contexts.

(OAuth Authorization Protocol, 2012)

The Resource Owner initiates the transaction, by requesting a service through the Client. The Client then redirects the user-agent to the specified Authorization Server. The Authorization Server then authenticates the resource owner in some manner, such as a username/password combination. If authentication is successful, the Authorization Server then redirects the user agent back to the Client, along with an authorization code. The Client uses the authorization code to request an authorization token from the Authentication Server. If the authorization code is valid, the Authorization Server returns the authorization token. The authorization token is then sent from the Client to the Resource server, where access to protected resources is granted based on the allowed scope indicated by the authorization token. (OAuth Authorization Protocol, 2012)

The token is an interesting element in this configuration. Embedded within the token are parameters. These include the period of time for which the token is valid and what protected resources the client has been authorized to access. This token may be refreshed at the request of the Client, if the resource owner has authorized refreshes. The Resource Owner specifies what resources the Client may access during the initial request, and a new token must be issued if the Client wishes to access additional resources. There
are two types of tokens defined in the OAuth specifications—access tokens and refresh tokens. Access tokens contain a string which represents the actual authorization issued to the client, and make include other authentication credentials. The structure, format, and methods of utilization are not defined, as those choices are left to the designers based on a particular server’s security requirements. Refresh tokens are sent by the client when the original access token expires, and include the authorization grant string. If the authorization server approves the request, a new access token is sent to the client. (OAuth Authorization Protocol, 2012)

Potential Attacks

There are a number of potential security issues with OAuth. While a great deal of work was put into ensuring the integrity of the various requests using signatures, there are several points at which the confidentiality of the keys, tokens, and other data are vulnerable. (Kiani, 2012) The vulnerabilities are organized below into those that occur at the Client, those that occur at the Authorization Server, those that occur at the Resource Server, and those that occur in transmission.

Client vulnerabilities

There are three major vectors for attackers to focus on the Client side of the flow. Client secrets may be compromised in several ways. Tokens are potentially vulnerable if not handled correctly. Finally, the Client itself may be compromise compromised, leading to expose of Resource Owner credentials as well as protected data.
One potential source of information for attackers might be Client secrets. If these are in any way hard-coded into the Client application, or if the Client application is otherwise compromised, an attacker could then have access to authentication server without Resource Owner approval. These secrets can also be obtained through guessing, unless there is sufficient entropy. If part of the secret is created dynamically, the attacker could potentially exhaust the pool of possible codes through rapid repeated requests. The most obvious solution to this threat is to avoid hard-coding any secrets in to a Client, or if this is unavoidable, make certain those secrets are well protected, deployment specific, and can be revoked should any compromised be detected. Secrets must contain a significant amount of entropy, and dynamically created codes should require some sort of verification of the requestor. Additionally, if the Client is a web server, the use of standard web server protection and monitoring will help to prevent compromised; if the Client is a stand-alone, developers must ensure that secrets are stored in a secured location. (Lodderstedt, McGloin, and Hunt 2012)

Access tokens can also be obtained illicitly if the attacker has access to the Client device. If the access token is stored in a location that is accessible by other applications, it can be easily copied and reused. To prevent this sort of attack, access tokens should be stored in private or transient memory space; access to the token should be limited to the Client itself; the token lifetime should have as short a lifetime as feasible; and the scope of the token should be limited to only the resources the client needs. (Lodderstedt, et al., 2012)

Refresh tokens are a particular problem, because they can be used to extend access to resources without direct intervention of the Resource Owner. Refresh tokens can be
obtained by compromising a web server, accessing the local file-system where a native application is installed, or stealing/cloning a host device which contains a refresh token. In all of these cases, standard security measures, such as strong web security and encrypted local storage can prevent refresh tokens from being exposed. If a compromise is detected, refresh tokens can also be revoked. Additionally, if refresh tokens themselves are configured thoughtfully, the potential damage of a refresh token hijack can be mitigated.

By validating the Client with each refresh, limiting the scope of what each refresh token authorizes, and by ensuring that tokens can be quickly revoked, attackers will be limited in their ability to exploit the tokens. If Refresh Token Rotation (replacing refresh token’s value automatically) is used by the Authorization server, this will also help prevent simultaneous use of the same refresh token. (Lodderstedt, et al., 2012)

Fraudulent or compromised clients are another point of vulnerability. A malicious client mimicking a legitimate client can obtain a Resource Owners credentials, or redirect the reply to another Client using open redirects. If that Resource Owner has been previously authenticated, another client can piggyback onto that authorization, causing the authentication server to redirect back to the client with the authorization code automatically. Pre-registered redirect URIs can prevent some of these incidents, as the authorization server will not redirect to just any client. User education regarding phishing and phishing prevention tools will also prevent these attacks from succeeding. (Lodderstedt, et al., 2012)

Clickjacking is a further method that can be used to obtain a Resource Owner’s credentials. In this attack, a malicious transparent iFrame site is overlaid on top of a legitimate site. Rather than clicking on the legitimate site’s authorization button, the
Resource Owner instead clicks a transparent button belonging to the malicious site, and unwittingly uses his or her credentials to authorize that malicious site. This can be prevented by proper client design, however. In a stand-alone application, an external browser should be opened when a Resource Owner’s credentials are requested. In web applications, there are two options. For those viewed through newer browsers, iFrames can be avoided by using the X-FRAME-OPTION header. Older browsers are somewhat more problematic, though. Javascript framekilling code can be used, but the effectiveness of this solution is dependent on the browser and local javascript implementation.

(Lodderstedt, et al., 2012)

**Authorization Server vulnerabilities**

Many of the vulnerabilities that affect Authorization Servers are part of the same vulnerabilities that affect Clients. If a Client is compromised in some way, it will be making unauthorized or unintended requests of the Authorization Server. The Authorization Server can potentially be deceived into issuing authorization codes because of hijacked tokens. However, there are a number of vulnerabilities that are aimed at the Authorization Server itself.

One such vulnerability is dependent on how tokens are stored. If the Authorization Server stores access tokens within a database as handles, that database may be compromised either by attacking the database access controls, or through a SQL injection attack. If the tokens are stored on the file system, an attacker can instead gain access to the file system itself. These threats can be mitigated by making use of standard system security protection measures. Additionally, if a database is used, storing the access tokens
as hashes and using SQL injection countermeasures will limit that vector for attack.

