SessionLock: Securing Web Sessions against Eavesdropping

Ben Adida
Center for Research on Computation and Society (CRCS), Harvard University
Children's Hospital Informatics Program (CHIP), Harvard Medical School
Cambridge, MA, USA
ben@eecs.harvard.edu

ABSTRACT

Typical web sessions can be hijacked easily by a network eavesdropper in attacks that have come to be designated "sidejacking." The rise of ubiquitous wireless networks, often unprotected at the transport layer, has significantly aggravated this problem. While SSL can protect against eavesdropping, its usability disadvantages often make it unsuitable when the data is not considered highly confidential. Most web-based email services, for example, use SSL only on their login page and are thus vulnerable to sidejacking.

We propose SessionLock, a simple approach to securing web sessions against eavesdropping without extending the use of SSL. SessionLock is easily implemented by web developers using only JavaScript and simple server-side logic. Its performance impact is negligible, and all major web browsers are supported. Interestingly, it is particularly easy to implement on single-page AJAX web applications, e.g. Gmail or Yahoo mail, with approximately 200 lines of JavaScript and 60 lines of server-side verification code.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*

General Terms

Design, Human Factors, Security

1. INTRODUCTION

The core component of the World Wide Web, HTTP [6], began its life as a stateless protocol: the page's HTML and all images were each downloaded using a new HTTP request made over its own TCP/IP connection. To provide a personalized user experience, early web "sessions" were implemented using tokens inserted into individual URLs, so that every click would send this session token back to the server. The tediousness of this approach and the fact that a new browser window would not automatically inherit this token made web sessions fairly unreliable.

In 1995, Netscape introduced cookies, small chunks of data that a web server can assign to a browser using HTTP return headers, which the browser is expected to send back to the server on every subsequent request. Using cookies, the web was made stateful. Over the years, in order to protect users' security and privacy, the details of cookie handling have become quite intricate, but the basic functionality remains: the server assigns the browser a token, and the browser sends this token back to that specific server on every subsequent request.

Web sessions are vulnerable to eavesdropping. A static token sent over a plaintext channel is obviously insecure: a network eavesdropper can easily read this token and replay it to tap into the victim's session, effectively impersonating the user for the length of the session. Interestingly, web sessions have not evolved much since the first days of web cookies: they remain quite vulnerable to eavesdropping.

Wi-fi networks make things worse. Wireless ("wi-fi") networks are now ubiquitous. Though wireless standards provide for password-based access-control and transport-layer encryption, wireless base stations found in hotels, conference lobbies, coffee shops, and airports are configured without this level of protection. Users connect freely to the wireless base station, and only then are asked to provide login credentials. This approach allows wireless operators to manage per-user password-based access control, rather than a single password for the wireless base station.

In this setting, eavesdropping on HTTP traffic and stealing web session tokens is so easy that it has recently received a new name: "sidejacking" [9]. Numerous common web applications, including most online webmail providers, are vulnerable to these trivial eavesdropping attacks.

SSL is not for everyone. One way to prevent eavesdropping is to use SSL to encrypt all web traffic. This approach is employed by financial institutions that cannot afford to have their clients' web sessions hijacked so trivially (or their customers' financial data read as easily as sniffing the network). When traffic is encrypted, an eavesdropper is powerless.

Unfortunately, delivering a web application over SSL triggers numerous complications. Even with significant server-side computational power, SSL's caching behavior, its need to download a complete resource and verify it before displaying any part of it, and its all-or-nothing nature result in a significantly more "sluggish" experience for the user.

Services like Gmail, Yahoo, Hotmail, Facebook could easily afford to deploy SSL: all of them use SSL to secure users' password at login time, acceleration hardware for SSL makes the server-side computational overhead quite manageable, and Gmail happily lets adventurous users access their mail

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

at `https://google.com/mail/`. However, likely because of the usability issues described above, all have chosen to deliver the bulk of their features over plain HTTP, leaving session tokens available for any eavesdropper to hijack.

Better security without SSL. We propose SessionLock, a method to improve the security of plain HTTP sessions. We use SSL in exactly the same way that Gmail, Yahoo, Hotmail, and Facebook already do: only to set up the session. In addition to the session identifier, we generate a session secret which is never sent over plain HTTP. This session secret is used by the browser to generate an authentication code for every HTTP request. The secret is passed from the HTTPS login page to the HTTP portion of the site, and from one page to another under HTTP, by way of the *URL fragment identifier.* Thus, although all URLs after login are requested over HTTP, the secret is never sent in plaintext over the network. Details are shown in Figure 1.

With SessionLock, the properties of HTTP, including progressive rendering of images and efficient caching, are preserved. With only private-data-containing URLs affected, images, scripts, stylesheets can all be delivered over HTTP without additional overhead, exactly as they are delivered today without SessionLock.

