A simple proposal for an S/Key based secure protocol for HTTP interactions

By, Fotios Basagiannis

This is an initial approach to specifying an S/Key like secure protocol of communication between a web client and a a web server. S/Key will be used as a means of detering replay attacks on HTTP requests, including their headers (headers like cookies that usually include sessioning information).

Please note that this paper is at its very initial stages. Please excuse any omissions and the poor presentation.

What is a replay attack?

A replay is another type of attack in which a login transaction is captured as it passes over an open HTTP connection. Microsoft Passport is currently vulnerable to replay attacks, even though the ticket and profile are encrypted, because they're exchanged over an open HTTP connection. Someone listening in on the conversation could capture these packets and replay them, which would allow this hacker to impersonate the user until their login ticket expires.

What is Passport .NET authentication?

The .net passport authentication routine is described here: ASP .NET Passport

Quoting from this online spec:

Passport is a cookies-based authentication service. A sample transaction conversation using Passport authentication might look similar to the following:

1. A client issues an **HTTP GET** request for a protected resource, such as http://www.contoso.com/default.aspx.
2. The client's cookies are examined for an existing Passport authentication ticket. If the site finds valid credentials, the site authenticates the client. If the request does not include a valid authentication ticket, the server returns status code 302 and redirects the client to the Passport Logon Service. The response includes a URL in the query string that is sent to the Passport logon service to direct the client back to the original site.
3. The client follows the redirect, issues an **HTTP GET** request to the Passport logon server, and transmits the query string information from the original site.
4. The Passport logon server presents the client with a logon form.
5. The client fills out the form and does a **POST** back to the logon server, using Secure Sockets Layer (SSL).
6. The logon server authenticates the user and redirects the client back to the original URL (http://www.contoso.com/default.aspx). The response contains an encrypted Passport cookie in the query string.
7. The client follows the redirect and requests the original protected resource again, this time with the Passport cookie.
8. Back on the originating server, the PassportAuthenticationModule detects the presence of the Passport cookie and tests for authentication. If successful, the request is then authenticated.

Subsequent requests for protected resources at the site are authenticated at the originating server using the supplied ticket. Passport also makes provisions for ticket expiration and reusing tickets on other member sites.

Passport uses the Triple DES encryption scheme. When member sites register with Passport, they are granted a site-specific key. The Passport logon server uses this key to encrypt and decrypt the query strings passed between sites.

It is quite obvious from the spec above that passport .NET does not protect from replay attacks

The S/Key Commando Authentication Routine

My proposed framework for strongly authenticated http requests is based on S/Key (with a twist) and goes something like this:

1. A new web user goes to the site's login page and expresses interest in opening a new account.

2. A site admin requests desired password from user via phone (preferrably mobile) or pgp protected e-mail (the latter is strongly recommended). This can be avoided but with considerable administrative privileges lost - see end of this paper.

3. The server side routine that inits the account applies an MD5 hash on that password say 100 consecutive times. MD5/SHA1 hashes are not reversible.

4. The user is informed that the account is now operational.

5. When the user tries to login, the web client first does a simple GET or POST that asks for the index number of the hashed password that the server expects for that user. At this phase, only the user name is transmitted and we can even have just that transmitted encrypted to the server's well known public key.

6. This first time the server responds 100.

7. The web client performs 100 MD5 hashes (using JavaScript) on the suplied (in the appropriate text-box) password and sends that (can be encrypted as well but even if it is not, it does not make it much less secure as it is a one time only password) to the server.

8. The server compares the received string to the one it has (after the consecutive hash operations) and if it is correct, allows login to proceed. It also now expects hash index 99 for the next http request (if we want to be totally secure, not just a login request but any request, as all the requests that follow the login request are indeed susceptible to replay attacks). Therefore no single HTTP request uses the same pass and therefore cookie. All cookies are one time only with an attacker having no way to guess the next valid cookie.

Some Comments on the S/Key Commando auth routine

A tricky point here is that the web client's requests should be strictly serialized (one after the other) for this to work as expected. You cannot have two secure http requests starting at the same time.

This can be done by either:

- Applying authentication to html page requests only (as browsers may download page images and other secondary page content concurrently)

- Having javascript build the page contents dynamically from content that it retrieves from a javascript engine that serializes requests to the server by procesing a queue of requests for content for various areas of the html page. This would make all requests, regardless of their content type, be serialized and S/Key authenticated.

