An Overview of Modern Web Authentication Methods

Daniel Guido

ISIS Lab, Polytechnic University

October 28, 2008

Table of Contents

Authentication on the Web

When someone begins talking about web authentication, the first thoughts that usually come to mind are HTTP Basic Authentication, HTML Forms, and passwords. These types of technologies perform authentication of the user to the web server and are used to provide a mechanism for the web server to distinguish between different users. These basic forms of user to web server authentication are both impractical and harmful in today's threat environment where targeted phishing, cross-site-scripting, and cross-site-request-forgery attacks are the norm. Several researchers have also noted that social problems exist with the use of HTML Forms to accept passwords. In their paper ""<input type="password"> must die!", Daniel Sandler and Dan S. Wallach make the observation that, "It is remarkably easy to accidentally give a valid password to the wrong website, whether the user is being attacked (i.e., phishing) or has simply forgotten which of a handful of passwords is the right one for the legitimate page."[1]

In this paper, we identify several improved authentication methods, which compromise the majority a web server administrator is likely to encounter. For each, we describe the benefits and costs associated with its implementation.

Phishing

In the prior section, authentication has meant authenticating the user to the web server. This type of one-way authentication has been sufficient in the past given that the targets of most attacks were the web servers themselves. Recently, new types of attacks have gained widespread adoption that target the user instead. These types of attacks, called *phishing*, exploit the fact that there is extremely weak authentication from the web server to the user; the opposite direction from the most common authentication schemes.

In a phishing attack, a *phisher* sets up a web site with content stolen from the target organization. This landing page is then communicated to the target users in such a way as to trick them into visiting it. Users then enter their authentication credentials to this fake *phishing website* believing that the phishing website and the legitimate website are one and the same.

I HAVE A NEW HOBBY. IT'S CALLED PHISHING.
scottadams@aol.com
www.dilbert.com
I SEND FAKE BANKING E-MAILS TO GULLIBLE EXECUTIVES. THEN I FIND OUT THEIR FINANCIAL INFORMATION AND USE IT TO STEAL THE MONEY THEY DON'T DESERVE.
8-12-05 © 2005 Scott Adams, Inc./Dist. by UFS, Inc.
Dear Customer,
This is your bank. We forgot your social security number and password. Why don't you send them to us so we can protect your money.
Sincerely,
I. B. Banker
LOOKS LEGIT.
© Scott Adams, Inc./Dist. by UFS, Inc.

[1] Daniel Sandler and Dan S. Wallach. ""<input type="password"> must die!", http://seclab.cs.rice.edu/w2sp/2008/papers/s1p2.pdf

Mutual Authentication

To mitigate the threat oh phishing, most new authentication schemes on the Web involve some form of mutual, two-way authentication in which the user and the web server are authenticated to each other, instead of simply in one-direction. An effective method of performing mutual authentication on the web would easily identify to the user whether the website they are at is the same website they *believe* they are at and thereby prevent phishing attacks.

SiteKey

Technical Overview

One such system that attempts to authenticate the server is SiteKey, a mutual authentication scheme driven by a 3-step login process where a secret image and/or secret phrase the user selects are stored on the server. SiteKey was one of the earliest such mutual authentication schemes and was famously adopted by the Bank of America for all of their users[2].

