

FragToken: Secure Web Authentication using the Fragment Identifier

Ben Adida
Harvard
33 Oxford Street
Cambridge, MA 02118

3 February 2007

Abstract

While web applications are increasingly used to manage important private data, the application programming environment provided by browsers is heavily constrained. In order to implement new features, including novel authentication schemes, many resort to browser plugins. Unfortunately, browser plugins present two deployment disadvantages: they require significant user involvement at installation time, and they expect the user to completely trust the plugin provider, since the plugin can effectively take over the user’s entire browsing experience.

In this work, we present *FragToken*, a set of novel techniques that retrofit existing token-based authentication protocols into major web browsers *without a plugin*. We exploit the well specified and consistent cross-browser features of the URL fragment identifier—the portion of the URL after the hash (#) character—to inject a secret token into the local page scope. Specifically, we use its two crucial properties: it is not sent over the network, and changing its value does not trigger a page reload. We design and implement two interesting and immediately deployable techniques: secure capability URLs without SSL, and phishing-resistant second-factor authentication with a bookmark.

1 Introduction

Web applications are increasingly used to manage important private data. Unfortunately, built-in browser security features are not always adequate, and developing novel authentication solutions in a Web environment is particularly difficult, as the application programming environment is highly limited (sand-boxed Javascript and HTML). To work around these limitations, many novel security ideas have been implemented as browser plugins.

While plugins provide an ideal avenue to prototype new potential browser features [9, 3, 27], they are a risky proposition for real-world deployment: the plugin architecture varies greatly from one browser to another, users must perform a relatively involved plugin setup process, and plugin authors must ask users for complete trust, as a plugin typically has full control over the user’s browsing experience.

Web application developers would be better served by solutions they can deploy today, using existing browser features. Much like “Web 2.0” [35] has pushed the functionality envelope by wedging new features into existing browser APIs—e.g. with Google Maps, Google Mail, Flickr, etc.—the security community may stand to benefit by finding ways to retrofit cryptographic solutions into existing protocols and agents. We propose to do just that for secret-token-based authentication, using a well-documented and long-standing cross-browser-consistent feature: the URL fragment identifier.

1.1 The Fragment Identifier

The fragment identifier is a long-standing portion of the URI specification [33]. It specifies a secondary resource within the primary resource: typically, it is simply a pointer to a specific chunk of an HTML document. For example, the URL `http://example.org/text.html#paragraph_4` specifies the fragment named `paragraph_4` within the resource designated by `http://example.org/text.html`. When they encounter a fragment identifier, Web browsers typically scroll the viewport to the location marked `paragraph_4`.

The specification states that, when dereferencing a URL, the fragment identifier is only meant for client-side processing: it is never sent over the network. In addition, the major browsers chose the natural implementation route: when a user navigates from one fragment identifier to another within the same primary resource, the page is not reloaded, it is only scrolled appropriately.¹ If no page fragment with the requested identifier exists, all major browsers simply ignore the fragment identifier, though it remains in the address bar.

1.2 Our Contributions

Given its well specified and consistently implemented properties, it is natural to think of the fragment identifier as a *local input to the web page*. Interestingly, the web page's response to this local input can be customized by the web application developer, using only standard Javascript to poll the fragment identifier and react accordingly.

We developed **FragToken**, a set of techniques which use the fragment identifier to inject a secret cryptographic token into the local page state. This secret token can then be used to implement cryptographic authentication protocols. We specifically design and implement two immediately deployable security features: *secure capability URLs without SSL*, and *phishing-resistant second-factor authentication with a bookmark*. We stress that our solutions require only a bit of Javascript and HTML, and can thus be deployed by any web application without browser modifications. Where certain browsers behave in slightly non-standard ways, we provide work-arounds to ensure that our proposals function in Internet Explorer, Firefox, Safari, and Opera.

1.2.1 Secure Capability URLs

Many web applications use URLs with secret tokens to let users return to the web site with automatic authentication. The secret token is large enough and “random enough” so as to be impractical to guess:

```
http://example.org/resource/123?code=AXWBMCMVDEUERCJKQME
```

Effectively, these are “capability URLs” [18], where a long unique string is used both for identifying a resource and authorizing access to it. A user may receive such a URL via email. Alternatively, Alice may be redirected to such a URL after logging into a single sign-on service, e.g. Yahoo's BBauth [41] or an OpenID identity server [6]. The capability URL is typically used to send her back to the web application that originally requested authentication, where the token is proof of her authentication.

We propose to relocate this capability token to the fragment identifier:

```
http://example.org/resource/123#[AXWBMCMVDEUERCJKQME]
```

¹with some exceptions addressed in Section 4.

Overview of the FragToken URL Mechanism. Upon de-referencing of this new *secure capability URL*, the server returns a Javascript program that reads the authentication token from fragment identifier *entirely on the client side*. The Javascript program then engages in a HMAC-based challenge-response protocol with the server to securely prove knowledge of the token. Upon successful completion of this protocol, the server returns the originally requested content. The authentication token is never sent over the network in the clear.

Application: Securing Login Credentials without SSL. Secure capability URLs are not a replacement for SSL: they do not protect against man-in-the-middle attacks. However, they are particularly useful in the numerous cases where SSL cannot be used, in particular for smaller web providers. Even without SSL, they prevent the most common and easiest-to-achieve web-based attacks: URL sniffing of credentials. Our optimized implementation requires only one extra sequential HTTP request with a small payload, typically adding less than a quarter of a second to the page load.