We implemented SessionLock using only a small JavaScript library and a simple server-side filter on protected requests. No browser add-on is required, and all major modern browsers are supported: web applications like Gmail can deploy this solution immediately, with very little code, and no action (or even awareness) required of their users.

Getting Closer to User Intuition. It is relatively intuitive for average users to understand that unencrypted wireless traffic can be "overheard": browsing over HTTP at a conference is a bit like having a private phone conversation on a crowded bus, where your neighbors might easily catch snippets. Session hijacking, on the other hand, is quite unintuitive: after all, login pages are usually served using SSL, the padlock icon is visible in the browser, and the average user would be justified in thinking that her session is safe from attackers. One goal of SessionLock is to make sure that reality matches this intuition more closely: having one's HTTP traffic overheard is still a concern, but having one's session hijacked is not.

1.1 This Paper

In Section 2, we review the current approaches that web developers can take to secure their users' sessions. In Section 3, we consider the simple SessionLock building blocks, and in Section 4 we describe SessionLock in detail. In Section 5, we consider some immediate extensions to the basic scheme. We evaluate implementation and performance issues in Section 6, highlight a number of points for discussion in Section 7, and review related work in Section 8.

2. CURRENT PRACTICES

Though techniques for maintaining web sessions have evolved since the early days of the Web, they have remained surprisingly stable.

2.1 Web Sessions

Maintaining web session state requires having the web client provide some unique identifier to the web server on every request, so that the server can identity each HTTP request more precisely. The earliest solution to this problem required the server to dynamically embed this unique token in every URL of every HTML page. A number of web development frameworks still offer a way to automate this URL-token embedding.

Since Netscape 1.1 in 1995, web browsers support *cookies*, which allow a web server to send, in an HTTP response, a special header:

```
Set-Cookie: session_id=8b3xdvdf3jg;
```

This header can also specify a number of additional fields, including:

- an expiration date,
- a secure flag, indicating whether this cookie should only be sent back over SSL,
- the domain, so that `foo.example.com` and `example.com` can share cookies if they so choose,
- the path, so that different sections of a site, e.g. `/foo/*` and `/bar/*` can have different cookies.

On every subsequent request, the web client will include all the pertinent name-value pairs it has received from that specific server. The security of these data is highly dependent on the transport layer: cookies sent over HTTP are easily accessible to a network eavesdropper.

2.2 Digest Authentication

HTTP offers protocol-level authentication, including the particularly interesting digest mode [7], which all modern browsers now support. In digest auth, just like in plain auth, the web browser provides a distinct user interface to prompt the user for her username and password. Unlike in plain auth, digest auth provides a challenge-response mechanism for sending along the password, which ensures that a network eavesdropper cannot extract the password. Web services could use digest auth as a way to secure sessions against eavesdropping.

Unfortunately, HTTP-based authentication has been shunned by most web services for a number of reasons [18]:

1. The HTTP-password-entry user interface cannot be customized or integrated into a more complete form, making it difficult for users to proceed if they've forgotten their password or want to register a new account.
2. On the server side, authentication is handled by the HTTP stack which *must* have direct access to a username/password database, since it needs to handle the challenge-response before handing over processing to the application code.
3. The server cannot easily trigger a logout, for example after some period of inactivity.

As a result, HTTP authentication, even in digest mode, is not likely to provide a deployable defense against eavesdropping.

2.3 Locking Sessions to IP Address

One natural reaction to the eavesdropping problem is to bind web sessions to the user's IP address at the time of session initiation: if a session token is received from a different IP address, the web server can prompt the user to