In the case of using S/Key Commando on all requests. all server replies would update the cookie that records the current hash index. This way we won't have to make multiple MD5 iterations on the client for every single request - the iterations will only be done once, during initial login, and all MD5 hashes stored in a client local array for future (meaning, this session only) use.

When the index reaches 1 (it should never reach 0 as this is the unhashed secret password), the server runs a new 100 iterations on the MD5 routine and updates the record that specifies the next expected one time password for that user. The client follows suit accordigly (the old array should not be used, at this time a random seed update occurs on both the client and the server, with the seed being transmitted from the server to the client in clear text - doing that does not decrease security). The client knows to follow suit because the cookie value (returned in the last client reply) also changes to reflect the new expected index.

Bear in mind that this is not fully S/Key compliant as true S/Key requires that not even the server knows the secret password (all it knows is the initial one time password to expect - then when it receives the next from the client, it does an MD4 or 5 hash on that and compares the result to the initial one). However, web admins may not want to relinquish possible administrative priviledges (like knowing the password of all users), plus a full S/Key compliant server implementation would make the S/Key Commando server implementation more complex.

Requiring that the admin knows the pass, makes the second step of the process (interaction with the admin in creating a new account) necessary and even vulnerable to eavesdropping (e.g. if a phone is used). However, use of strong assymetric encryption over email (e.g. PGP email) removes this vulberability.

Encryption can be added on top of this web request authentication mechanism for securing the privacy of the transmitted data.

S/Key Commando Implementation

S/Key Commando's ambition is to become a pluggable component for existing web sites that once introduced will make them s/key authenticated and impervious to replay attacks with only minimal changes needed (and those only to the existing login page logic of the site).

S/Key Commando consists of a client component done in cross-browser javascript and a server component that will be server specific (e.g. ASP, PHP, or whatever). The server component will be inserted at the top of each existing server-side page, in the form of an appropriate "include directive" in the local dialect.

The client component will be deliverable by the login page and will consist of

- An xmlhttp component (that is for IE browsers; Gecko based browsers have a similar component) for managed content pulling
- A request serialization engine based on a managed queue (similar to the one I have built for the Soq system)
- Javascript binary processing code for fully standard compliant SHA hash generation (the server component will also contain a similar SHA component)
- Logic that creates an IFrame for displaying the rewritten content as it comes from the server.

Note that the SKey commando - using advanced camo techniques - will be completely hidden from view. Note also that there will be a proprietary protocol for S/Key hash index synchronization when such things like a killed socket occur.

The page content will be rewritten so that any links it contains (only those pointing back to the site's domain/path) will invoke commando's url relay logic instead of invoking the web site's urls directly. This way the requests will be serialized and managed by the commando component and each request will contain the appropriate S/Key cookie in the proper sequence.

The component will be able to apply S/Key security to all requests (including even images) or just ASP pages. For securing images, the server has to be able to process plain image web requests (which cannot be normally done in ASP but it can be done with something more powerful like ISAPI).

Things like forms will be screen scraped and packed in specially coded POST requests that will be compiled by S/Key Commando. All incoming cookies will be recorded and appropriately replayed.

Also note that the S/Key conmmando's cookies will work on top of any existing cookies (like automated ASP SessionID cookies) and will not interfere with them.

Commando's cookie will contain the userid in plaintext and the appropriately hashed pass.

Note that the IFrame will be used as the area where the secured page renders.
I am also looking into ways of easily integrating existing login pages with the requirements of S/Key Commando.

Reservations

I realize that there are various tech challenges in making the S/Key Commando really work, however I feel that I have the abilities and the expertise/experience to resolve them quite successfully and thus introduce a system that will significantly increase the security of any site with very minimal hassle and extra work.

Notes

* As long as there is no encryption, an eavesdropper can see the content of the client/server transactions (S/Key or not). However, with S/Key, the eavesdropper cannot perform replay attacks that would allow him/her to either see information again (some time in the future) on his own or perform some action on the server again (that would probably change some kept state on the server that the legitimate user did not mean to)

* SSL explicitly protects against replay attacks by using a sequence number in each direction. Thus, Commando is just a way to improve your site's security without having to pay for certificates and without needing an SSL compatible web server. Commando does not do encryption though. However, Commando has the added advantage that a password in never transmitted (even encrypted), assumming that nested hashes are harder to reverse than breaking the SSL encryption of the password - if there is a password.

* S/Key Commando can be extended to support encryption as well.
  - The SSL protocol could be emulated at the application protocol level