(Lodderstedt, et al., 2012)

Denial of Service attacks can also be used against the Authentication Server. In one scenario, if the authorization codes or access tokens include a nontrivial amount of entropy and are distributed without direct Resource Owner intervention, then it is possible to exhaust the pool of available codes or tokens. This threat can be mitigated by limiting the number of codes or tokens available to a single user. In another scenario, an attacker in control of a botnet becomes aware of the redirect URI of the authentication server and is able to direct an attack directly on the authentication server, denying legitimate users the ability to authenticate legitimate authorization codes. Use of SSL certifications and client validations of the certificates can help mitigate this particular threat, as can disallowing connections from Resource Owners or Clients which send too many invalid authorization codes. (Lodderstedt, et al., 2012)

Authorization Servers are also vulnerable to giving access to unauthorized users via Cross-Site Request Forgery (CSRF) attacks. This attack occurs after a Resource Owner has been authenticated. The attacker can substitute an authorization code or access token to the attacker’s private resource, rather than the Resource Owner's information. This allows the attacker to then be authenticated from a different client, and access the Resource Owner’s protected information. The best defense for this attack is utilizing the “state” request parameter that is part of the specification. This creates a value unique to the session (e.g., a hash value of the session cookie) that a Client can then use to verify the authorization code or access token as valid. (Lodderstedt, et al., 2012)
Resource Server vulnerabilities

Resource Servers also share common vulnerabilities with Clients and Authorization Servers. Illicitly obtained tokens may be used to gain access to protected resources, as can compromised authorization credentials. As with each of the other systems, Resource Servers have their own unique vulnerabilities as well.

Two of the biggest concerns related to the scope allowed by the access token. Generally, the concern is that a Client has access to more information than is intended by the Resource Owner. This is primarily a privacy and usability problem, and will be discussed further below. However, one potential technical mechanism to mitigate this vulnerability is to limit authorized scope based on the Client requesting access. If the Client has a pre-existing relationship with the Authentication Server and a trust relationship has been established, full access might be granted. A less trusted request would have the scope of its request limited, or access denied completely. (Lodderstedt, et al., 2012)

Transmission vulnerabilities

Transmission vulnerabilities are fairly straight-forward. Throughout the OAuth flow, whenever one system must communicate with another, there is a risk of eavesdropping and data capture, which includes Resource Owner credentials, authorization codes, access tokens and refresh tokens. The OAuth specifications specifically recommend the use of transport layer security, such as SSL or TLS. (OAuth Authorization Protocol, 2012) If, for some reason, a connection cannot be encrypted, it is strongly recommended that the scope and expiry of the access tokens be limited, to minimize the
access granted should any part of the communication be compromised. (Lodderstedt, et al., 2012)

Privacy

When examining this protocol, privacy and usability involve the same issue. The goal of OAuth is primarily to allow user information to be shared between unlinked services without divulging the authentication credentials of the end-user. Traditionally, for a third-party to gain access to end-user owned resources stored by another service, the end-user would need to release her credentials to the third party, and allow them to log in to the service on her behalf. This scenario is very problematic from a privacy perspective. First, the third-party now has access to the end-user’s credentials, and there is no way for an end-user to know how those credentials are stored, and if they are being re-used without the user’s knowledge. Second, by allowing a third party to access an account, the end-user is forced to give the third party full access, as though the third party is the end-user themselves. OAuth seeks to improve this scenario, by hiding the end-user’s credentials. An access token is issued instead, which may limit the time period and scope of access to the user’s information. If there is any concern, the access token may also be revoked. There is little controversy that OAuth improves end-user privacy in this scenario. (Shehab & Marouf, 2012)

However, this improvement is dependent in large part on the user understanding the scope of access they are granting to a third-party. End-users rarely have a thorough understanding of file permissions, and technical terminology regarding permissions will likely lead to users granting access they don't fully grasp. This gives a clear avenue for
malicious and/or unscrupulous Clients to take advantage of this lack of understanding. (Leiba, 2012)

There may be ways to mitigate this problem, however. It is imperative that Authentication Servers explain permissions in a plain manner, with information about the potential ramifications of granting this access. Added warnings for unusual access requests might prevent the more egregious information leaks. (Leiba, 2012) Other systems might prove useful as well. Shehab and Marouf (2012) have recently examined the feasibility and usability of a community-based recommender system. While the development is still the early stages, this appears to be a promising tool in assisting users in making reasonable choices when granting access to third-party Clients.

As mentioned above, there is a split in which version of OAuth is used by different providers. Google, Microsoft, and Facebook have opted for OAuth version 2.0, while Yahoo, LinkedIn and Twitter have chosen to only support OAuth 1.0a. Eran Hammer, community lead for OAuth, outlined three major changes between OAuth 1.0a and OAuth 2.0- the dropping of client side signatures, greater number of authorization flows, and better scaling performance. Hammer concedes that performance was a significant issue. Resource servers needed access to client credentials in OAuth 1.0a, and state management across different stages was difficult to maintain. The increase in authorization flows was also needed. After OAuth 1.0a began to be implemented in a variety of contexts, it became clear that the single flow in OAuth 1.0a was limited and difficult to use properly for both developers and end users. The biggest difference-the dropping of signatures-has produced significant discussion. One the one hand, signatures provide a way to encrypt tokens without relying on SSL or TLS for the entire exchange. Signatures also help prevent
exposure of secrets contained in a token, negating many possible attacks. However, cryptography is difficult. Requiring unknown clients to handle cryptography in a correct and secure manner often results in client side errors and unknown potential vulnerabilities. (Hammer, 2010) These are both valid arguments, and explain why there hasn’t been universal adoption of either version of OAuth.

**Security Assertion Markup Language (SAML)**

**Overview**

The SAML is a standard defined by the Organization for the Advancement of Structure Information Standards (OASIS). The goal of the standard is to create a framework for sharing security information between online business partners, with a focus on use in Identity Federation, Single Sign-On, and integration with other Web services and industry standards. SAML v1.0 was certified as a standard in November of 2002, and updated in September 2003 as v1.1. After input from organizations implementing and customizing SAML, v2.0 was standardized in March of 2005. (OASIS 2005)

There are several defined components in a SAML interaction. There is an Asserting Party, which is a system that makes SAML assertions. There is a Relying Party, which receives the SAML assertions. Additionally, if the one system makes a direct request of another, they are referred to as the SAML Requestor and Responder. There are other roles that exist in certain contexts. For example, in a Single Sign-On configuration, there are Identity Providers (IdP), who validate identities, and Service Providers, who make use of those identities. A Subject, then, is the entity being validated. (OASIS, 2008)
SAML is composed of four core building blocks that provide flexibility in the system architecture. At the heart of SAML are assertions. An assertion is a set of statements about a subject, including authentication validity and means, attributes belonging to the subject, and the scope of authorization. There are a number of protocols for sending these assertions, depending on the appropriate request or response required. Among these generalized protocols are the mechanisms to request an authentication or other assertion, force logouts from all active sessions, and change a value within an assertion. (OASIS, 2008)

The next sets of components are called bindings. These identify how SAML protocols are transported over various types of application protocols. The defined bindings for SAML v2.0 are HTTP Redirect, HTTP POST, HTTP Artifact, SAML SOAP, Reverse SAML SOAP, and SAML URI. Finally, all of these components-assertions, protocols, and bindings-are contained in specified combinations known as profiles. The profiles specify how each combination should interact, to provide better interoperability in particular scenarios. Some of the defined profiles include Web Single Sign-On, Enhanced Client and Proxy, Identity Provider Discover, Assertion Query, and Name Identifier Management. (OASIS, 2008)