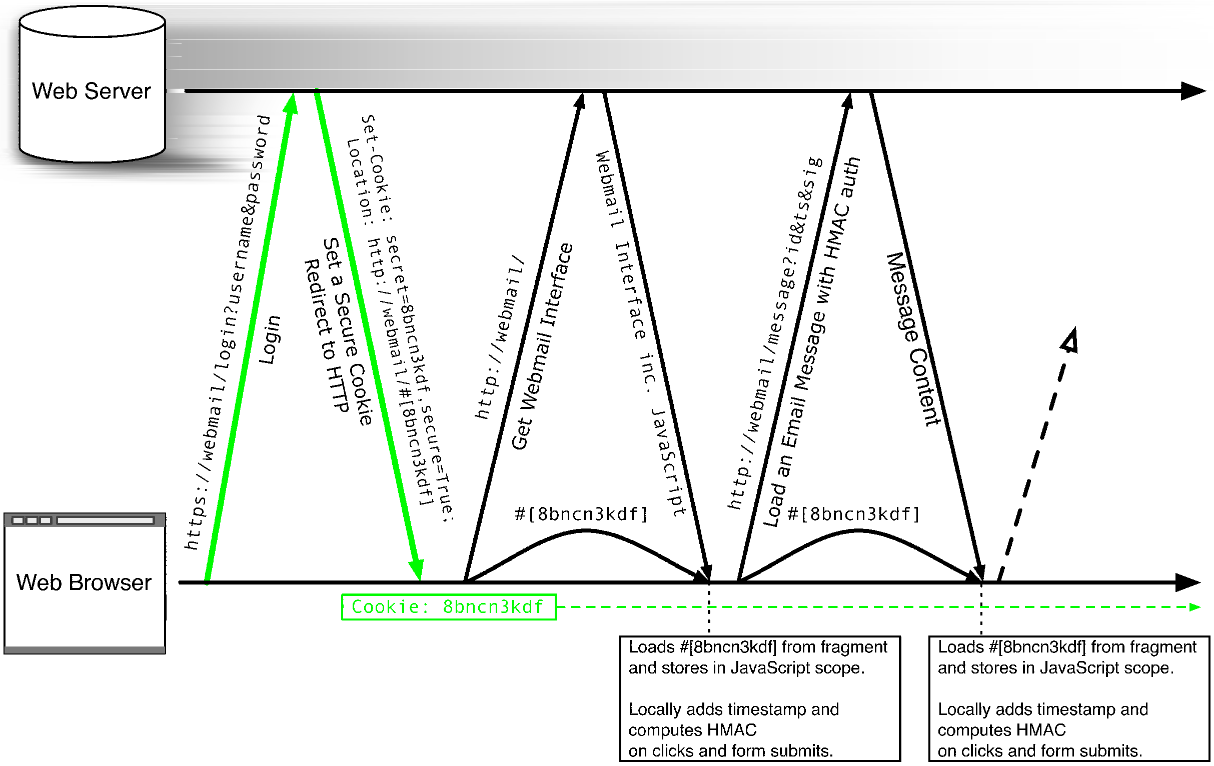

Web Server
Web Browser
https://webmail/login?username&password
Login
Set-Cookie: secret=8bncn3kdf,secure=True;
Location: http://webmail/#[8bncn3kdf]
Set a Secure Cookie
Redirect to HTTP
http://webmail/
Get Webmail Interface
Webmail Interface inc. JavaScript
#[8bncn3kdf]
http://webmail/message?id&ts&sig
Load an Email Message with HMAC auth
#[8bncn3kdf]
Message Content
Cookie: 8bncn3kdf
Loads #[8bncn3kdf] from fragment and stores in JavaScript scope.
Locally adds timestamp and computes HMAC on clicks and form submits.
Loads #[8bncn3kdf] from fragment and stores in JavaScript scope.
Locally adds timestamp and computes HMAC on clicks and form submits.

Figure 1: The SessionLock Protocol: An initial SSL exchange sets up the session secret, which is passed from HTTPS to HTTP, and then from one HTTP page to the next, using the fragment identifier. Note how the use of the fragment identifier effectively creates a client-only channel from one page to the next. Each HTTP request is then timestamped and HMAC'ed with this secret for authentication. SSL portions are noted in green. The secure cookie stays around in case the secret needs to be recovered and resent into the HTTP realm (a process not represented here.)

re-authenticate. Unfortunately, especially in our important wi-fi use case, many users surf the web behind a Network Address Translator so that many users are effectively using the same IP address. In our specific use case, the attacker on the same wi-fi network is, by default, already using the same external IP address as the victim. From the point of view of the server, if an attacker can steal a victim's cookie behind a network router, there is no detectable difference between the victim and the attacker, and IP-address-binding is useless.

2.4 SSL

SSL provides end-to-end encryption between the web server and browser, clearly foiling passive eavesdroppers. Unfortunately, SSL requires more work on the server side and, more importantly, triggers a number of sub-optimal behaviors on the client side.

An SSL server must run on its own IP address (no virtual hosting), because the SSL certificate handshake occurs before the browser is able to specify a virtual hostname [2] [1]. In addition, an SSL server must deliver all resources, including static graphical layout elements that typically require no protection, under computationally intensive SSL in order to prevent browser warnings about mixed content. This typically prohibits the use of latency-reducing, geography-based caching by content-delivery networks.

In addition, web browsers behave differently under SSL. Resources are not cached nearly as well or as often, increasing the average page load time over the course of a web session. Resources cannot be displayed until they are fully loaded and their signature verified, preventing progressive loading and generally making the application feel more sluggish than the identical site over plain HTTP.