1.2.2 Second-Factor Authentication with a Bookmark

Using the same techniques, we also propose and implement a two-factor web authentication technique to prevent phishing attacks. The second factor is simply a bookmark to a particular web login page with a secret token in the fragment identifier. The bookmark setup may require a special procedure, i.e. an email loopback confirmation. Then, when Alice arrives at her login page, her “login ritual” contains only one extra step: clicking the login bookmark before entering her password. Since the bookmark click can also be used to inject Alice’s username, it may even become an advantageous usability addition. Of course, the instructions on the login page can guide Alice accordingly.

Overview of the Mechanism. If Alice is already at her login page as expected, the bookmark click only appends the fragment identifier to the existing URL, which *does not cause a reload*. Javascript code within the page can then pick up the secret token from the fragment, prompt Alice for her password, and then use both tokens for login, either securely over SSL or using a challenge-response protocol. Since the page does not reload, any page state, e.g. a hidden-input return URL for post-login continuation, can be maintained inside the HTML page without additional server-side state. (Safari and Opera are somewhat limited in this respect: we present work-arounds in Section 4.)

Importantly, if Alice is being phished and is *not at her login page*, the bookmark click will cause the browser to load the real login page. Though Alice may become somewhat confused, she will not be successfully phished. If the phishing site convinces her not to click her bookmark (or she forgets), her password may be compromised, but this is only one of two required tokens for login. In other words, Alice can’t easily be phished: even if she makes a simple mistake, she remains safe.

Application: High-Value Single Sign-On Sites. The overhead of this technique is almost entirely in the procedural setup of the bookmark: Alice must drag and drop a link to her bookmarks toolbar, and a separate bookmark must be maintained for each supporting web site. Thus, this technique is most interesting for high-value sites, like financial institutions or single sign-on systems, including OpenID identity providers [6], where Alice is likely to invest more time (and bookmarks-toolbar real estate) into securing her login. Note that some of these high-value sites already require a browser-specific setup procedure, e.g. Bank of America’s SiteKey [2], which roughly requires the same procedural overhead as our proposal. It should be noted that, whereas secure capability URLs are meant as an alternative when SSL is not available, bookmark-based authentication actually *augments the security* of both non-SSL and SSL-based logins with a second authentication factor.

1.3 Limitations

Javascript Required. FragToken requires Javascript. Non-web-browser clients, RSS readers or command-line applications such as `wget`, cannot make use of our secure capability URLs, as they do not evaluate Javascript on the client and may not even support fragment identifiers. In the case of bookmark-based authentication, the login provider should detect and require Javascript at registration time.

Certain Attacks Still Succeed. A man-in-the-middle attack, using either IP or DNS spoofing, can easily defeat secure capability URLs by replacing the honest server's Javascript with malicious code that simply reads the fragment identifier and sends it to the adversary. If our bookmarks-based authentication is used with a non-SSL login, the same man-in-the-middle vulnerability exists. Malware can defeat our bookmark-based authentication solution, even if it uses SSL.

Adaptation Necessary. As new browser versions are released, our proposed techniques will need to be re-checked, and some work-arounds may need to be implemented depending on specific browser quirks. Such is the fate of most web applications, unfortunately.

1.4 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 [26], an HTML slide presentation tool, uses the fragment to indicate which slide to display, with the whole slideshow contained in a single file. The Dojo Javascript toolkit [22] and other similar projects use the fragment identifier to maintain state information, so that a user may click the forward and back buttons normally in an AJAX [21] web application without full page reloads. As far as we know, fragment identifiers have not yet been used as secure tokens in cryptographic authentication protocols.

Building Security into 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. [23] 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 [20] explore various existing browser features to enable secure cross-domain communication for web mashups. These are powerful tools for web application developers. FragToken aims to achieve the same deployability for secure authentication protocols.

Challenge-Response Protocols for the Web Until recently, Yahoo offered a hashed-password challenge-response login process [31] (Yahoo has since switched to SSL), though their approach did not provide secure capability URLs. HTTP offers a hashed-password authentication protocol—HTTP Digest Auth [13]—which can be made to support certain types of secure capability URLs:

```
http://username:password@hostname/rest/of/url
```

Challenge-response authentication protocols have been implemented in various browser-extension proposals, including notably Dynamic Security Skins [9], which addresses full-fledged username-and-password-based secure authentication using a browser plugin.

Bookmarks for Security. Two recent Internet blog postings [4, 25] suggest the use of a bookmark to prevent OpenID phishing: Alice would first and independently navigate to her OpenID provider by clicking the bookmark, rather than allowing the site to redirect her to her OpenID server. Though these are very interesting and useful suggestions, our technique is a bit different: we do not attempt to change the way that existing single sign-on protocols function, only how Alice enters her credentials. We also use the bookmark as more than a server locator: our bookmark serves as a second authentication factor.

1.5 This Paper

We present the core `FragToken` technique in Section 2. Section 3 describes secure capability URLs, and Section 4 describes bookmark-based authentication. Implementation details and performance metrics are described in Section 5. Threats, defenses, and potential impact are discussed in Section 6.

2 FragToken Basics

The core `FragToken` features are quite simple: a secret cryptographic token is “injected” into the local page scope via the fragment identifier, a portion of the URL never sent over the network but accessible from Javascript code. In this section, we present these issues in detail, especially their specific implementation differences across the four major browsers: Internet Explorer (6 and 7), Firefox (1.5 and up), Safari (2.0.3 and up), and Opera (8 and 9).