There are also two additional concepts that are generally useful when creating SAML environments. The first of these is metadata, which includes configuration information needed by identity providers and service providers, such as identity attributes and information needed for encryption and signing. The second concept is the authentication context. This is needed when a service provider needs to know the type and strength of authentication used to verify a subject. (OASIS, 2008)
There are two potential service flows for browser-based single sign-on in SAML. This is dependent on who initiates the request. Most commonly, a user will visit a service provider. At the SP’s site, the user will choose to access a resource that requires authentication. The SP will send the user to an identity provider, who authenticates the user. The IdP then builds an assertion based on the user’s authentication, then refers the user back to the IdP, along with the assertion. The SP then processes the assertion, and uses that information to determine if the user is allowed to access the requested resource. (OASIS 2008)

In the second flow, the IdP is the initiator, rather than the user. In this case, the user has visited the IdP, and has already authenticated. The user then travels to a SP through a link on the IdP site. The IdP builds an assertion based on the user’s current authentication status, and the sends the assertion to the SP, along with the user’s browser. The SP processes the assertion, and then determines the access rights granted to the user at the SP. If warranted, local accounts at the IdP and SP can then be linked. This can be done at the time of the SSO or performed out-of-band, and can make use of either persistent or transient pseudonyms rather than a real name. (OASIS 2008)

Potential Attacks

The design of SAML is meant to be flexible, allowing for customization as needed. As such, many of SAML’s security issues will be discussed in combination with specific implementations, rather than in general. However, there have been a few analyses of SAML itself.
**Replay Attack**

One potential compromise is a tamper/replay attack. Groß and Pfitzmann (2006) closely examined the SAML v2.0 protocol, and found a potential replay attack, playing on how IdPs and RPs will invalidate authentication based on artifacts such as nonces. In this scenario, the attacker has gathered or made up several artifacts prior to the attack. He will then visit an RP and attempt to authenticate. This causes the RP to send an authorization request to an IdP, which the attacker captures before sending it on to the IdP. The IdP, upon receiving the request, will issue a valid artifact, then redirect to the RP. The attacker also captures the message containing the valid artifact. The message is then modified to remove the valid artifact, and instead a fake artifact is inserted. As is appropriate, since the artifacts do not match, the RP and IdP invalidate that request. The attacker is now in possession of a valid artifact, which can be used to impersonate the user.

Armando, Carbone, Compagna, Cueller, and Tobarra (2008) discovered a very specific attack against Google Apps using the SAML messages. When a user initiates a request to authenticate, a malicious RP can establish a second parallel session. Once the user is authenticated, the RP can send the authentication assertion back to Google for both sessions, and both sessions will be granted access. If the second session requested additional access, that access will also be granted. The researchers were able to replicate this discovered vulnerability in a lab setting, proving the existence of the vulnerability. It should be noted that once informed about this potential attack, Google immediately released an updated version of their SAML based SSO which did not include the above mentioned vulnerability.
OpenID

Overview

OpenID is meant to be a method of identifying an end-user for authentication purposes. It is decentralized by design, to give users freedom in choosing their OpenID provider, and to allow users to retain their identifier if they switch providers. It is also meant to work with minimal technology requirements, using only standard HTTP/HTTPS communication. Cookies are not used, nor is there any specific session management required. (OpenID Foundation, 2007)

There are a number of components to OpenID authentication as defined by the v2.0 standard. There is a User-Agent, which is a browser with HTTP capabilities that a user controls. There is an OpenID Provider (OP), which asserts the identity of the end-user. There is a Relaying Party (RP), which requests confirmation of a user’s identity. There is an OP Endpoint URL, which is an absolute HTTP or HTTPS URL that accepts the OpenID protocol messages. Finally, there is the Identifier, which is either an HTTP or HTTPS URI or an XRI, whose purpose is to identify the end-user. (OpenID Foundation, 2007)

The process begins when a user begins the authentication process by giving a RP his or her identifier. The RP first normalizes the identifier to ensure it’s in the proper format, which uses it to discover the OP endpoint URL using either the Yardis or XRI resolution protocol. (Recordon and Reed, 2006) If desired and available, the RP and OP can establish an association through a Diffie-Hellman shared secret exchange, which is then used to sign
future communications. The RP now redirects the user’s browser to the OP with an authentication request. The OP then authenticates the user in some manner, and then redirects the browser back to the RP with either an approval or failure message. Finally, the RP verifies the information from the OP, and then proceeds to grant local access based on the authentication status. (OpenID Foundation, 2007)

Potential Attacks

**Malicious Provider**

Once a user has requested to use an IdP, and that IdP has been discovered by the RP, the user is redirected to the IdP to authenticate. As part of the redirect, the IdP is server address is listed in the redirect message as the URL itself. A malicious RP, then, could instead send a user to an identical looking site to capture the user’s credentials. This phishing attack is probably the best known OpenID attack, and is problematic to solve, because it relies on the user recognizing that a site isn’t legitimate. (Tsyrklevich & Tsyrklevich, 2007)

In another attack, a malicious provider can use the authentication token in ways unforeseen to the user. In one possible circumstance, by using hidden iFrames, the malicious site can send unknown requests to other sites where the user is able to use the same OpenID URI. Tsyrklevich and Tsyrklevich (2007) give the example of a banking site, where one hidden iFrame is used to log a user in to a second website, and then a second hidden iFrame performs commands at that site-in this instance, transferring money from a bank account.
**URL Exploitation**

When a user first attempts to log in, they give a URL to the RP, which then downloads it and attempts to connect. If the user is malicious, that URL could point not to a set of OpenID credentials, but to a problematic URL. Examples of this include a random port on a legitimate site (making the RP appear to be port scanning), another port on the local system, a file to be downloaded, or a script or large file that causes the system to DoS itself. (Tsyrklevich & Tsyrklevich, 2007)

The use of HTTPS provides a false sense of security as well. When a user first requests to log in, he or she is then redirected to the IdP by the RP, usually over HTTPS. However, if any of the redirects are HTTP, not HTTPS, the user will receive the redirect over HTTPS, but the initial communication with the IdP will be over HTTP. Later, once the user has been authenticated and the assertion sent to the user, the user’s browser then looks to the “location” header from previous responses, which contains the HTTP endpoint of the RP, not the HTTPS location. This results in the assertion being sent to the RP through insecure channels. (Sovis, Kohlar, & Schwenk)

**Man-in-the-Middle**

There are two types of man-in-the-middle to which OpenID is vulnerable. The first of these involves the use of Diffie-Hellman as the encryption algorithm. While the use of a symmetric cipher is simpler and cheaper, they are vulnerable to interception between
parties and modification, potentially without either party discovering the modification. (Oh & Jin, 2008) Tsyrklevich and Tsyrklevich (2007) argue that simply requiring the use of HTTPS for all exchanges would render Diffie-Hellman unnecessary.