As a result of these complications, especially those which raw server computational power cannot address, a number of common web services are delivered over plain HTTP, with only the login page processed using SSL to protect the password. Interestingly, even if SSL is offered as an option, the existence of a service over plain HTTP is sufficient to exploit the weakness with a minor social engineering attack that surreptitiously tricks the user into visiting the plain HTTP URL, thereby leaking the cookie into an insecure network [2].

3. BUILDING BLOCKS

We now cover the SessionLock building blocks. We note that the technical components are particularly simple and require only a cursory explanation.

[1] Server Name Indication (SNI), a TLS extension [4], provides support for virtually hosted secure web sites, but not all browsers and servers support it yet.

[2] `http://seclists.org/bugtraq/2007/Aug/0070.html`

3.1 Fragment Identifier

The URL specification [3] defines the *fragment identifier*, the portion of the URL that follows the # character. As its name implies, the fragment identifier designates a portion of the resource. For example, consider the following URL:

```
http://host/rest/of/url#paragraph4
```

Here, #paragraph4 is the fragment identifier. When the primary resource, in this case http://host/rest/of/url, is an HTML document, the fragment identifier tells the browser to scroll the viewport to the section of the document that reads:

```
<div id="paragraph4">
    ...
</div>
```

When no such portion of the document exists, the browser doesn't scroll, and the fragment identifier remains in the URL, unused.

The fragment identifier is never sent over the network: the browser requests the full resource and uses the fragment identifier to scroll. In other words, the web server is never aware of the fragment identifier to which a user navigates: a user can click from one fragment to another within a page, with his browser scrolling automatically to the appropriate location, never performing any additional network request.

Though it is never sent over the network, the fragment identifier does appear in the browser's URL bar. As a result, it is accessible to JavaScript code running within the page using the command document.location.hash.

3.2 Authenticating Web Requests with HMAC

Simple message authentication between two parties with a shared secret is easily achievable using a Message Authentication Code (MAC) [16] algorithm. In particular, HMAC [10] is a hash-function-based message authentication technique which is easily implemented and quite efficient in just about any programming environment, including browser-based JavaScript.

A number of web-based APIs, including Google APIs [3] and the Facebook Platform [4] already use HMAC for authenticating requests. Typically, web clients and the web service share a secret. When making an HTTP request, the client prepares the entire request including all parameters and a timestamp, HMACs the full request string using the shared secret, and appends to the request an additional HTTP parameter whose value is the resulting HMAC. The server verifies the timestamp and re-computes the expected HMAC on the rest of the parameters (minus the HMAC parameter itself), checking it against the HMAC submitted by the client. Though the same result could be accomplished using digital signatures, HMAC is easier to set up between two parties that share, during some setup phase, a secure channel, and it is generally far more efficient.

4. THE SESSIONLOCK PROTOCOL

At a high level, SessionLock functions as follows:

1. At login time over SSL, the web server delivers a session secret to the web browser.
2. The web browser uses this secret token to authenticate, using HMAC, every subsequent, time-stamped plain HTTP request it makes.
3. The session token is never sent over the network in the clear: it is communicated from the SSL login page to the first plain HTTP page, and to each subsequent plain HTTP page thereafter, using the URL fragment identifier.
4. An attacker limited to eavesdropping capabilities never sees the session secret and cannot generate valid HTTP requests on behalf of another user's session, other than the ones it intercepts.

We now provide additional detail for the above outline.

4.1 Generating the Secret Token

Alice visits her webmail site, example.com. She is directed to a login page over SSL, where she enters her username and password. The server sets up her session, sets a non-SSL session_id cookie, then an SSL-only cookie session_secret, and redirects Alice to

```
http://example.com/login/done#[session_secret]
```

Because this redirect command is sent to Alice's browser over SSL, its content is secure against eavesdropping. Then, when Alice's browser loads the new, non-SSL URL, the session_secret remains secure from eavesdropping, because it is located inside the fragment identifier and thus not sent over the network.

4.2 Keeping the Session Secret Around

To keep the session_secret around from one page to another, it must be appended as a fragment identifier to every URL the user navigates within the web application. Importantly, this cannot be done on the server side, as it would then be available to the eavesdropper when the HTML is transferred over plain, unencrypted HTTP. The appending of the session secret can only be done on the client side using JavaScript.

Thus, upon page load, SessionLock JavaScript code traverses the page, appending the fragment identifier to every clickable link and every form target. Interestingly, in the case of AJAX applications [8], where requests are made in the background without clearing the page's JavaScript scope, it is not necessary to append the session secret to URLs after the first page has loaded, because this first page and its JavaScript scope stay put. In other words, it is easier to use SessionLock with an AJAX application than with more typical page-to-page web navigation.