2.1 The URL Fragment Identifier

A URL [33] may contain a fragment identifier, which, as its name implies, addresses a fragment of the resource denoted by the fragment-free URL. Specifically, a URL with a fragment identifier looks like:

```
http://hostname/rest/of/url#fragment_id
```

The resolution of a fragment identifier within a given document depends on the document’s MIME type. In the case of an HTML document, with MIME type `text/html`, the fragment identifier specifies a portion of the HTML page identified accordingly. Importantly, a web browser will resolve the above URL as follows:

1. connect to host `hostname` on port 80,
2. request `/rest/of/url`,
3. scroll the viewport to the position indicated by `fragment_id` if it exists, ignore it otherwise.

Note how `fragment_id` is never sent over the network. This property has been confirmed on all major browsers, and it is, in fact, part of the URI specification.

Navigation. When a user navigates from one fragment identifier to another within the same primary portion of the URL, the browser *does not trigger a page reload*: the page simply scrolls to the position indicated by the new fragment identifier (or, again, does nothing if no such position exists). In the two dominant browsers, Internet Explorer and Firefox, this absence of reload remains true no matter how the initial URL was loaded, and no matter what the cache preferences on the downloaded page might be: even a page with strict no-cache HTTP headers that results from a `POST` operation is not reloaded when only the fragment identifier changes. Thus, all page state, be it local Javascript variables or HTML form inputs entered by the user, remains unaffected when the fragment identifier changes.

There are exceptions to these otherwise mostly consistent rules. In Opera, changing the fragment identifier on a page that results from a `POST` does, in fact, trigger a reload, this time as a `GET`. In Safari, a change in fragment identifier triggered by an external source, e.g. clicking a bookmark or manually entering a new fragment identifier rather than clicking a link within the web page itself, does trigger a reload even if the primary URL does not change. These two browsers make up less than 5% of web users [39]: it is reasonable to special-case their support with extra server-side overhead.

2.2 Javascript Features

Javascript code can perform a number of tasks within the user's browser.

Fragment Identifier and URL Manipulation. In all browsers, `window.location.hash` is a read/write Javascript variable that corresponds to the fragment identifier as it appears in the browser's address bar. Changing the value of this variable updates the address bar without reloading the page, scrolls the viewport to the appropriate location (if it exists), and results in the new URL being added to the browser's history.

When we want to change the value of this fragment identifier without leaving a trace, for example to clear a secret token from view, we will use a slightly different mechanism. The Javascript function `window.location.replace()` updates the URL (including the fragment identifier) without adding the previous URL to the history. It is as if the previous URL was never visited.

Javascript Bookmarks and Overriding Javascript Default. It is tempting, for our purposes, to use a Javascript bookmark, also known as a bookmarklet, which is effectively a small piece of Javascript code that is executed when the bookmark is clicked. Some early prototypes of this work were implemented accordingly. Unfortunately, this code cannot be expected to behave correctly, because the bookmark's Javascript is executed in the context of the current page. A malicious page might override any Javascript command, even the standard Javascript API, thereby completely altering the behavior of the bookmark code. (Some variables are declared `constant` in the Javascript specification, but most browsers do not respect these restrictions.)

2.3 Web Security

Two core security features of HTTP are available to web application developers: HTTP authentication [12] for managing logins, and SSL/TLS [1, 34] for securing HTTP content and authenticating HTTP servers (and sometimes clients). We briefly review the types of attacks that web users most often face and how current HTTP security features address them.

Types of Attacks. Web security threats generally belong in three categories:

1. **passive sniffing:** users often access web sites over open or insecure wi-fi access points, unswitched local wired networks, or corporate proxies. The URLs they request and the content they receive are easily sniffable when SSL is not used. The damage from these kinds of attacks is unclear, as most non-SSL-using web sites are small providers. However, the threat is clear: while the W3C does not mandate SSL, they specify that login credentials should never be sent in the clear [30].
2. **social engineering:** users are easily fooled by malicious sites that spoof legitimate sites to steal credentials. Financial institutions are the typical target, and users generally don't check the URL or even the SSL padlock of their connections [8]. The damage from these attacks is well documented and significant [16], and carrying out such an attack is fairly trivial.

3. **desktop compromise:** a surprisingly high number of desktop computers are compromised with malware [24]. Users of these compromised machines have zero guarantee of any security: all security indicators may be faked, and all hostnames may be hijacked. Damage from these attacks is significant, though carrying out such an attack is quite involved.

HTTP Authentication. HTTP authentication [12] offers a challenge-response protocol which fulfills the W3C recommendation: Digest Mode. However, it is used quite sparingly on the public Web, because browsers do not provide a mechanism for application developers to customize the user experience, and because the server components must engage in an HTTP-specific challenge-response protocol, which is not always easily supported in development environments. It should also be noted that browsers and back-end programming environments generally do not obtain a clear indication that Digest Mode is indeed activated.

Instead of using the built-in HTTP Authentication in Digest Mode, Yahoo effectively re-implemented the same features in Javascript [31]. The lesson seems to be that web developers want to handle authentication in an end-to-end manner, not as an abstracted layer deep within the protocol stack.