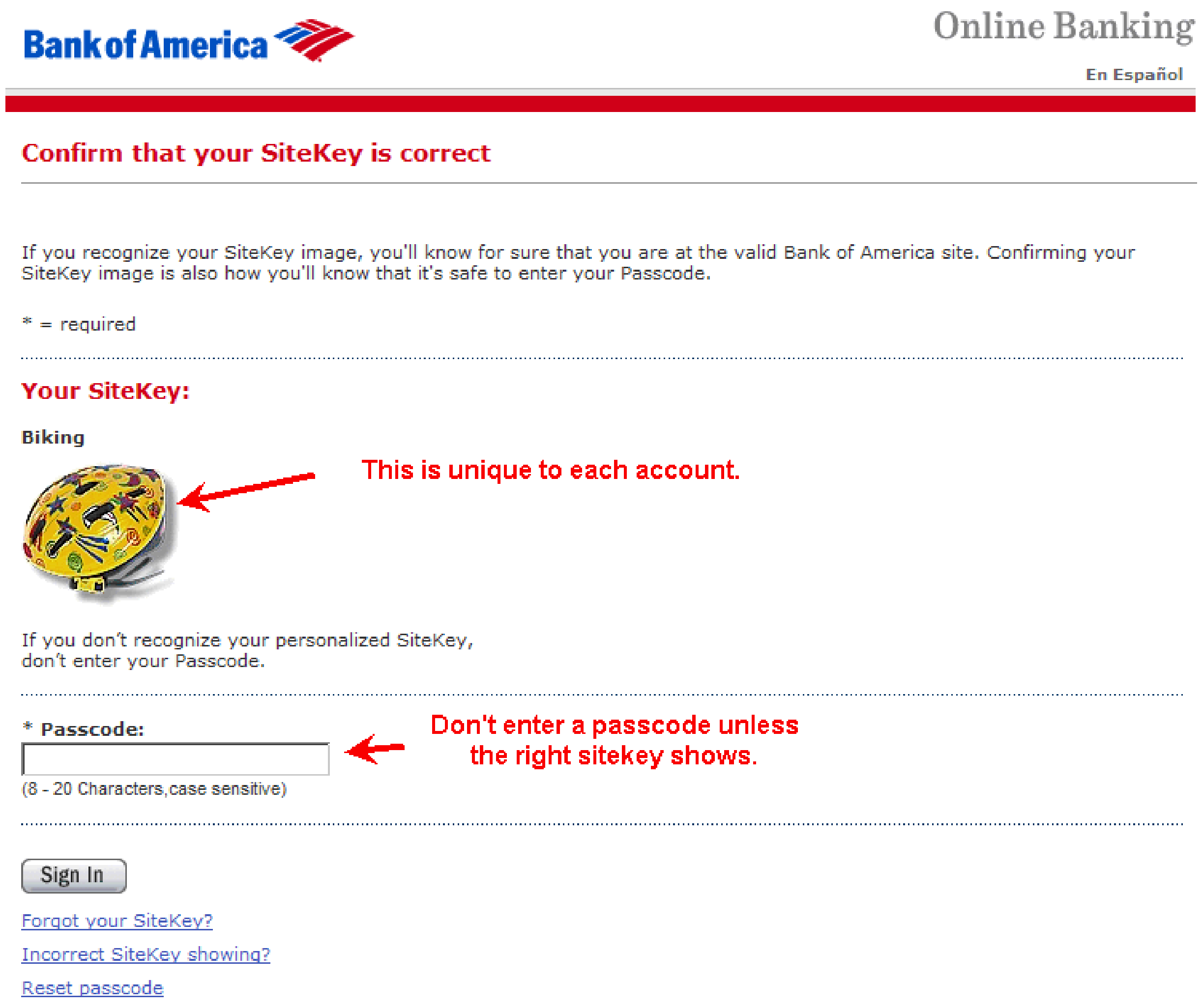
There is an enrollment process prior to usage of SiteKey where the user selects a secret image, a secret phrase, or both and their answers are stored on the server.

The authentication process for SiteKey is accurately described in its Wikipedia article[3] as follows:

1. User *identifies* (**not** authenticates) himself to the site by entering his username (but not his password). If the username is a valid one the site proceeds.
2. Site authenticates itself to the user by displaying an image and accompanying phrase that he has earlier configured. If the user does not recognize them as his own, he is to assume the site is a phishing site and immediately abandon it. If he does recognize them, he may consider the site authentic and proceed.
3. User authenticates himself to the site by entering his password. If the password is not valid for that username, the whole process begins again. If it is valid, the user is considered authenticated and logged in.

Any phishing website attempting to impersonate, for example, the Bank of America website would have to know which image and what phrase each user stored there. If the phishing website displays the incorrect information to the user, the user will be able to identify the phishing website as a forgery.

[2] How Bank of America SiteKey Works for Online Banking Security, http://www.bankofamerica.com/privacy/sitekey/
[3] SiteKey, http://en.wikipedia.org/wiki/SiteKey


Bank of America

Online Banking

En Español

Confirm that your SiteKey is correct

If you recognize your SiteKey image, you'll know for sure that you are at the valid Bank of America site. Confirming your SiteKey image is also how you'll know that it's safe to enter your Passcode.

* = required

Your SiteKey:

Biking

This is unique to each account.

If you don't recognize your personalized SiteKey, don't enter your Passcode.

* Passcode:

(8 - 20 Characters,case sensitive)

Don't enter a passcode unless the right sitekey shows.

Sign In

Forgot your SiteKey?

Incorrect SiteKey showing?

Reset passcode

SiteKey at the Bank of America website[4]

Social Vulnerabilities

The effectiveness of SiteKey rests on two assumptions, one of which is that users will not provide their password to a website with an incorrect SiteKey. In a controlled, academic study, 58 out of 60 participants who were shown an incorrect SiteKey image still provided their passwords[5]. While it may be possible through repeated training and negative reinforcement to force users to behave in the way the SiteKey authors intended, it is still user's natural inclination to disregard SiteKey as a security indicator.