4.3 Timestamping and HMAC

With the session secret in JavaScript scope, we must then ensure that every HTTP request is augmented with an HMAC. For clickable links and form submissions, a JavaScript event handler intercepts the user request, appends a timestamp parameter, generates the HMAC on the entire request line, and adds a second parameter with this HMAC as its value. Once these modifications are done, the event proceeds as initially requested, only with two new parameters that authenticate the request to the server.

For AJAX requests, JavaScript can intercept all calls to XMLHttpRequest to achieve exactly the same task. In this

[3] http://code.google.com/more/

[4] http://developer.facebook.com

case, the additional parameters are not even reflected in the URL bar, making SessionLock even more transparent. Once again, it appears that SessionLock is easier to implement with AJAX applications.

4.4 Recovering From Failure

Because of our ad-hoc approach to communicating the session secret from one page to another, it is conceivable that the session secret will be lost. The user might type in a URL manually, click a bookmark, or otherwise access the service without the session secret in the fragment identifier. To force the user to re-login at this point would break existing expectations for web services.

Fortunately, it is easy to recover the session secret, using an IFRAME that accesses a small SSL page that minimally affects the user experience. The web page, noticing that it does not have a session secret, opens up an invisible IFRAME with the SSL URL `https://example.com/login/recover`. The document in the IFRAME is tiny:

```
<script language="javascript">
document.location =
  'http://example.com/login/recover#[' +
  get_cookie('session_secret') + ']';
</script>
```

(This code assumes the existence of a `get_secret()` function, which can be implemented in a few lines of code that performs a regular expression match on `document.cookie`.)

This code, which runs in the SSL scope, simply recovers the session secret from the SSL-only cookie, then redirects the browser to the plain HTTP portion of the site with the secret in the fragment identifier. This plain HTTP page, loaded within the IFRAME, can access the secret using `document.location.hash`. Then, since it is now in the same scope as the containing page, it can make a simple procedure call to the parent frame to deliver the token and close the IFRAME. This recovery protocol is diagrammed in Figure 2.

5. EXTENSIONS

The basic SessionLock protocol can be extended to support alternate use cases.

5.1 Optimizing Link Setup

The SessionLock JavaScript that traverses the DOM to add appropriate event handlers is likely one the weakest pieces of the puzzle, where some links may be missed and the time taken to traverse a complicated HTML DOM may be onerous. If the web application is built with SessionLock in mind, then this click handler can be added explicitly in the HTML, only on the links that explicitly need authentication:

```
<a href="next.html"
   onclick="sessionlock_patch(this);">
     next page
</a>
```

This approach will increase the size of the HTML a bit while speeding up the JavaScript execution significantly, since no SessionLock code is executed until the user clicks a links, and even then only a small amount. In addition, with application-level involvement, only links that require authentication will be patched.

5.2 Local Browser Storage

The latest versions of Internet Explorer and Firefox, which together cover about 95% of web users [5], both offer simple mechanisms for client-side data storage that is never automatically sent over the network, respectively `window.userData` and `window.globalStorage`. In addition, the upcoming HTML5 specification [11] standardizes this JavaScript API for client-side, domain-specific data storage along the lines of Firefox's implementation. Safari, the third largest browser, is expected to implement this API, making client-side storage a virtual certainty in the near future. In HTML5, the following JavaScript code stores data:

```
globalStorage['example.com'].session_key =
   '8xk3jsldf';
```

which can later be retrieved by another page from the same domain using the following code:

```
do_stuff_with_key(
   globalStorage['example.com'].session_key
);
```

Local browser storage cannot solve everything on its own: it cannot be used to transfer the session secret from the HTTPS session-setup URL to the HTTP post-login URL, because those two URLs are of different origins, and client-side data stored while at an SSL URL cannot be read by JavaScript from a non-SSL URL, even if they share the same domain.

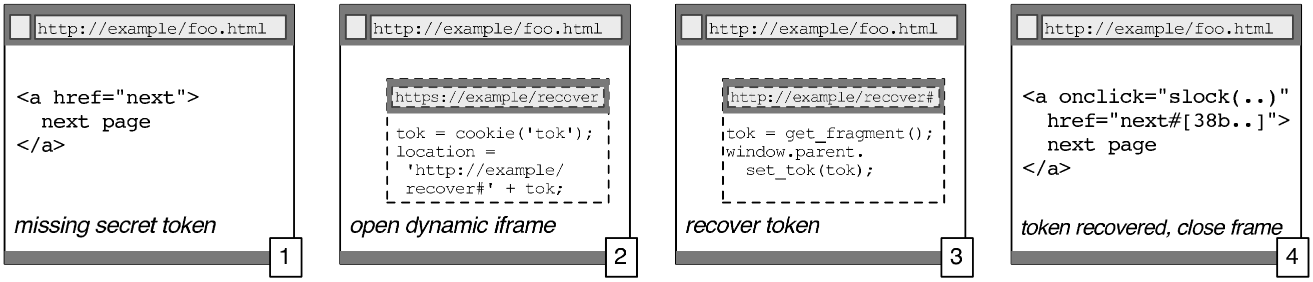
However, once the token is transferred to HTTP using the SessionLock fragment identifier approach, it can be stored in local session storage so that, even if a user subsequently loses the secret token by deleting the fragment or opening up a new browser window, the JavaScript code can easily recover it with simple API call, instead of an IFRAME and additional network access. Augmented with local browser storage, the overhead of SessionLock becomes quite negligible, even in edge cases.