The use of HTTPS would also help prevent parameter manipulation attacks. As part of the initial communication between the RP and IdP, a message is sent from the RP to the IdP requesting an authentication assertion. Part of the message level security includes two fields, openid.sig (which is the auth code) and openid.signed, which contain the Message Authentication Code (MAC). The MAC is a hash value of all parameters xor-ed with the shared key. To verify an authentication message, a RP must check that the assertion is valid, and that all the required parameters have been protected. However, if a parameter is not considered required and not included in the MAC, that parameter can be appended to a protected message without invalidating the authentication assertion. This means that the RP may be given information it did not request, and which may or may not be valid. (Sovis, Kohlar, & Schwenk)

An extension of this injection is parameter forgery. In this scenario, the attacker takes advantage of the fact that requests from the RP to the IdP are relayed through the user, and only use HTTPS between the RP and the user. The request is sent from the user to the IdP over HTTP, without protection. This scenario, then, is ripe for a man-in-the-middle attack. When the attacker receives the request, the list of parameters requested by the RP is modified, removing the “required” tag from parameters. This means that the IdP will essentially return a void assertion, allowing the attacker to then inject any parameters s/he wishes. (Sovis, Kohlar, & Schwenk)
**Replay Attack**

Once a user has been authenticated, the IdP redirects them back to the RP with a signed assertion. If an attacker is sniffing between the IdP and the user, the attacker can capture and reuse the assertion. Generally, this attack is mitigated by the use of nonces, although if the attacker is fast, the user might be beaten to the RP by the attacker, negating the protection of the nonce. (Tsyrklevich & Tsyrklevich, 2007) Oh and Jin demonstrated a similar attack by using two browsers. In the first browser, a user is authenticated as normal. The assertion message from the IdP to the RP is captured, and the nonce recorded. In this message, the nonce exists in two places, the nonce parameter itself, and as part of the return_to URL. In a second browser, they submitted the same OpenID URI, captured the RP redirect message to the IdP, and again noted the nonce. The assertion received by the first browser was then modified, changing the nonce parameter but leaving the return_to URL, with the original nonce, intact. The researchers were able to successfully log in to the RP site in the second browser with the modified nonce, and access all of the restricted resources. (Oh & Jin, 2008)

**Phishing**

As mentioned above, there is a significant chance of phishing within the OpenID model. If you have a malicious or compromised RP, that RP can send the user to a site that appears identical to the actual IdP, but in fact will capture the user’s credentials. It’s been shown that even experienced users have difficulty identifying spoofed sites (Dhamija, Tygar, & Hearst, 2006), and it’s even less likely in the case of an expected redirect. (Bellamy-McIntyre, Luttroth, and Weber, 2011) The nature of Single-Sign On, combined
with the tendency towards password reuse, (Ives, Walsh & Schneider, 2004), makes this a particularly thorny issue.

There are a couple methods that may prevent phishing in OpenID. The first of these involves the use of a digital token. When a user first signs up for an OpenID URI, a passphrase or image is requested in addition to the password. This, along with the user name, is hashed and saved on the user’s computer. When the user is redirected to the IdP, the token is retrieved and decrypted. If the message or image is the same, the user continues to enter her user name and password. This token is portable, which allow the user to authenticate safely from public systems. The second method involves the use of a key. As in the previous case, when the user creates the OpenID URI, the IdP requests a passphrase or image. The IdP then stores that information in a hash, and gives the user a randomly generated authentication number that acts as the key. When the user is redirected to the IdP from an RP, she enters the authentication number. The passphrase or image is then displayed, and if correct, the user continues to sign in. (Lee, Jeun, Chun, & Song, 2008)

**Tracking**

By its very design, OpenID has a potential privacy flaw. Once a user is authenticated to one site, they can be automatically logged on to other sites from the same browser. This is convenient for the user, and part of the intent of OpenID. However, while authenticating a user to multiple sites, an unscrupulous IdP can track a user’s browsing behavior. Unless a user authenticates against multiple IdPs, there is no way for a user to avoid this tracking behavior. (Tsyrklevich & Tsyrklevich, 2007)
Potentially even more problematic, there is a way for a third-party to track a user's browsing habits through OpenID. As part of the HTTP header, there is a Referer (sic) tag that specifies the full URL of the site which was used to access the current site. This tag is considered very helpful in monitoring traffic patterns. During the processing of an OpenID authentication, though, the user's unique URI often appears as part of the Referer tag. If any of the site accessed during this period contain third party scripts (such as analytics or advertising), those third-party scripts will have access to the Referer [sic] tag, and the user's OpenID URI. In a random sampling of OpenID RP sites, researchers discovered that every site was vulnerable to this to this data leakage, meaning this is likely the most widespread privacy vulnerability in OpenID. (Uruena & Busquiel, 2010)

**SINGLE SIGN-ON CONFIGURATIONS**

**Shibboleth**

**Overview**

Shibboleth was first designed by Internet2’s Middleware Architecture Committee for Education, with the goal of creating an open-source federation protocol to allow research and higher education institutes to share information more easily. Shibboleth 1.0 was first released in June of 2003, and was based on SAML v1.1. The most recent version, 2.0, was released in 2008, and leverages the better security of SAML v2.0. Shibboleth has proved to be very popular, especially among higher education institutes, and federations based on the protocol have been created around the world. (Internet2, Project, 2012)
The biggest strength of Shibboleth is its federations. Rather than each organization attempting to ensure the policies and security of each organization it partners with, federations are set up. Each federation creates a framework of policies and expectations, and each member retains the right to determine who may access what resource. User accounts and attributes are managed by their home organization, but are leveraged to allow users access to resources at another member organization. (Internet2, High Level Introduction, 2012)

The protocol flow for Shibboleth is, unsurprisingly, much like that of SAML. The user first attempts to access a protected resource at the RP site. Shibboleth intercepts the authentication, and determines—possibly with user intervention—which IdP should authenticate the user. The user then authenticates against the IdP, using a SAML AuthRequest. The IdP responds to the RP with a set of relevant attributes, encoded in a SAML assertion. To protect the assertion, it is signed by the IdP, then encrypted with the RP’s key. As the message is sent to the RP, the user is redirected back to the RP. The RP decrypts the assertion and determines if the assertion is valid. The attributes are examined and translated to local properties if needed. The user is then granted access to resources based on those attributes. (Internet2, High Level Introduction, 2012)

Shibboleth is fairly flexible to implement, which may account in part for its popularity. The IdP software is written in Java, and therefore can run on any standard servlet. The authentication process can be handled by a number of common authentication systems, including Kerberos and LDAP. Attributes can be pulled from any database or directory, or included in the authentication process. The RP runs on the major web servers, Apache, IIS, and NSAPI, and can be run either as a stand-alone, or allow another application
to handle access control. These are all applications that most organizations are likely to have in place already, allowing them to leverage existing resources to use Shibboleth, rather than having to completely rebuild their systems to accommodate it. (Internet2, High Level Introduction, 2012)

**Potential Attacks**

Since Shibboleth is based on the SAML 2.0 standard, it naturally inherits any security flaws in the underlying protocol. There are also a number of unique potential vulnerabilities that may result in improper access to user information or other resources. It should be noted that these are only generic attacks-the nature of specific implementations affects the potential vulnerabilities significantly, and is far too broad a topic for a single study.