When SSL is too much. SSL encrypts the content of an HTTP exchange and authenticates the server (and sometimes the client.) For a number of web applications, this is the right security solution: the content should be secured. However, for many smaller web applications, SSL may be difficult to implement. SSL requires a certificate, which can cost too much for small operations. In addition SSL cannot be configured when multiple domains are virtually hosted on the same IP address: the SSL negotiation happens before the web server gets the chance to send the `Host` HTTP header [10]. This situation occurs rather often at small-usage hosting providers, e.g. at blogging sites where users share IP addresses but are able to virtual-host the domain of their choice [32]. Thus, in a number of use cases, SSL is simply not an option.

When SSL is not enough. On the flip side, for high-value applications, SSL may not be enough. In particular, SSL does not prevent social engineering attacks, as evidenced by the depressingly high success of phishing attacks. The key issue is that, even with SSL, the web remains treacherous: a momentary lapse in judgment can result in a successful credential theft. As a result, many suggest that high-value sites resort to two-factor authentication, where at least one factor is not easily stolen from an inattentive user.

2.4 Goals of Our Proposals

We aim to prevent the “easy attacks” on real-world web sites: passive sniffing of small web site credentials, and social engineering of high-value web site customers. Small web applications should be able to secure at least their login credentials in every use case, even if they cannot use SSL. High-value web sites should have an easy way to implement two-factor authentication without resorting to browser plugins or physical tokens for users to carry around at all times.

3 Secure Capability URLs without SSL

When SSL is not an option, we already know how to secure a typical username and password login system: use the Yahoo challenge-response system. Here, we show how to use the `FragToken` technique to accomplish the same level of security for capability URLs. Recall that a capability URL is one that identifies a resource and includes an authorization token for that resource at the same time: resolving the capability URL should be enough to locate the resource and authenticate appropriately.

Server-Side Features We assume the server-side software has the ability to maintain simple user sessions across HTTP requests. Such sessions can be implemented using cookies or tokens within the URL (*not* in the fragment identifier). This session does not need to survive across browser restart or cookie resets, but it should survive through a few requests on the site, and remain valid for a few minutes. Note that all known web application environments support such a feature, usually using short-lived cookies, long unique unguessable session identifiers, and server-side storage of session data.

3.1 Use Cases

Secure capability URLs are useful in two major settings:

- **Links by Email:** web-based applications often contact their users via email, including a link to click in order to perform some action. The purpose of such an email may be an invitation to view content made available by another user, e.g. an Evite or a Google calendar.

Secure access to email content is fairly common, with POP3 or IMAP over SSL². Thus, using secure capability URLs, a web-based service can deliver a URL with built-in authentication, where the authentication token resists passive sniffing.

- **Third-Party Authentication:** Yahoo has recently begun offering third-party, single sign-on capabilities [41]: Alice can log in to `MyBlog.com` using her Yahoo account, assuming `MyBlog.com` has set up a secure relationship with Yahoo. In this protocol, `MyBlog.com` sends Alice to Yahoo, where she logs in. Yahoo then redirects her to a URL at `MyBlog.com` with an embedded security token. Yahoo provides this return-URL over SSL. If Yahoo uses a secure capability URL, the authentication token remains secure from passive sniffing, even if `MyBlog.com` does not implement SSL.

At a high level, secure capability URLs are useful when transferring authentication tokens from an encrypted channel to a plaintext channel. Secure capability URLs utilize the confidentiality of the first channel to secure authentication over the second channel.

3.2 The Secure Capability URL Protocol

A secure capability URL is simply a URL with a secret token as its fragment identifier:

```
http://hostname/resource#[secure_token]
```

Using the `FragToken` technique, the page's local Javascript extracts the secret token from the fragment identifier and engages in a challenge-response protocol with the server to prove knowledge of this secret. The steps of the protocol are diagrammed in Figure 1 and are detailed below:

1. Alice receives a secure capability URL

```
http://hostname/resource#[secure_token]
```

via some secure channel and navigates to it.

2. The web server receives a request for

```
http://hostname/resource
```

(Note that this does not contain the token, which is part of the fragment identifier.)

²We worry less about securing mail-server-to-mail-server communication, as that link is far less vulnerable than the last mile. If this link does need to be secured, it can easily be done using SMTP/SSL [19].