Technical Vulnerabilities

The second assumption upon which the effectiveness of SiteKey relies is that a third party will be unable to reproduce the correct SiteKey image given an arbitrary username. This assumption has proved false as any action a human can do to retrieve their SiteKey is an action that a robot can replicate. In a simple demonstration using Perl and LWP, two students at Indiana University were able to perform a man-in-

[4] Thwarting Phishers in Online Banking, http://blog.maysoft.org/blog.nsf/d6plinks/FPAO-7CWLWA
[5] The Emperor's New Security Indicators, http://www.usablesecurity.org/emperor/emperor.pdf

the-middle attack against SiteKey. Their attack allows a 3rd party website to present the correct SiteKey given an arbitrary username[6].

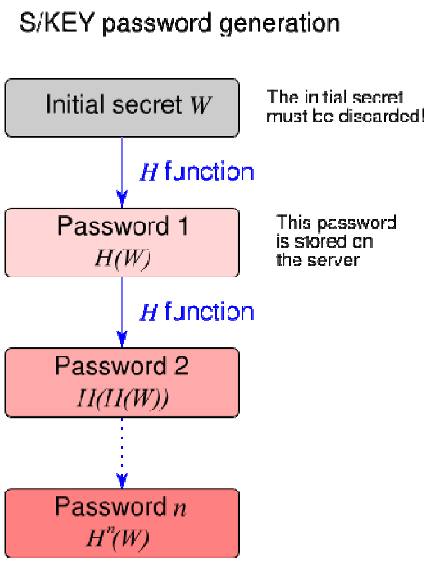
Another common vulnerability in SiteKey and other related authentication schemes are that users fail to randomly choose secret images. In some cases, the authenticating party fails to provide a large enough set of potential images to choose from. In other cases, users will gravitate towards a specific image or group of images more than others based on their content or their placement on the page. The low randomness of many SiteKey images gives an attacker a slim, yet significant, probability of displaying the correct SiteKey image by chance. When performing a large phishing attack, and combined with fact that many users will still provide their password despite an incorrect SiteKey, this may be all the attacker needs to consider his campaign a success.

One-Time-Passwords

One-Time-Passwords, or OTP, are single-use passwords that are usually generated in one of the following three ways:

1. Hash-Chaining

When using Hash-Chaining, passwords are generated by starting with a random seed and then generating passwords by hashing the seed once, twice, three times, etc. for as many single-use passwords are necessary. Passwords must be used in the order they were generated. The server discards the random seed and only stores the first password, as it is able to generate future passwords by reapplying the hashing function. This technique is called hash-chaining and it is most popularly used with the S/KEY system[7].


S/KEY password generation
Initial secret W
The initial secret must be discarded!
H function
Password 1
$H(W)$
This password is stored on the server
H function
Password 2
$H(H(W))$
Password n
$H^n(W)$

2. Time-Synchronized OTP

When using Time-Synchronized OTP, passwords are generated by embedding a small computer with an accurate clock into a hardware token, synchronizing its time with a server, and then combining a random seed and the time-of-day in a hash function at pre-determined intervals. The server shares a secret and the time-of-day with the token and is therefore able to generate the same OTPs at the same intervals. This technique is called Time-Synchronized OTP and it is most popularly implemented in RSA SecurID[8] hardware tokens.

[6] SiteKey Man-in-the-Middle Demonstration, http://www.phishcops.com/sitekeyMITM.asp
[7] S/KEY, http://en.wikipedia.org/wiki/S/KEY
[8] SecurID, http://en.wikipedia.org/wiki/SecurID

According to a message board post in comp.security.misc[9], current-generation RSA SecurIDs use AES in ECB mode to hash the following:

- 128-bit token-specific true-random seed
- 64-bit standard ISO representation of the current time (year/month/day/hour/minute/second)
- 32-bit token-specific salt (the serial number of the token)
- 32-bits of padding (reserved for future use)

The output of the hash is 6-8 digits or alphanumerics long and is regenerated every 60 seconds.

3. Challenge-Type OTPs

Challenge-Type OTPs are similar to time-synchronized OTPs, however, they also take a PIN number as input into the hashing algorithm.

Mutual Authentication with OTP

At a high level, mutual authentication with OTP works by having the user choose a password (something he knows) and providing the user with a hardware token that generates unique OTPs (something he has). The authentication protocol then works as follows:

1. User sends username and OTP challenge to server
2. User's token generates what the server should respond with
3. Server sends its unique response to user
4. User checks server's response vs. token's , and enters password if match

The user starts the authentication procedure by essentially requesting the server prove its identity by responding with a shared secret. When it is clear that the server and the client possess the same shared secret, the client sends over his password.