*Impersonation*

In this attack, a malicious entity representing itself as a trusted service provider sends a request to an IdP. Instead of responding back to the trusted SP, the request directs the IdP to respond to a third site, which is under the attacker's control. In the case of a POST SAML profile, the attacker can gain a valid assertion, and use that to access the user's private information as specified by the assertion. In the case of an Artifact profile, the attacker could use the response to immediately gain access to the secured resources of the RP. To prevent this attack, the IdP should have a secure association with the RP. (Cantor, et al., 2005)
**Redirect**

A related attack involves the TARGET parameter in the authentication request. This parameter is generally a URL, but the IdP does not actually confirm the parameter as valid. Allowing this to remain in plain text can expose information about the user’s browsing habits, and assists malicious entities in crafting redirect attacks. The obvious solution is to encrypt or obfuscate the TARGET parameter. (Cantor, et al., 2005)

**Lack of Anonymity**

To protect user privacy, Shibboleth has an explicit mechanism for anonymity. To achieve this, the user identifier is substituted with an anonymous identifier. However, if the anonymous identifier is used again for the same user across multiple sites, or repeatedly at the same site, information about that user can be aggregated to create a profile, which can then be used to identify the user’s real world identity. To mitigate this, it’s recommended that anonymous identifiers be one-time transient identifiers, rather than static. (Cantor, et al., 2005)

**Amazon Identity and Access Management**

**Overview**

Recently, Amazon Web Services (AWS) has expanded its Identity and Access Management tools to include Identity Federation. Currently, AWS clients use a single billed
account for access to AWS resources. This meant that if multiple users from the same organization needed to access a particular service, the account information must be shared, and that user would have access to all available resources. In this case, identity federation allows AWS clients to leverage local authentication systems, and to give access permissions to AWS resources based on local credentials. Additionally, access is given on a time-limited basis, which allows organizations to better limit access as well as control billing for resources used. (Barr, 2011)

The AWS Federated Identity process begins when a user at a participating organization attempts to access an AWS resource. This will call the Identity Broker, which requests the user's credentials. The credentials are validated against the organizations LDAP server. If the user is validated, a token request is sent from the Identity Broker to the AWS Security Token service. This request includes a length of duration (up to 36 hours), the associated AWS account policy, and specific permissions to be granted to that user. If the Security Token service accepts the account policy allows for federation for this user, a token is created and returned to the Identity Broker, along with an access key, a secret key, and the duration of the token (which may be different than the requested duration, depending on the account policy). The Identity Broker then passes this set of temporary security credentials back to the original AWS service user-agent. The application uses the temporary credentials to request the AWS service. The AWS service verifies the credentials with the Amazon Identity and Access Management service, and if valid, then grants access based on the permissions and duration indicated by the token. (Barr, 2011)
Potential Attacks

While there have been no formal examinations of Amazon IAM Federation, we can infer potential problems from examinations of other similar systems.

The key point of vulnerability in this system is the configuration of the URL used to contact and sign-in to the Amazon resources. After the temporary security credentials (access key, secret access key, and token) are created by the AWS Security Token service, they are sent back to the Identity Broker. The Identity Broker then formats them into a single JSON string along with the URL of the issuer and the URL of the AWS federation endpoint. This creates a request for a sign-in token. This URL is then sent to application, which forwards it to AWS federation endpoint. The federation endpoint validates the temporary credentials, and if correct, returns a sign-in token to the application. This sign-in token is added to the JSON string, creating a URL that will log in the user. This final string is valid for fifteen minutes after the sign-in token is created. Amazon strongly recommends that this URL be treated as a secret, and suggests that the 302 HTTP response be used over SSL to ensure the integrity of the URL. (AWS Documentation, 2012)

If not properly protected, there are a host of possible attacks that a malicious entity could use if it is in possession of the secret constructed URL. If the attacker has in their possession the request URL from the application, the issuer URL could be altered to redirect the approved sign-in token to a different site, thus giving the attacker permission to access the protected resources. Likewise, an attacker could either intercept the final string, or copy it and use it again during the 15 minute window of validity. In both of these
cases, the attacker is able to impersonate the intended user and gain access to restricted resources

**Social Network Providers**

**Overview**

Currently, the most prominent examples in public web space of the single sign-on concept are those provided by social media sites. Many of these sites have begun offering APIs to web developers that allow those developers to give users the option of using their social network site credentials to log in, rather than creating a new site-specific account. For the most part, these providers are leveraging currently available SSO technologies. Among the many sites offering this service, several of the biggest names include Facebook, Google, Twitter, and Yahoo.

Facebook announced their Connect service in November of 2008. Connect made use of proprietary APIs to authenticate users and share information with partners. (Stone, 2008) In May of 2011, Facebook announced that it would be dropping the Connect system in favor of a new platform based on OAuth 2.0. In addition to following the OAuth standard, two additions were included in the implementation. First, developers were required, rather than encouraged, to use HTTPS for all communication between Facebook and the third party site. Second, they require the use of a base64url encoded JSON object to wrap around the OAuth token. The idea behind this “signed_request” parameter is to ensure that the RP that receives the authorization token is the same RP that requested the token. (Shah, May 2011)
As of August 31, 2010, Twitter began exclusive support for single sign on requests using OAuth 1.0a. Prior to this, Twitter allowed connections through BasicAuth, and simple authentication scheme that required the application to store the user's login and passwords. This was a very obvious security hole, and thus Twitter made the decision to move to OAuth. (Moving from Basic Auth to OAuth, 2011) Unlike Facebook, Twitter does not extend its API beyond the OAuth specification, and as of this writing, there has been no indication that Twitter will move to using the more recent OAuth 2.0. (Implementing Sign in with Twitter, 2012)

Yahoo!, by contrast, supports several methods of authentication and authorization. The first attempt at providing a single sign on service came in the form of Browser-based Authentication, or BBAuth, in 2006. (Arrington, 2006) BBAuth is a proprietary system not unlike OAuth, where registered applications may redirect users to Yahoo! For authentication, and if successful, returns a user token to the application. (Browser Based Authentication, 2012) In 2008, Yahoo added support for OpenID 2.0, automatically enabling creating OpenID URIs for all users. (Arrington, 2008) Late the same year, the company included support for OAuth 1.0 as part of its Open Strategy. (Hoffman, 2008) The API was updated to use OAuth 1.0a in April of 2009, but like Twitter, there are no plans to move to OAuth 2.0. (OAuth Security Issue FAQ, 2012) Finally, in September of 2009, Yahoo! announced an OpenID-OAuth hybrid protocol. This hybrid combines an OpenID authentication request with an OAuth authorization request, and is Yahoo’s preferred authentication service. (Tom, 2009)