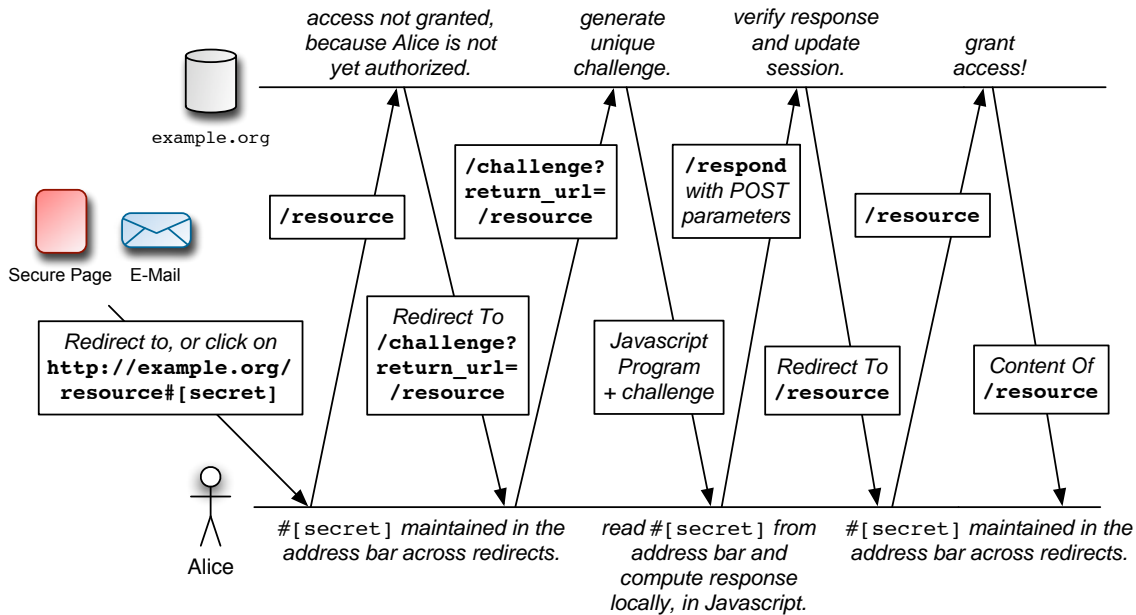


Figure 1: The Secure Capability URL Protocol. The protocol proceeds from left to right between Alice and the `example.org` web server. Alice receives a secure capability URL from some secure channel, either email or web over SSL. When accessing this URL at `example.org`, the secure token is *not* sent over the network, as it is contained within the fragment identifier. The `example.org` server returns a redirect to the challenge page, which Alice’s browser loads, giving her a Javascript program and unique challenge. The Javascript program runs automatically within Alice’s browser, accessing the secure token in her address bar, computing the resulting response to the challenge, and submitting this response to the web server. Upon checking this response, the server authorizes Alice’s session and redirects her to the original resource, where her request is now authorized.

3. The web server checks if Alice’s session is authorized to access the given resource. If so, it serves the resource, and the protocol is done. On a first click, though, this should not happen: Alice is not yet authorized. In that case, the server uses an HTTP code 302 response, redirecting Alice to

```
http://hostname/login/challenge?return_url=/resource
```

Note again how the security token is not part of this redirect on the server side.

4. On the browser side, the fragment identifier is preserved across the redirect, as per the HTTP specification [29], and the URL in the browser bar is now:

```
http://hostname/login/challenge?return_url=/resource#[secure.token]
```

5. The server generates a random challenge, stores it in Alice’s server-side session, and delivers it to Alice, along with the HTML and Javascript code necessary to perform the client-side computation of a HMAC-challenge-response protocol.

6. Alice’s web browser runs this code, which *locally* accesses the secure token using the Javascript call `window.location.hash`, then computes the proper response to the given challenge, and performs an HTTP POST to

```
http://hostname/login/respond#[secure.token]
```

with parameters

- response, the computed response and

- `return_url`, the intended destination, in our case `/resource`.

Note how, again, `secure_token` is not sent over the network.

7. The web server checks the response and, if it is correct, sets the proper server-side session information to note that Alice is now authorized to access the original URL, then redirects to the `return_url`, in this case `/resource`.

Thus, the sequence of URLs accessed by the browser is:

- `/resource#[secure_token]`
- `/login/challenge?return_url=/resource#[secure_token]`
- `/login/respond#[secure_token]` with additional POST parameters
- `/resource#[secure_token]`

The secure token remains in the URL the whole time but is never sent over the network. Because the redirects happen fairly quickly, Alice may not even notice anything other than the original URL, `http://hostname/resource#[secure_token]`. Note also how Javascript is used quite minimally: only bit-level operations for HMAC purposes, a single access to the address bar, and a redirection.

Use of Secure Capability URLs. Secure capability URLs are self-contained and secure: they can be bookmarked or sent to a friend, all the while retaining their complete functionality. Yet the token cannot be recovered by a passive adversary, even if the URL uses only plain, unencrypted HTTP. In addition, because of the special redirects, the secure capability protocol maintains expected user behavior: the intermediary steps are not recorded in the browser history, and a single click of the back button will take Alice to the logical previous page.

Getting Around Non-Compliance. Of the four major browsers, only Mozilla/Firefox and Opera carry through the fragment identifier appropriately when a server issues an HTTP redirect. To get around the non-compliance of Internet Explorer and Safari, a simple tweak in step 3 can be used: instead of an HTTP redirect, the server sends a small Javascript program that manually reads the fragment identifier inside the browser, then uses the call `window.location.replace()` to send the browser to the appropriate URL. This recreates a proper, fragment-identifier-preserving redirect, as per the HTTP specification. Like an HTTP redirect, this Javascript call does not affect the browser history.

Graceful Degradation. The single-sign-on use case is particularly interesting, because the identity provider can detect the user's browser's capabilities and act accordingly. Thus, if Alice's browser does not have Javascript enabled, the identity provider can back out to the normal, less secure, token-embedding technique. Here, secure capability URLs degrade gracefully.

3.3 Variations on the Secure Capability URL Theme

We now present a few variations of secure capability URLs for different programmatic or security needs.

Optimization. The protocol as described above requires 3 additional HTTP requests. While these requests carry a very small payload, a low-latency connection might suffer because the 3 requests must be sequential. It is fairly straight-forward to optimize this process down to only 1 additional HTTP request:

1. Send the challenge and the HMAC code in response to the very first request.
2. Respond to the challenge using the same original resource URL, with an extra URL parameter `response`.
3. Send the protected content in response to this second request.

Single-Page Functionality. Using the `XMLHttpRequest` Javascript feature [21], all of the challenge and response steps of the protocol, before or after optimization, can be performed without a complete page reload, with HTTP GETs and POSTs performed in the background. The browser stays at the same URL the entire time, displaying a message for the user indicating that authentication is under way. When the actual content is finally obtained, the Javascript program can dynamically update the page's HTML.