In many ways, mutual authentication with OTP tokens is analogous to the previously described SiteKey system, where users choose a secret image to be stored on the server. In that system the user identifies himself, is presented with a shared secret, and if the shared secret is correct, enters his password.

Technical Vulnerabilities

In a phishing attack, the phisher is in the perfect position to perform a man-in-the-middle attack that nullifies any enhancement a constantly rotating OTP would provide. As long as the steps in the man-in-the-middle attack take up less than the timeout of the OTP (60 seconds), a phisher would be able to proxy the victim’s responses to the server and log in to his account. Of course, to later log in to his victim’s account again, the attacker would have to re-phish the victim to obtain the current OTP.

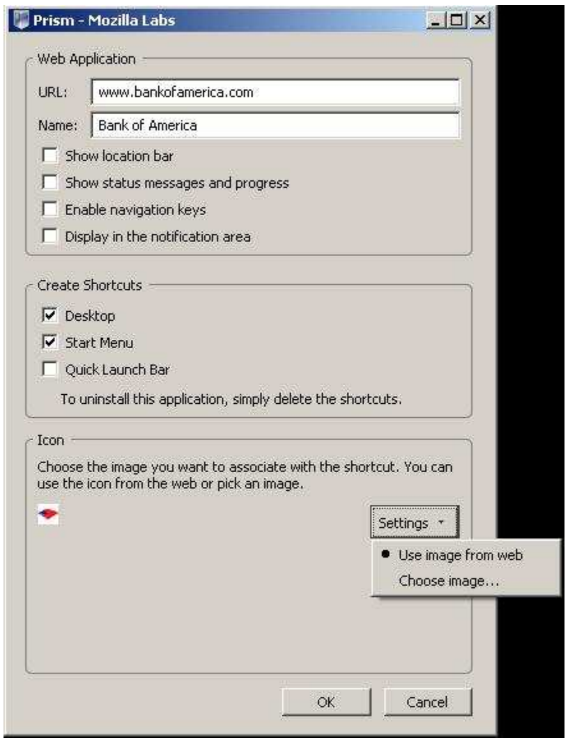
[9] http://groups.google.ca/group/comp.security.misc/msg/2de1eb5e8a73469a?dmode=source

Single-Site-Browsers

Technical Overview

A Single-Site-Browser, or an SSB, is a stripped-down web browser that constrains its users to visiting a single website. They generally lack a navigation bar, a status bar, bookmarks and other familiar browser chrome. SSBs usually come branded with the logo of the particular website they are locked to and look like a normal desktop application that has been installed on a user's computer. The Mozilla Foundation and The WebKit Open Source Project each have SSB frameworks made from their mainstream browsers: Mozilla Prism[10] and Fluid[11], respectively.

If opened directly, Prism will prompt the user to create a new bundle, or SSB configuration file. In the figure to the right, this prompt is filled out to create a new bundle for the Bank of America. The figure on the next page shows the SSB in operation alongside a normal instance of Mozilla Firefox. To return to the Bank of America SSB in the future, a user can simply open its shortcut.



Prism - Mozilla Labs

Web Application

URL: www.bankofamerica.com

Name: Bank of America

[ ] Show location bar

[ ] Show status messages and progress

[ ] Enable navigation keys

[ ] Display in the notification area

Create Shortcuts

[✓] Desktop

[✓] Start Menu

[ ] Quick Launch Bar

To uninstall this application, simply delete the shortcuts.

Icon

Choose the image you want to associate with the shortcut. You can use the icon from the web or pick an image.

Settings

● Use image from web

Choose image...