Like Yahoo!, Google supports several methods of authentication as well. In June of 2008, they introduced support for OAuth 1.0 for all Google Data APIs, including Gmail,
Documents, Calendar, and YouTube. (Kirkpatrick, 2008) In October of the same year, OpenID support was announced. (Lardinois, 2008) A hybrid protocol combining OpenID and OAuth was released the following January, in 2009. (Schonfeld, 2009) Finally, in March, 2011, Google began support of OAuth 2.0. Currently, Google recommend the use of OAuth2.0, OpenID 2.0, and the hybrid version of OAuth and OpenID. (Authentication, 2012) As of April 20th, 2012, OAuth 1.0 was depreciated, as were two proprietary protocols, AuthSub and ClientLogin. (Authentication, 2012) Additionally, in 2008, Google began supporting SAML 2.0, to allow enterprise organizations to leverage their own accounts to access Google applications. (Armando, et. Al, 2008)

**Potential Attacks**

As with Shibboleth, all of the social network providers are building upon existing protocols. Thus, they inherit the security flaws of those protocols. As mentioned above, in the descriptions of OpenID and OAuth, there are a number of potential vulnerabilities. OpenID is vulnerable to various sorts of URL exploitations, phishing, man-in-the-middle and replay attacks. OAuth suffers from problems related to exposure and manipulation of the token by unintended parties, forged authorization requests, and broad scope of access to unintended data.

These problems are exacerbated by the fact that these are freely available APIs, meaning that any site can add SNP single sign-on without the SNP consent or review. If any part of the accepted best practices is not implemented properly, accounts and data may become vulnerable to unauthorized leakage. For example, just recently facial recognition startup Face.com created a vulnerability for users logging in with their Facebook or Twitter
accounts. Face.com had not properly secured storage of their OAuth tokens, which meant it was possible for others to gain access to tokens other than their own, log in to that account and gain access to that user’s data. (Constantin, 2012) TweetGif opened a similar hole for Twitter users. Again, tokens were improperly stored, and the hacker group LulzSec Reborn captured and posted tokens for approximately ten thousand users. (Yin, 2012)

**Discussion**

**Security**

Having examined these systems and the protocols they are built upon, the question is now, are these an improvement over the traditional single-site authentication scheme? Or are the risks involved with single-sign on systems too great? Do these systems provide greater usability for users? How will single-sign on systems interact with the rise of cloud storage and Software-as-a-Service?

In the protocol examination above, many potential attacks were identified. Tokens and assertions can be captured, either in transit or due to insecure storage. This is particularly problematic with tokens that can be refreshed, as they tend to have a long life and long time storage. The tokens can then reused or altered to impersonate a user or access their resources. Secrets between the client, IdP, and/or RP can be stored insecurely or poorly generated, leading to theft or guessing of the secret. Various messages between entities can be captured and altered, tricking the IdP into issuing valid tokens or assertions to someone other than the intended RP. URIs can be altered to produce a wide range of
attacks, particularly in the case of OpenID systems. Finally, attackers can take advantage of mismatched expectations between client, IdP and RP requests.

Many of these attacks can be mitigated by good security practices, however. Storage, both on disk and in memory, of any sort of sensitive item can be hashed and otherwise encrypted. Access to the sensitive information should be carefully designed and extremely limited to only the entities that need access. User inputs can be cleaned to prevent SQL injections or URL exploitations. Secure channels between the various entities can be required before any unique information is transmitted. Once the secure channels are established, all communications between the entities should be through that secure channel. While SSL/TLS have been shown to vulnerable at times, the use of proper key length provides a considerable amount of security at a relatively inexpensive cost (Heninger, 2012), and thus should be considered a standard part of secure communication in these situations. Other measures, such as nonces, can be used if they are allowed by the protocol. All of these are standard practices that would go a long way to diminishing the types of available exploits, and give the various parties a reasonable confidence in the security of these protocols.

Protocols as written, however, are not the same as systems once they are implemented. While the research on protocol is somewhat limited, it is always done using models rather than real world implementations-with one exception. In May of 2012, Wang, Chen, and Wang published their findings on eight real world implementations of web-based single sign-on systems. Due to the limited nature of documentation for the APIs provided by the IdPs, the researchers instead analyzed the traffic to and from the client browser, much like an attacker would to discover points of attack. In all eight cases, the
researchers were able to successfully find a way to impersonate a user. In every case, the problem was a logic error in the implementation, rather than a technical flaw in the protocols used. (Wang, et al., 2012)

In one case involving Google’s implementation of OpenID, the researchers were able to exploit a difference in expectations between the IdP and the RP. The initial request did not require the email address of the user to be included as a field in the request, and as a result, in subsequent requests the email address field was blank. When the authorization request to the RP was relayed through the browser, the researchers were able to capture it and enter the email address of another user, and gain access to that user's account. In another exploit, the researchers were able to exploit the Facebook Connect service through the RP’s use of Adobe Flash. Flash was used to try and leverage the same-origin request policy that is part of the software, thus ensuring that a response to the IdP could only come from the RP. However, the researchers discovered a flaw in how Flash handles unpredictable domain names, and were able to exploit that flaw, so that if a user visits a malicious RP running Flash while logged in to Facebook, the token that RP was given could then be used to log in to a legitimate RP running Flash. (Wang, et al., 2012)

A third exploit involves a popular software package by JanRain, which bundles the major SNP APIs into a single service. RPs are registered with JanRain, and all SSO logins are routed through JanRain, which only accepts communication from registered domains. To exploit this system, the attacker must have their own registered domain to get past this white list scenario. With that tool, though, the attacker is able to bypass the white list and send a request containing two URLs, one for the attacker’s registered domain, and one for the RP he wishes to impersonate. This then gives the attacker the credentials to
impersonate the legitimate RP. If a user then visits the impersonated RP, the attacker is able to capture their authorization token, and use it on the legitimate RP to gain access to the user’s resources. Finally, in a fourth exploit, the researchers were able to take advantage of a system that links local account on a RP with those at an IdP. In this case, the researchers were able to link an attacker’s Facebook account with another user’s local account on the RP during this linking process. This allowed the attacker access to all of the user’s information on the RP. It should be noted that all of these exploits were reported and fixed before the research was published. (Wang, et al., 2012)

What this diversity of exploits suggests is that there is a wide variety of implementations of the standard APIs. Security of the authentication process is reliant on not only the security of the underlying protocols and APIs, but on the logic of the implementation and the flaws of the platforms themselves. This means that the onus for good security is on the RP developers. These developers must perform their own security checks on their sites, not merely implement a few API calls. This responsibility begs the question of how prepared web developers are to perform these types of checks, given the complexity of some of these exploits. Some professionals are clearly aware of the potential for exploits, as the use of Flash and JanRain’s whitelisting show. Better understanding of the nature of exploits is needed, and with this information, better testing tools and implementation best practices can be created.