Keeping Tokens out of the Browser History. In scenarios like single sign-on, keeping the secure token in the browser's history may work against the security goals of the application. As soon as the secret is read from the fragment identifier into a local Javascript variable, the Javascript code can use the `window.location.replace()` call to update the URL and remove the fragment identifier. Thus, the browser history will not record the URL containing the secret token.

Encrypted Content. One might want to protect not just the access control token, but the content, too. Of course, in such cases, SSL is likely the right solution. However, if one wishes to keep the content encrypted even on the server, decryption on the client side is an interesting option. Using our solution, the secure token can serve as a decryption key, where the local Javascript code performs in-browser decryption and dynamically updates the HTML with the resulting plaintext. Note that an AES Javascript implementation already exists [14].

Combined with the history-clearing redirect trick from the previous paragraph, it is possible to direct a user to a URL that immediately decrypts the content, then clears the decryption key from the browser history. In other words, the content is decrypted long enough to be displayed in the browser window. Once the user navigates to a new page, the encrypted content is no longer available, and the key has also been cleared from the browser history.

4 Two-Factor Authentication with a Bookmark

Phishing attacks have had a significant negative impact on high-value online web applications, particularly financial institutions [16]. The situation is urgent enough that radical new browser extensions are being developed, including Microsoft's CardSpace [5], to provide a more secure authentication experience. While these solutions are extremely promising, their deployment will take quite some time, as they require a significant browser upgrade.

4.1 Use Case

We consider specifically the single-sign-on use case in its many forms, where Alice is sent to her login page by a third-party web site. For example, Flickr sends its users to Yahoo for authentication, and any web application can use Yahoo in the same way with Yahoo BBauth [41]. A growing number of web

applications use OpenID [6] for authentication, where the third-party web site is expected to redirect Alice to her OpenID server. A number of university networks also use this same technique: Harvard University’s PIN system [36] and Stanford’s WebLogin system [37] are two prominent examples, where peripheral sites send users to the central login site which, after authentication, redirect the users back to the peripheral site with an authentication token.

In all of these cases, phishing is of great concern, since Alice is sent to her login page *by the site requesting authentication*. It has been noted in particular that OpenID may make phishing easier because Alice explicitly discloses her identity provider to a potentially evil site [4, 25]. We aim to mitigate phishing attacks in this widespread scenario, using a bookmark as a second authentication factor.

4.2 The FragToken bookmark

We propose to use FragToken to transform a typical browser bookmark into a second factor for web-based authentication, simply by using a secure capability URL:

```
http://site.com/login#[username|secret_token]
```

The goal is to send the browser to `http://site.com/login#[username|secret_token]` such that, if Alice is already at `http://site.com/login`, the bookmark click does not trigger a reload.

The User Login Ritual. When Alice is presented with a login page where she is already set up to use FragToken bookmark-based authentication, she follows a login ritual only slightly more complicated than the typical username/password process:

1. The web site prompts Alice: “click your FragToken bookmark Login Bookmark.”
2. Alice clicks her bookmark, which updates the login page with her username, and the page now prompts her for her password.
3. Alice enters her password and clicks “Submit.”
4. If both the bookmark token and the password are correct, Alice is correctly logged in.

Setting up the Bookmark. To set up the FragToken bookmark within a browser, Alice must follow some kind of initial authentication process that is inherently more involved than the everyday login. This should be done using a second-channel authentication mechanism, using, for example, a cell phone SMS [40], or an email mailback [15]. Many web sites already perform this kind of verification to ensure that the user’s email address is correct. The mailback can then include a verification URL that is only ever sent by email.

When Alice clicks on this verification URL, she obtains a link that she can easily drag and drop onto her bookmarks/favorites toolbar. Of course, the verification link sent via email should be secure in authentication *and in content*: the second-factor bookmark should never be sent in the clear. This can be done using SSL:

```
https://site.com/confirm?vc={verification_code}
```

or it can be achieved using a simplified secure capability URL using our fragment identifier technique:

```
http://site.com/get-bookmark#[username|secret_token]
```

where the server responds with a template for the bookmark and a piece of Javascript code that fills in the template on the client side using the secret token in the fragment identifier.

4.3 The FragToken bookmark Mechanism

Normal Operation. Behind the scenes, the login page contains the usual FragToken Javascript that regularly polls the value of the fragment identifier (entirely locally, causing neither network activity nor server-side processing). When Alice clicks her bookmark, the URL fragment identifier is updated, but the page does not reload: instead, the FragToken Javascript poller reads the token `[username|secret_token]` from the fragment identifier. It then fills in the login form with Alice's username, and saves the secret token into a local variable. Optionally, the Javascript can clear the fragment identifier so that the secret token is no longer visible in the URL address bar.

When Alice submits the form with her password, the FragToken Javascript code intercepts the form submit, HMAC's the password with the secret token, and securely submits this resulting credential to the server. Security of this transfer is ensured either via SSL or, if SSL is not available, using the usual challenge-response approach implemented by Yahoo and our secure capability URL code. In this latter case, the value sent over the network is thus:

$$\text{hmac}_{challenge} \left(\text{hmac}_{secret_token}(\text{password}) \right)$$

where *challenge* is provided by the server in the login form. Importantly, the server need only store $\text{hmac}_{secret_token}(\text{password})$, never the password in the clear. The server may also store *secret_token* if it wants to let Alice regenerate a bookmark in the future without invalidating her other already installed bookmarks (e.g. on her other computers.)