5.3 No SSL Whatsoever

We can implement SessionLock without any SSL, even on the login page. On session setup the following steps are taken:

1. the server assigns the browser a session cookie.
2. the client-side JavaScript code initiates a Diffie-Hellman key-exchange [5] with the server, effectively generating a shared secret between the browser JavaScript scope and the server.
3. this shared secret becomes the HMAC key used in the SessionLock protocol, with the server storing the secret in a server-side session, and the browser passing on the secret from one page to the next using either the fragment identifier or the local-browser storage as explained above.
4. if the browser loses its secret, it can re-perform a Diffie-Hellman key exchange with the server, using a number of `XMLHttpRequest` calls.

[5] `http://en.wikipedia.org/wiki/Usage_share_of_web_browsers`, last visited on February 3rd 2007.



```
<a href="next">
  next page
</a>
```

```
tok = cookie('tok');
location =
  'http://example/
  recover#' + tok;
```

```
tok = get_fragment();
window.parent.
  set_tok(tok);
```

```
<a onclick="slock(..)"
  href="next#[38b..]">
  next page
</a>
```

Figure 2: If the secure token is lost for any reason (1), it can be recovered from the HTTPS cookie using a dynamically generated IFRAME (2), which looks up the cookie and redirects the IFRAME to a non-SSL URL with the token in the fragment identifier (3), which can then pass the token back up to the calling frame (4). In a production implementation, the IFRAME would be made invisible since the it requires no user interaction.

The no-SSL approach is clearly less efficient at recovering from a token loss, since a token loss requires the complete re-generation of a new token, rather than the SSL-based retrieval of the existing token. This indicates that the no-SSL approach probably shouldn't be used unless the browser supports local storage, which significantly curtails the chance of this token loss.

6. EVALUATION

We built a SessionLock prototype, available for demonstration and full source code download in the near future at:

`http://ben.adida.net/projects/sessionlock/`

In this section, we review interesting details of our implementation and the associated performance of our prototype.

6.1 Implementation Details

We use a JavaScript library [13] that implements HMAC-SHA1. Note that, while SHA1 has recently been shown to have certain weaknesses [19], its security in an HMAC setting has not been compromised. If it were to be compromised, a move to SHA256 would be fairly straight-forward and only slightly more computationally intensive.

In addition, we use a JavaScript library [6] that implements robust URL parsing, so that we can dynamically insert the SessionLock parameters into any URL, no matter how complex.

Then, our custom SessionLock JavaScript library implements:

- detection and parsing of the secret token in the fragment identifier,
- recovery of the secure token using an invisible IFRAME,
- timestamp-and-HMAC patches for links and forms,
- `XMLHttpRequest` interception for timestamping and HMAC.
- automatic traversal of the Document Object Model (DOM) to add event handlers to hook up the link-and-form patching.

Hardware and Connectivity. We used a typical shared-hosting provider, using a small portion of a quad-processor Intel Xeon 3.2Ghz server with 4GB of RAM, located in Houston, Texas. We tested Firefox 2.0.1, Safari 2.0.3, and Opera 9 on a Macintosh Powerbook G4 running at 1.5Ghz with 1.5 GB of RAM. We tested Internet Explorer 6 and 7 on Windows XP Professional running on a 1.8Ghz Intel Core Duo with 1 GB of RAM. Both client PCs were connected via a Comcast home broadband connection in Mountain View, California.

6.2 HMAC Performance

We evaluated client- and server-side computational needs for performing HMACs. We determined that, on the slowest browser (Safari) using the specified Mac laptop, an HMAC operation requires just under 50ms. As this is entirely client-side computation, it is negligible and barely noticeable to the user. On the server side, in Python, one HMAC operation took 300μs on our setup, a modest computational requirement compared to the average database query.

6.3 Link Patching Performance

We tested the link-and-form patching overhead on a page with 100 links and 10 forms, and found that the worst browser performance for page setup on the client hardware specified above required no more than 25ms.