Of course, single sign-on systems are vulnerable to many of the same problems as single site password systems. Malicious providers can still potentially gather credentials through phishing and pharming tactics. Username and password combinations can still be compromised through other forms of social engineering. Client systems can still be
compromised, exposing credentials as well as other sensitive information. Single sign-on does not solve these problems.

What we gain, though, is potentially a more robust authentication scheme. Currently, we have a situation where users are required to register at single sites so frequently, many accounts are abandoned. We also have a situation where the combination of poorly guarded single sites and heavily reused passwords allow otherwise well-secured high-value sites to be compromised. By reducing the number of accounts a user must maintain, we allow for the possibility of more stringent authentication measures. Additionally, much like in a strong castle design, reducing the number of points of entry reduces the potential vectors of attack, and allows defenders to focus their resources. Additionally, at the risk of privacy concerns, we gain the ability to model and monitor user behavior to detect unusual activities that might indicate a compromised account. Should an account be compromised, there are then fewer points of revocation. (Madsen, Koga & Takahashi, 2005)

In general, the protocols associated with single sign-on systems are mostly secure, and the vast majority of problems can be mitigated by good security practices. Updates are needed to expand and solidify best practices, and provide developers with a clearer understanding of secure implementation. There is still quite a bit of work to do, however. Wang, et al., proved that there is a considerable distance between studying protocol models for exploits, and searching for exploits in real world implementations. While it is a more difficult task, security researchers need to recognize the importance of this distinction, and place more emphasis on real-world scenarios.
Usability

As has already been discussed, ultimately for a system to be secure, it must be usable by those for whom it was designed. Without a reasonable level of usability, users will either choose to avoid the system, or will use it improperly and open the system up to vulnerabilities. Security is an abstract concept for most users with little real feedback, either positive or negative. As such, they will underestimate their risk and put a low value on their own personal security. In general, all of the mechanisms surrounding security-logins, encryption, permissions, and so on-are secondary to their primary goal of accomplishing a task. (West, 2008) With the low value users place on security, any security task that requires more than a small amount of effort is too onerous for users, and will generate frustration. It is with this focus we must examine single sign-on systems.

There are two primary areas of focus when looking at the usability of SSOs-the ability for a user to successfully log in using an SSO system, and the ability for users to correctly identify and set limited authorizations for third-party access to information. As of this writing, there have only been two formal usability studies done on SSOs. Both focused on OpenID, due to the more standardized implementation, but the results of these studies provide a wealth of information related to all SSO usability concerns.

The first of these studies was conducted by Yahoo! researchers in 2008 to check the usability of the Yahoo! OpenID login implementation. Subjects were asked to write a review on a third party site using their Yahoo! account, then to log back in to the site to update the review. The initial login process was very problematic. The users were unfamiliar with the
OpenID name and concept, and were confused when they only saw the option to log in with OpenID. The use of a URL to authenticate was confusing as well, as they expected to enter their username and password combination. The users also felt the initial authorization took too many steps—first, finding the correct method of login on the third party site, a second page where the users entered their credentials, a third page authorizing the use of Yahoo! to authenticate at third party sites, and a fourth page authorizing the specific third party site as well as containing considerable information about OpenID. This fourth page was especially problematic, as there was a great deal of information in which the users had little interest, the actual authorization was somewhat hidden by all of the information, and if users did not click specifically on the authorization (“Let Me In”) button, they were never redirected back to the third-party site. Later attempts to log in at the third party site were also confusing to the subjects, many of whom attempted to use their Yahoo! username/password combination at the third party site, rather than the OpenID login, and the error messages provided little indication that this was the problem. (Freeman, 2008)

These frustrations suggest that this authentication process needs to be streamlined to be a quicker task, which relates back to the concept that security is a secondary concern for users. Freeman suggested a number of recommendations that would improve the authentication process. First, keep the number of steps to a minimum, or it will be more work than creating another account. Focus on utility and well known branding, not the technology. Users expect to use username/password combinations, so login choices should be clearly specified. Provide information on technology concisely early in the process, and open more detailed information in another window to prevent derailment. Make it clear that they are logging in to the third party site. Once authenticated, return the user to the
task they initially tried to access. Finally, when a user returns to the third party site and logs in incorrectly, there should be information suggesting they might be using an SSO account. (Freeman, 2008)

Over the course of the study, it became clear that the users had two different mental models of the process. The first model suggested that the SSO process was analogous to a master key—it could open all doors, but only one at a time. Users who understood the process as this model believed they were only logging in to the third party site, not to Yahoo! itself, and likewise would only log out of the third party site. The second model was likened to a keyless remote to a car that could open all the doors at the same time. These users thought that the log in process would log them in to all Yahoo! related accounts, and subsequently, if they choose to log out of a third party site, they would be logged out of all Yahoo! accounts. In the case of Yahoo! OpenID, both models were correct at different times. During the login process, users did authenticate to both Yahoo! and the third party site. However, when a user logged out at the third party site, they were only no longer authenticated to that site, but remained logged in to Yahoo!. This inconsistency was very confusing to the users, and potentially meant they would be logged in to Yahoo! when they did not intend to be logged in. (Freeman, 2008)

Many of these problems and concerns are reiterated in Sun, et al.’s 2011 study. In this study, participants were asked to sign up for and sign in to several OpenID enabled resource providers, including a phishing demonstration site, then log out as though they were walking away from a public computer. After completing the task, the users were then asked to complete a questionnaire regarding the process, followed by a contextual interview. As with Freeman’s study, the initial log in process was very confusing. Many
users believed that the login form was already integrated with an Identity Provider, and attempted to log in using their Google or Yahoo username and password. They did not realize they needed to select the Identity Provider, assuming the icons were instead advertisements. They also had difficulty with the mental model, not understanding they were authenticating to both the third party site and the Identity Provider-again, must like Freeman’s study. Once of the sites used account linking rather than direct authentication and most users found this confusing and purposeless. The majority of the participants successfully identified the phishing website used in this study, but all expressed concerns about identifying a phishing site in the wild. The users were also concerned with the privacy of their accounts. They were unclear on how much information they were sharing and whether it would be used inappropriately. (Sun, et al., 2011)

From these findings, Sun, et al. created a list of requirements for an OpenID enabled site to be usable. First, the login form must be clear that users are able to choose and sign in to their existing accounts through the Identity Providers, and must be consistent with other types of login experiences. These sites must not rely on users’ ability to detect illegitimate sites. There must also be a clear and single method to log out of all accounts. Finally, Sun, et al. recommended that users need a central location with fine grained controls to manage their privacy settings. (Sun, et al., 2011)

As of this writing, there has been no research on how easily users are able to determine access permission given to resource providers. There has, however, been some work done in determining how users are able to set permissions using the Facebook interface. Given that Facebook is a major identity provider, this research should give us some insight into the nature of user set access controls. What researchers have discovered
is that the available controls are quite often not used to their full advantage, and when they are, many users find them ambiguous and inadequate. The vast majority of users either do not set any access controls, or use them only to limit access to their friends. When directed to perform certain tasks involving access control, the tendency was to overshare rather than undershare. When attempting to use the tools to set particular access permissions, users generally complained that the tools were either too complex to understand, or not complex enough to complete the task properly. Most were unsure if they’d correctly set permissions, and less than half of them actually set the permissions properly. (Egelman, Oats, and Krishnamurthi, 2011) These findings suggest that users are correct to be concerned about privacy. They also suggest that more attention should be paid to designing usable interfaces and providing clearer instructions and feedback.