Phishing Attack Operation. If Alice is being phished and does not realize it, she may or may not remember to click her bookmark. If she forgets, the attacker may successfully steal her password. Fortunately, this password is not enough to let the attacker log in as Alice, and, since Alice does not actually *know* the value of the secret token hidden in her bookmark, it is unlikely that she can be tricked into revealing it.

If Alice remembers to click her bookmark, her browser will be sent to the actual login page. The legitimate login server will then log her in normally, notifying her that, since she clicked her bookmark at a remote site, there is no return URL to send her to. It may have been Alice's intention to log in independently of any third-party request, for example at the beginning of the day. Alternatively, Alice may realize she was initially being phished. In any case, even if she isn't paying attention, Alice never surrenders both of her authentication factors to the adversary.

4.4 Extension: Secret Token for Further Data Access

The identity provider may offer more than shared-secret authentication services: both Yahoo and Google offer additional data services that third parties may access on behalf of Alice, if she allows it. Usually, the identity provider accomplishes this by returning a secret token which, by way of redirect within Alice's browser, is sent to the third-party site and can later be used to request more information directly from the identity provider.

In this case, the token is not a shared secret between the identity provider and the third-party site: its actual value must be delivered to the third party. Using FragToken techniques, this can *still* be accomplished securely, even if the third party cannot use SSL. We use a variant of the Javascript-encryption trick: the third-party site and identity provider already have a shared secret, from which they can easily generate a unique AES secret-key for that session—e.g. by HMAC'ing a timestamp-dependent nonce which can be sent in the clear from the identity provider to the third-party site (via redirect). Alice's web client can then encrypt the token it received securely from the identity provider using this AES key, and send it back to the third-party encrypted accordingly.

4.5 Limitations

The URL of the login page must match exactly what the bookmark expects, otherwise a reload will be triggered. If the login page needs certain parameters, they should thus be sent via `POST`.

Unfortunately, Safari and Opera do not fully support this approach. In Opera, the login page must be loaded via a `GET` operation if the local secret-token injection is to succeed without a page reload. In Safari, the page will *always* reloads. Thus, for both of these web browsers, the login server should store parameter information regarding the user's return-URL in a server-side session, so that a page reload will not mistakenly delete them. Note also that, if the page is reloaded and the login server uses SSL, the bookmark need not use a secure capability URL: it can simply embed the token in the usual way. This work-around requires more server-side state. As Safari and Opera together make up less than 5% of the browsing public, this compromise is likely reasonable.

5 Implementation

In this section, we describe our implementation of the various `FragToken` techniques. All functionality, including performance measurement code, is available for live demonstration at

<http://labs.adida.net/fragtoken/>

5.1 Setup

Hosted Server and Web Client. To test all of our non-SSL techniques, we used a typical shared-hosting provider, using a small portion of a quad-processor Intel Xeon 3.2Ghz server with 4GB of RAM, co-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 Boston, Massachusetts.

Web Server and Application Logic. We use Python 2.4 [17] as the backend programming language, with the CherryPy web environment [7] that simply maps URLs to Python methods. The resulting code should be fairly easy to read even if one is not entirely familiar with Python or CherryPy. We use the Apache [11] web server to handle all HTTP requests, with a `mod_proxy` interface to bridge Apache and CherryPy. We built the server-side `FragToken` features using the built-in CherryPy session support and the built-in Python HMAC API. The back-end code for secure capability URLs, including both the basic and optimized versions, contains less than 100 lines of Python plus a few HTML templates, while the back-end code for `FragToken` bookmark, including the mailback implementation, contains approximately 200 lines of code plus a few HTML templates.

Javascript. We use a Javascript library [28] that implements HMAC-SHA1. Note that, while SHA1 has recently been shown to have certain weaknesses [38], 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. We wrote a small Javascript library to implement the various `FragToken` features:

- polling and reading the fragment identifier,
- performing a `window.location.replace()`-based redirect,

- performing the `FragToken` bookmark login process, including UI updates and double HMAC.

Our `FragToken`-specific Javascript is less than 50 lines of code (not counting the HMAC library).

```
SCURL.getToken = function() {
    var hash = window.location.hash;
    // take out the '#' sign and the square brackets
    return hash.substring(2, hash.length-1);
};

SCURL.computeResponse = function(challenge) {
    return hmac(SCURL.getToken(), challenge);
};
```

Figure 2: The challenge-response protocol in Javascript: the fragment identifier is extracted and parsed, then the HMAC is taken on the challenge, using the secret as the HMAC key.

5.2 Performance

We evaluated client-side computational needs for performing HMACs, which are used in both `FragToken` URLs and bookmarks. We also evaluated the network overhead of the additional requests required in implementing `FragToken` URLs. We provide all of these testing scripts on our live server, so that anyone may re-perform these tests using their own machine, browser, and connectivity.

HMAC in Javascript. On our test machine, we used a Javascript loop to compute 500 HMAC operations. We used the Javascript `Date` object to time the computation. On the Mac, Opera computed one HMAC in 17ms, Firefox in 19ms, and Safari in 49ms. On the PC (whose processor is faster than the Mac we used), IE computed one HMAC in 9ms. In the case of `FragToken` bookmark-based authentication, two HMACs are necessary, bringing the slowest client-side processing time—on Safari—to 98ms. Though not entirely negligible, this is likely not noticeable by the average user. Note also that this is client-only computation.