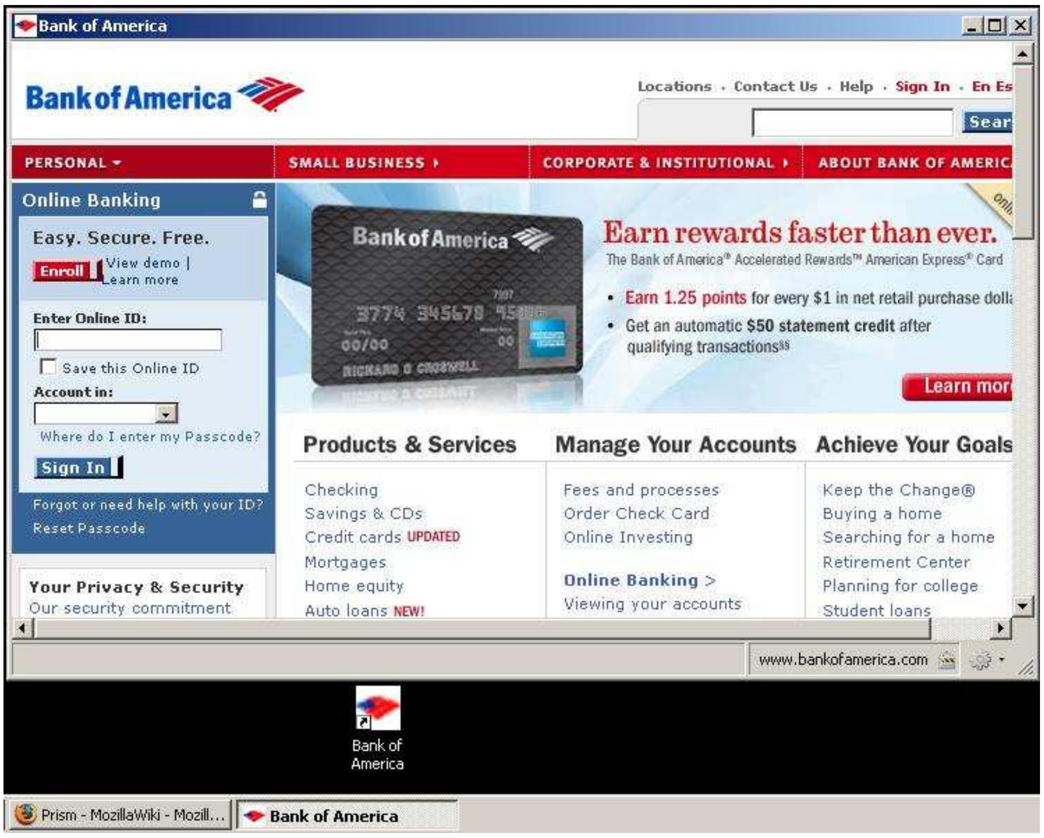
OK Cancel

Configuring a Prism SSB

Alternatively, Prism registers the .webapp file extension with the host operating system upon installation and bundles can be delivered as downloaded configuration files. These bundles are simple to create for the website owner and they will leave the same desktop shortcuts as the above method. The configuration file to create a Bank of America SSB would look like the following:

```
[Parameters]
id=boa-ssb@bankofamerica.com
uri=https://www.bankofamerica.com
location=no
sidebar=no
navigation=no
```

[10] http://wiki.mozilla.org/Prism
[11] http://fluidapp.com/



A Prism SSB in operation

Advantages of SSBs

In reality, the web already has somewhat of a server authentication system: DNS. Arguably the largest problem in phishing is that users are simply unaware and unable to identify and understand URLs and DNS. If there was user that could accurately decipher URLs and always paid attention to them, you would have an SSB. By locking users to a particular domain with an SSB, you ensure that, to the best ability of DNS, the user will end up in the same place every time.

Drawbacks to SSBs

While the same could be said of any other solution, it is important to note that if the user's local machine has been compromised then the integrity of the SSB cannot be guaranteed. While SSBs attempt to provide as much isolation as possible they are no substitute for endpoint security products like Anti-Virus (AV), application white-listing, and Network Admission Control (NAC). Locally installed malware, trojans, or keyloggers will be able to perform their attacks regardless of whether the user is surfing through an SSB or their normal browser.

The user’s own DNS resolver represents another potential avenue of attack. If the user's DNS resolver has been compromised, the SSB may not load the webpage it was “made for.” Such attacks are difficult; however, little can be done to mitigate this threat.

Technical Vulnerabilities

Single-Site-Browsers are at an early stage of development, the implementations of them have not yet fully matured, and their developers do not see them as a potential security technology. Hence, SSBs are missing essential features that are required for them to fully satisfy their potential to be more secure than normal web browsers.

The design goals for Mozilla Prism are currently listed as[12]:

- **Separate process:** When the webapp goes down or locks up, I don't want anything else affected. Thankfully, Firefox does have session restore, but that is beside the point. When I open many tabs and have several webapps running in a browser, things get slow and unstable after a day or two.
- **Minimal UI:** A generic browser UI is not needed for webapps. If any UI is present, make it specific to the webapp I am using.
- **Basic desktop integration:** Create shortcuts to start the webapp, add ability to show specialized icons in the tray or dock and ability to display notifications.
- **Platform with extensions:** I don't want to download a full browser runtime for each webapp. I do want to be able to add some custom code/features that are not directly supported in the webapp. I should be able to install one runtime and then get packages or extensions for each webapp. Think Firefox extensions or Greasemonkey scripts. These extensions should be able to tweak the SSB UI as well.
- **Open external links in real browser:** If I click a link in the webapp that opens a new site, don't change my webapp browser window. Open all external links in my default/real browser.