While this discussion has focused on end users, they are not the sole usability concern in the single sign-on realm. Web developers who are called upon to implement these technologies must also have a clear understanding of best practices and secure design. As mentioned above when discussing OAuth, there has been disagreement among the designers of OAuth over the inclusion of signatures in OAuth 2.0. The use of signatures in OAuth 1.0a made the protocol technically more secure, but in practice the complexity of the cryptographic requirements meant that many implementations failed to work as intended. Ultimately it was decided that the usability cost of signatures was too great, and would hinder the ability of web developers to properly and securely implement OAuth, and thus curb the overall adoption of the protocol. (Hammer, 2010) If protocols are not implemented correctly and according to best practices, it ultimately does not matter if the protocols are technically secure. As such, the usability of the standards must be considered.
Despite all of these difficulties and concerns, users are quite positive about the idea of single sign-on systems. (Freeman, 2008; Sun et al., 2011; Goings and Abel, 2011) There has been an increase in users who would like to use a social network provider to log in to third party sites, and a corresponding increase in those who would prefer to log in with a SNP. (Goings and Abel, 2011) If users are interested in a feature, it’s likely to be implemented by third party sites. To help ensure proper use and security, these implementations need to be unambiguous about user decisions. The above research suggests that more work needs to be done to create best practices for usability in single sign-on implementation.

Benefits

While there are still security concerns and usability issues, there are a number of benefits. As mentioned above, users are increasingly interested in using single-sign on technologies, and have been known to leave sites rather than register (Going and Able, 2011) Resource providers will benefit not only from happier customers and increased page views, but from decreased costs. Resource providers will be able to reduce the costs of password management and recovery, as well as the cost of securing those resources. Identity providers will benefit from increased user data. (Madsen, et al., 2005)

There are positives from a security perspective as well. The use of single sign-ons may reduce phishing success. Login frequency will decrease through the use of refresh tokens, and will become more unusual events for users, potentially heightening their
awareness. When users do need to enter their login credentials, they will become accustomed to only doing so through identity provider interfaces. Authentication resources will be concentrated at only a few providers, allowing for the use of stronger authentication methods. Since there are fewer points of authentication breaches can be discovered more quickly, and user account privileges revoked immediately, as needed. (Madsen, et al., 2005)

Finally, with the increased popularity of cloud based technologies and the rise of Software as a Service, single site identity management solutions are becoming unmanageable for individual users and larger organizations. These various services generally require some sort of interaction, either between the software service, or by consolidating subscriptions. Federated identities mean that users can access a variety of cloud-based services from different providers with a single log-in, and that access between services can be integrated based on the user’s needs. Without some form of federated identity, cloud based services will be significantly less valuable.

Future Needs

At this point, web based single sign-on systems are well on their way to being a popular and useful service, but the technologies involved are still maturing. There are a number of areas where work must still be done. Documentation and best practices are in desperate need of clarification. Best practices need to be better understood and more readily available to web developers. APIs, particularly those made available by social
network providers, need a great deal more information to avoid the sort of logic error that Wang, et al. discovered. Security researchers need to take on the task of examining real world implementations rather than relying on modeling. Better testing tools will help both researchers and developers. Protocols should continue to be updated and modified as better processes are discovered, with a goal of improving security while simplifying implementation. As has been mentioned previously, proper cryptography is difficult, even for security professionals. It may be naïve to assume that web developers will be able to properly implement such security measures, and this consideration should be part of the overall design of the protocols.

One area which has not been addressed thoroughly in this discussion is the concept of trust. In some scenarios, such as Shibboleth, trust is built into the design of the system. In others, such as OpenID and OAuth, there are no trust requirements. Some identity providers, such as Google, Yahoo, and JanRain, have attempted to make up for this problem by only accepting requests from registered partners. However, this is problematic, especially for sites with open APIs, because registration does not scale very well. Furthermore, end users are unaware of which sites are considered trustworthy resource providers, beyond general reputation. The Kantara Initiative (formerly the Liberty Alliance) is currently working on creating an assurance system, with standard levels of assurance that are certified by the Kantara Initiative. There is a high level of buy in from a wide range of organizations, which helps lend weight to the certifications. (Kantara Initiative, 2012) A widely regarded certification program to establish that partners meet certain levels of security and process management will go a long way towards creating a
trust network between providers, while informing users of the provider’s trustworthiness. The spread of this program, and others like it, will do much to solving the problem of trust.

More transparent application permissions will help as well. As it currently stands, most users are confused by the access they are granting to third party systems. Better explanations and better design will help this problem. Centralized management of permissions granted will also help. Without these key steps, user privacy is likely to be violated, quite likely without the user having any inkling of a problem.

Finally, one of the benefits of fewer identity providers is that stronger authentication methods can be used. A number of identity providers, including Google (Shah, February 2011), Facebook (Song, 2011) and Blizzard (Danchev, 2008) have begun experimenting with two-factor authentication, primarily using SMS, mobile applications, or keychain code generators. Researchers have been experimenting with using local key storage, either built in to the browser (Sun, et al., 2011), or as a separate applications (Perlman and Kaufman, 2008) Additionally, the simple solution of enforcing strong password restrictions-possible due to fewer actual authentication-will help increase the authentication security supplied by identity providers.

**Conclusion**

Single sign-on systems are becoming more popular. While the designs are still somewhat immature, the provide benefits to all parties involved. Identity providers are able to protect themselves from vulnerabilities due to password use while capitalizing on
increased user data. Resource providers are able to reduce their support costs while incorporating a service users’ prefer. Users are relieved of the burden of an ever increasing number of accounts. As more services move to the cloud, SSO systems will become more valuable to providers and users. There is still significant work to be done to help ensure the robustness SSO security and usability, but the reward is worthwhile.
Work Cited


http://www.computerworld.com/s/article/9228322/Researcher_Face.com_iOS_flaw_could_haveAllowed_Facebook_Twitter_account_hijacking


Heninger, N. (2012, February) “New Research: There’s No Need to Panic Over Factorable Keys—Just Mind Your P’s and Q’s” Freedom To Tinker: Research and Expert Commentary on