We note that the performance overhead on a single-page AJAX application is negligible: only one 15-line function definition is required, no matter how large the page.

6.4 Code Overhead

Our SessionLock-specific JavaScript library is 200 lines of code with copious comments, before any JavaScript minimization. The JavaScript HMAC and URI parsing code together take up less than 7K of code before minimization. On the server side, which we implemented in Python, the login logic required approximately 20 lines of code, and the verification logic about 40 lines of code.

7. DISCUSSION

7.1 Security Model

In this work, we explicitly exclude man-in-the-middle attacks, where the attacker either controls IP routing or DNS,

[6] `http://stevenlevithan.com`

e.g. malicious base stations and routers, DNS poisoning, etc. We assume the attacker can eavesdrop on all network traffic. We do not consider the strength of SSL encryption: our attacker focuses on plaintext traffic and considers encrypted traffic "unbreakable."

Thus, we assume that anything a user browses over plain HTTP can be read by the attacker: if a user is reading their web-based email, the emails she reads are available to the attacker. With SessionLock, our security expectation is that the user cannot be impersonated by the attacker: data not read by the user cannot be read by the attacker, and actions not taken by the user cannot be taken by the attacker on her behalf.

We also consider a slightly stronger attacker who, using social engineering techniques, can trick the user into visiting a particular URL. This may be done using a phishing-like email or instant message. In particular, it is not enough to rely on a user visiting the SSL version of a site if an attacker can trick him into visiting an equally functional plain HTTP version of the same site (without SessionLock protection.)

7.2 Effects of typical web user behavior

With the session secret now a necessary portion of navigation, we must consider the side-effects of carrying this secret as a fragment within every page. In particular, we consider web page reload, bookmarking, and sharing with a friend by copy-and-paste or by posting to a social bookmarking service. We note that these situations should happen rarely on sites that require SessionLock, since the pages that are protected by SessionLock are typically not ones that will be bookmarked or shared with friends. However, it is important to begin to understand how these edge conditions might be handled, even if we don't handle them fully yet in our prototype. We note, again, that none of these potential complications affect AJAX single-page applications like Gmail.

Page Reload. Page reload is explicitly supported by SessionLock: the token stays in the URL as a fragment identifier, and an `onload` JavaScript event handler captures it on reload exactly the way it was captured on first load. However, if a user waits too long on a page that contains a SessionLock timestamp, it may be out of date and fail proper authentication. In this case, a SessionLock web server returns a JavaScript page that locally recreates a freshly timestamped version of the same URL, then uses `document.location.replace` to reload the page with the appropriate authentication. One exception remains: reloading the result of a POSTed form after the timestamp has expired is recoverable only if the user is prompted to manually resubmit the form. This can be accomplished using a JavaScript-triggered click of the back button: `history.go(-1);`

Bookmarking. Bookmarking a page that uses SessionLock will include the timestamp and HMAC at the time of the bookmarking action. When the page is later reloaded, it is almost certain that the timestamp will be outdated. In this case, the SessionLock server can behave exactly as in the page-reload case, issuing JavaScript code that, within the browser, generates a freshly timestamped and HMAC'ed URL.

That said, when a bookmarked page is loaded, the original session from which that page was bookmarked is likely to have expired. In this case, the web server will notice an expired session cookie even before it checks the timestamp and HMAC, and will redirect the user to a login page, which should easily allow the user to log in, create a new session, a new session secret, and a redirect to a newly timestamped and HMACed version of the bookmarked page.

Sending to a friend, social bookmarking. If a user sends a SessionLock-augmented link to a friend via email, or especially if she posts it to a social bookmarking site, she runs the risk of revealing her session secret. If a recipient of this session secret is also on the same local network and can find her plain HTTP `session_id` cookie, the user's session may be fully compromised. Although it is unlikely that a user would post a protected link to a social bookmarking site, this issue merits further careful consideration.

7.3 Deployability

Simple web sites are easily upgraded to SessionLock using our existing JavaScript toolkit that dynamically traverses and appropriately updates links, forms, and AJAX calls. For more complicated web sites, building from the start with SessionLock in mind is relatively straight-forward: using simple abstraction layers can enable the easy patching of links, forms, and AJAX calls, even when they use custom event handlers.

Upgrading an existing, complex web site is a bit trickier and may require a bit of re-factoring: forms and links with existing `onsubmit` or `onclick` event handlers are particularly difficult to patch generically, and will, in most cases, require manual patching.

There is one interesting exception to this rule for "legacy" web applications: AJAX-only applications that function entirely within one URL, sometimes called "single-page applications" are particularly easy to patch for SessionLock, assuming they use `XMLHttpRequest` relatively consistently. In particular, sites like Gmail [7] should be quite easy to upgrade to SessionLock: no HTML forms or links need to be updated, thus no DOM traversal is ever required. A single patch to the AJAX handler suffices.