These goals embody many attractive principles in computer security -- least privilege, privilege separation, and low complexity for instance -- however, it is clear that, currently, their main focus is to provide a reliable mechanism to bridge web applications onto a user's desktop. The SSB platform has the potential to provide for many things, such as mitigations against Cross-Site-Request-Forgery (CSRF) and non-persistent Cross-Site-Scripting (XSS) attacks through the isolation of a user's cookies, however the absence of specific security goals in their development has left this potential unrealized. For example, Prism will load off-site content in an iframe or through an http-redirect even when it is locked to a particular domain. This unintended behavior breaks some of Prism's design goals, depending on how you read them, and allows an attacker to successfully execute XSS attacks which could result in the disclosure of the application users' authentication tokens.

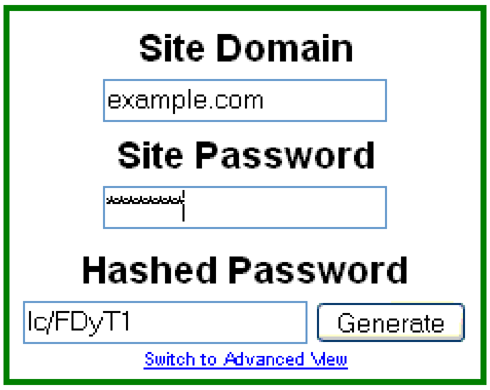
If SSBs develop the capability to enforce constraints on resource loading or assist in the deployment of encryption (X.509) keys they may become a more attractive platform for protecting a user's data.

[12] Prism / SSB Objectives, https://wiki.mozilla.org/Prism#Prism_.2F_SSB_Objectives

PwdHash

Technical Overview

PwdHash is a web browser extension that creates and manages domain-specific passwords[13]. When a user creates a password at alice.com, PwdHash creates a hash of the concatenation of ‘alice.com’ and the user's password and submits the hash to the website instead. If the user ends up at the phishing website alic3.com, PwdHash will perform hash(alic3.com + entered_password) which will result in a different hash than the hash given to the real alice.com. PwdHash does not constitute two-way authentication, rather it provides a fail-safe guarantee that only the legitimate server (as identified by DNS) will receive the correct password.



PwdHash

Social Vulnerabilities

PwdHash is plagued by usability problems that will discourage its use and prevent its adoption. Benign actions on the part of the server administrator such as changing the domain or location on the domain of the login box will affect the calculated hash value and prevent the PwdHash user from logging in Additional problems arise when the PwdHash user wants to log in from another computer without PwdHash. For these cases, PwdHash provides a panel for users to recover their hashed passwords so they can write them down and take their passwords with them; however, this process is unlikely to be discovered by many until they attempt to login to one of their PwdHash-protected accounts and cannot, causing great frustration.

It should be noted that a hard drive failure when using PwdHash is not a catastrophic event. PwdHash is simply a password-generating algorithm and it operates without the use of a salt or any other stored secrets. Therefore, upon operating system and PwdHash re-install, the user will find that the same passwords they had been using will continue to work.

Technical Vulnerabilities

As with previous authentication schemes, the effectiveness of PwdHash relies on the dependability of DNS. An attacker that is able the control the victim’s DNS resolver will have no problem obtaining their password for the domain.

To increase its usability, the PwdHash authors chose to only generate hashes based on the 2nd-level domain of the target server rather than the full URL or the FQDN, both of which likely change too often. To attack this, a phisher could poison the victim’s DNS for abc123.target.com (an easy task, as this domain does not exist) and set up his landing page there. The PwdHash would be the same at this domain and the phisher would collect the correct password.

[13] http://crypto.stanford.edu/PwdHash/

Mutual Authentication with TLS

Technical Overview

It is a little known ability of TLS to provide mutual authentication through the use of client certificates. This ability is rarely taken advantage of due to the large infrastructure costs associated with maintaining your own Certificate Authority (CA) and other usability problems. Two popular users of this type of authentication are the US Department of Defense (DOD) and Amazon in their Web Services division (AWS, EC2, S3, SQS, etc).

Steps to setup mutual TLS authentication

To begin, the server administrator must set up a Certificate Authority to serve as the root of trust. This should be done on a separate computer from the web server, as it is critical that the CA not suffer from any security problems. If a CA is broken into or suffers from other security problems, none of the certificates signed by that CA can be trusted and the entire trust hierarchy must be rebuilt.

The web server must now create itself a certificate and present a certificate signing request to the CA. This signature on the web server certificate will prove to the user, via the CA, that they have arrived at the right server. This signed certificate and the CA certificate should then be installed on the web server. Users will now be able to access the web server over HTTPS and view that it has a TLS certificate that has been signed by their own CA.

Client certificates are created exactly the same way that server certificates are i.e., generate an additional certificate and then present a certificate signing request for it to the CA. Be careful to associate the proper common name and e-mail address with it as they will be used to authorize the user to protected resources. To be used as a client certificate, you must convert the file to PKCS#12 format and have it, and the organization's CA certificate, imported into the user's web browser.

Finally, you must configure the web server to verify the CA of the user certificates presented to it, perform access-control based on client certificate fields, and install a Certificate Revocation List (CRL) to manage deleted users. On the Apache web server, you can verify client certificates match the installed CA and specify a CRL with the following three lines in a configuration file:

```
SSLVerifyClient require
SSLVerifyDepth 2
SSLCARevocationFile conf/certs/CA/cafesoftCA.crl
```

To configure directory-level access control based on client certificates, you can add the following to a directory block in Apache. In this example, only client certificates signed by the root CA with the common name "Basic Checking Customer" will be able to access the specified directory.

```
<Directory "/var/www/htdocs/basic_checking">
<IfDefine SSL>
SSLRequireSSL
SSLRequire %{SSL_CLIENT_S_DN_CN} eq "Basic Checking Customer"
```

```
    </IfDefine>
    </Directory>
```

A detailed tutorial explaining each of these steps can be found on the CafeSoft website[14].

Drawbacks to Mutual Authentication with TLS

Mutual authentication with TLS provides a cryptographically robust method of authentication that is based on simple standards such as HTTP and TLS, however, this enhanced resistance attacks does not come without downsides.

- How do you let your users securely interact with your CA? Many organizations who choose to use client-side certificates enroll users through a manual process that includes some amount of physical verification due to the high amount of trust placed in TLS certificates.
- Client-side certificates are typically stored as files on the user's computer, which can be misplaced, or lost along with a stolen laptop. It is comparatively "cheaper" to change a password than to replace a certificate.
- If the user's certificate is stored on their computer as a file, it is possible for their client certificate to be compromised if their computer is compromised (for instance, via malware).
- For the above reasons, many organizations choose to deploy client-side certificates inside of hardware tokens, separate from a computer. This choice comes with a large monetary cost in deploying the tokens and card readers to users, as well as additional support for their installation and maintenance.
- It is still possible for users to be tricked into giving up their certificates through social engineering or phishing, even if it requires more steps than simply entering a password into a form.

OpenID

Overview

OpenID is a general-purpose identity management system that allows a user to create a single on-line identity (which may be a screen name and not a real name) and use that credential to authenticate to a large number of web services. Several similar systems exist, such as CardSpace, The Higgins Project, and Parity. OpenID is attractive to some because it does not require the user to create and store a password at each website they wish to have an account with nor does it expose the user's password to each website he authenticates to.

These systems do not, in and of themselves, authenticate the individual when the identity is created to ensure the identity truly represents any particular person. For example, every AOL user has an OpenID, which is their screen name, created by AOL. Some organizations, such as financial institutions or other

[14] Configuring Apache 2.0 for SSL/TLS Mutual Authentication using an OpenSSL Certificate Authority, http://www.cafesoft.com/products/cams/ps/docs30/admin/ConfiguringApache2ForSSLTLSMutualAuthentication.html

"identity providers," may provide an enhanced enrollment process in which the true identity of the user is verified before the on-line identity credentials are granted. The enrollment process is the key to the amount of trust that can be put into the on-line identity truly representing a given individual.

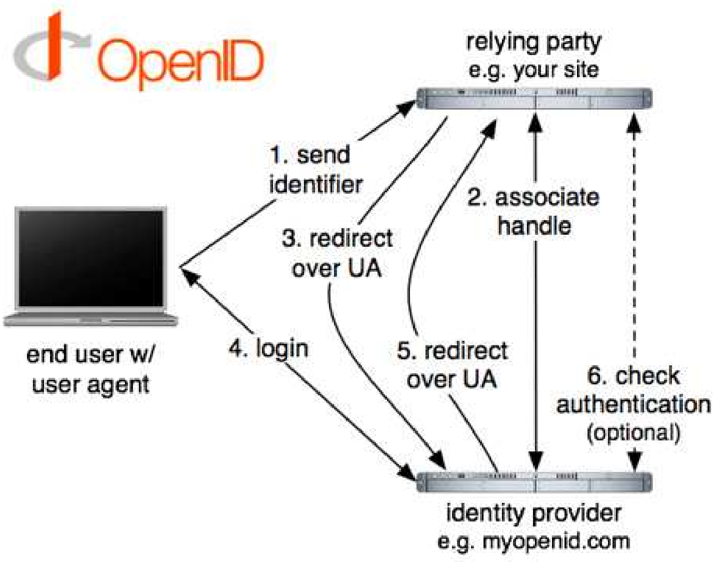
Technical Overview

OpenID is a decentralized single-sign-on system that involves three "parties"[15]:

- **End User:** the person who wants to assert his or her identity to a site.
- **Relying Party:** The site that wants to verify the end-user's identifier. Sometimes also called a "service provider".
- **Identity Provider:** A service provider offering the service of registering OpenID URLs or XRIs and providing OpenID authentication (and possibly other identity services). Note that the OpenID specifications use the term "OpenID provider" or "OP".

When a user attempts to log in to an OpenID-enabled service provider, the service provider contacts the identity provider to verify the end user. This interaction is defined in the OpenID Authentication Protocol 2.0 document. The steps in the 2.0 protocol are defined as follows[16]:

1. The end user initiates authentication by presenting their identifier to a relying party.
2. The relying party performs discovery on the identifier and establishes the location of the identity provider.
3. (optional) The relying party and the identity provider establish an "association" (a shared secret established with Diffie-Hellman Key Exchange). The identity provider signs subsequent messages and the relying party verifies them, removing the need for subsequent direct requests to verify the signature after each authentication request/response.
4. The relying party redirects the end user to the identity provider.
5. The identity provider establishes whether the end user is authorized to perform OpenID authentication and wishes to do so. The manner in which the end user authenticates to their identity provider and any policies surrounding such



OpenID 1.x Login Process

[15] http://en.wikipedia.org/wiki/Openid
[16] http://openid.net/specs/openid-authentication-2_0.html

authentication is out of the scope of this document.

6. The identity provider redirects the end user back to the service provider with either an assertion that the authentication is approved or a message that authentication failed.
7. The relying party verifies the information received from the identity provider including checking the return URL, verifying the discovered information, checking the nonce, and verifying the signature by using either the shared key established during the association or by sending a direct request to the identity provider.

Vulnerabilities

The complexity of the OpenID single-sign-on system lends itself to a similarly large attack surface, one which researchers have almost been stumbling over themselves to find holes in[17][18][19][20][21]. Of the many security issues that have been uncovered in the protocol, the most serious issue may be that redirection to the identity provider is under the control of the service provider that the end user is logging in to. Paul Crowley explains the threat as the following[22]:

> "I decide to comment on a badguys.com blog entry, so I go to log in. I get redirected to livejourna1.com (note the 1) and presented with a log in page. I wonder briefly what happened to my LJ login cookie, and type in my username and password. badguys.com and livejourna1.com conspire seamlessly to make it look like a successful login attempt.
>
> The thing that makes this attack cunning is that (1) it won't ring any alarm bells in me - unlike an email saying "For security reasons, LiveJournal requires you to validate your login, please click the link below", everything that happens is completely part of the normal course of events, including events after typing in my password - and (2) it captures my SSO password, making it a valuable target for phishing attacks."

Additional issues arise due to step 5 of the OpenID Authentication Protocol 2.0 document:

> *The manner in which the end user authenticates to their identity provider and any policies surrounding such authentication is out of the scope of this document.*

There is no centralized organization which evaluates the trustworthiness of an identity provider or the authentication mechanisms it chooses to use due to OpenID's nature as a decentralized protocol. In the worst case, one could create an OpenID provider for the sole purpose of stealing its user's identities. In a better case, a legitimate identity provider could be compromised due to poor security practices as they are not vetted the same way that Certificate Authorities are[23].

[17] "OpenID: Phishing Heaven." http://www.links.org/?p=187
[18] "Phishing attacks on OpenID." http://lists.danga.com/pipermail/yadis/2005-June/000470.html
[19] "OpenID still open to abuse." http://www.computing.co.uk/itweek/comment/2184695/openid-open-abuse
[20] "Beginner's guide to OpenID phishing." http://marcoslot.net/apps/openid/
[21] "CVE-2008-3280." http://www.links.org/files/openid-advisory.txt
[22] "Phishing attacks on OpenID." http://lists.danga.com/pipermail/yadis/2005-June/000470.html
[23] WebTrust Trust Services. http://www.webtrust.org/index.cfm/ci_id/43988/la_id/1.htm