7.4 Limitations

SessionLock suffers from two important limitations that should be carefully noted.

JavaScript Required. SessionLock is entirely dependent on JavaScript: it simply cannot work without. Thus, SessionLock should be reserved for web applications that already require JavaScript.

No Defense Against Active Attacks. SessionLock does not protect against active network attacks. An active attacker can trivially inject code in a plain HTTP URL, steal the session secret and hijack the session. We make no attempt to "fix" this issue in this work: we are solely trying to address the "sidejacking" attack, which is far too easy to launch without leaving a trace.

[7] `http://gmail.com`

8. RELATED WORK

JavaScript Use of the Fragment Identifier. The fragment identifier has been usurped in other ways, usually as a mechanism to maintain state in a single-page JavaScript web application. S5 [17], an HTML slide presentation tool, uses the fragment identifier to indicate which slide to display, with the whole slideshow contained in a single HTML file. The Dojo JavaScript toolkit [14] and other JavaScript libraries use the fragment identifier to enable the normal forward and back buttons in an AJAX [8] web application without full page reloads. The first use of a fragment identifier for security purposes that we know of is BeamAuth [1], where a bookmark including a secret token in the fragment identifier is used to defend against phishing attacks for web authentication. BeamAuth and SessionLock both use the fragment identifier but can be made to work together, since they use the fragment identifier at very different times.

Security in the Web Application Layer. Others have proposed security protocols that make use of existing browser features in novel ways, effectively building security into the web application layer. Juels et al. [15] propose to use "cache cookies" for security: the browser cache stores secret tokens for two-channel authentication at secure sites, e.g. online banking. Jackson and Wang [12] explore various existing browser features to enable secure cross-domain communication for web mashups. BeamAuth [1] provides some defense against phishing using only existing web features.

9. CONCLUSION

Using the existing HTTP fragment identifier feature to create a secure, client-side channel between HTTPS and HTTP, we have designed and implemented SessionLock, a way to protect plain HTTP sessions from eavesdropping. We believe our proposal is relatively easy to implement, especially in the case of heavily AJAX-enabled applications such as Gmail. In fact, it appears that Gmail HTTP sessions can be secured with minimal web-application-level code and negligible performance overhead.

We note the appeal of solutions, like SessionLock, which use only web-application-level modifications: they can be deployed immediately by web developers. We hope that exploration of improved security features using only the existing Web stack will be informative to the improvement of the web browser as an *extensible platform* for security.

10. REFERENCES

[1] Ben Adida. BeamAuth: Two-Factor Web Authentication with a Bookmark. In *Fourteenth ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[2] Apache Software Foundation. SSL/TLS Strong Encryption FAQ – Apache HTTP Server. http://httpd.apache.org/docs/2.0/ssl/ssl_faq.html#vhosts2 last viewed on 1 November 2007.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): General Syntax, January 2005. http://www.ietf.org/rfc/rfc3986.txt.

[4] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. http://www.ietf.org/rfc/rfc3546.txt.

[5] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transport Protocol – HTTP/1.1, 1999. http://www.ietf.org/rfc/rfc2616.txt.

[7] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication, June 1999. http://www.ietf.org/rfc/rfc2617.txt.

[8] Jesse James Garrett. Ajax: A New Approach to Web Applications, February 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[9] Robert Graham. Sidejacking with Hamster, August 2007. http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html.

[10] R. Canetti H. Krawczyk, M. Bellare. Hmac: Keyed-hashing for message authentication, February 1997. http://tools.ietf.org/html/rfc2104.

[11] Ian Hickson and David Hyatt. Html 5. http://www.w3.org/html/wg/html5/.

[12] Collin Jackson and Helen Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th international conference on World Wide Web (WWW 2007), Banff, Canada*, 2007.

[13] Paul Johnston. A JavaScript implementation of the Secure Hash Algorithm. http://pajhome.org.uk/crypt/md5.

[14] JotSpot. DojoDotBook. http://manual.dojotoolkit.org/WikiHome/DojoDotBook/Book0.

[15] Ari Juels, Markus Jakobsson, and Tom N. Jagatic. Cache cookies for browser authentication (extended abstract). In *S&P*, pages 301–305. IEEE Computer Society, 2006.

[16] Message Authentication Code. http://en.wikipedia.org/wiki/Message_authentication_code.

[17] Eric A. Meyer. S5: A Simple Standards-Based Slide Show System. http://meyerweb.com/eric/tools/s5/, last viewed on October 26th, 2006.

[18] Bill Venners. HTTP Authentication Woes, April 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=155252, last visited on October 31st 2007.

[19] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.