HMAC in Python. On our shared hosted server, we ran a Python loop to compute 10,000 HMAC operations, using the Python `timing` module for timing. One HMAC operation took $300\mu\text{s}$, a relatively modest CPU requirement compared to the average database query.

Network Overhead of `FragToken` URL. To determine the network overhead of `FragToken` URL, we prepared a Javascript program that loads, in an `IFrame`, 100 capability URLs one after the other, using either insecure capability URLs with the token sent in the clear, secure capability URLs, or optimized secure capability URLs. The content page for each capability URL was set up with 3 images of different sizes (76K, 15K, and 11K), to mimick the typical web page (3 items and a total of less than 100K is likely a conservative estimate for measuring the relative overhead of our technique). We used a number of cache-busting techniques to correctly measure multiple page loads as if each one were the first:

1. the Javascript `FragToken` URL code, including HMAC, is included inline each time,
2. each request includes a cache-busting parameter set to the current time in milliseconds, and
3. the images on the final content page also included a cache-busting parameter.

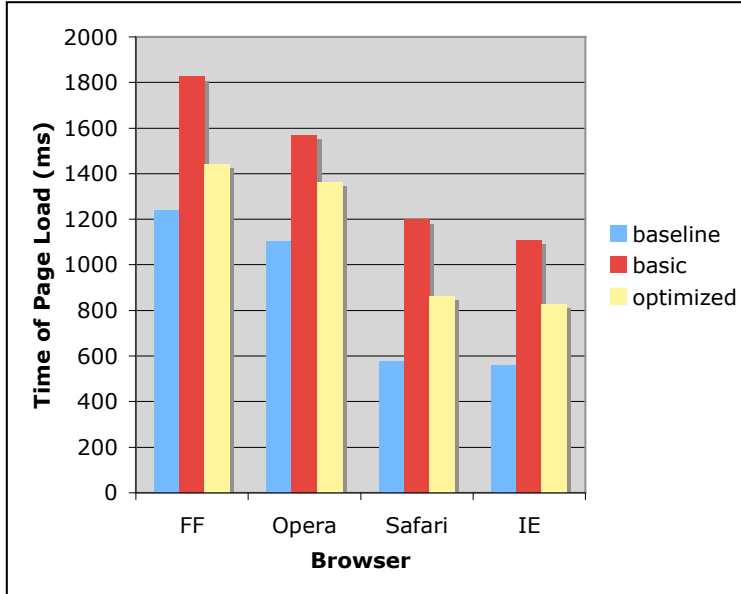


Figure 3: FragToken URL Performance: we use a fairly small page, less than 100K in 4 files, for conservative comparison. The overhead appears to be additive. The baseline is the page loaded using an insecure token embedded directly into the URL. The basic implementation uses three extra HTTP requests, while the optimized uses only one extra request compared to the baseline. Safari’s poor HMAC performance in relation to other browsers is slightly noticeable on this graph.

The basic FragToken URL client-side network-and-computation overhead is diagrammed in Figure 3. In our tests, the overhead of the basic implementation ranged between 465ms and 620ms over the insecure embedded-token URL, while the overhead of the optimized implementation between 201ms and 270ms. The overhead was fairly uniform even when the baseline time varied by a factor of 2.5 from the slowest to the fastest browser, confirming our model that the FragToken URL overhead is additive, as is to be expected from sequential HTTP requests with small payloads and little to no rendering.

6 Discussion

6.1 Threat Model

We’ve assumed that the “low-hanging fruit” of improving web security lies in defending against adversaries that are trivial to implement:

- passive network sniffing to gain long-lived authentication tokens, followed potentially by simple spoofed HTTP requests using this passively gained information.
- active social engineering of web sites, including notably phishing attacks.

We specifically point out that we do not try to defend against the more involved attacks, including malware that effectively turns a user’s machine against him, or DNS/IP spoofing web sites that cannot use SSL.

We believe that on these low-hanging threats already provides significantly improved security for everyday web applications: it is far too easy to sniff unencrypted content and URLs on wi-fi networks, non-switched end-user networks, or at the proxy server level. It is also far too easy to perform phishing attacks,

and there are no current, workable solutions to this problem that do not involve a browser extension. Our solutions are meant to close these large holes in web security for everyday web developers.

6.2 Considering Specific Attacks

Secure Capability URLs. The intent of secure capability URLs is to protect the user's long-lasting credentials at a small, non-SSL web site when clicking on a capability URL. In our proposed solution, an attacker might obtain the secret token in two ways: injecting malicious Javascript code in the HTTP response that captures the secret token, or reverse-engineering the token by passively sniffing the network traffic. In the former case, the adversary must be able to actively spoof an IP address or a DNS host, which we assume is not the case against the small web site. In the latter case, the adversary would have to violate the security property of the HMAC algorithm, which we assume is beyond the adversary's capabilities.

An adversary might simply hijack the session identifier. Tying the session to an IP address helps a bit, but the attacker may well be within the same internal network with the same external IP address, e.g. if the attacker and the victim are both attending a conference with wireless access. In this case, the adversary is able to hijack a short-lived authentication token: the web application should enforce that such a short-lived token not be transformable into a long-lived token. That said, there remains a weakness here that should be further explored in order to fully benefit from secure capability URLs.

Bookmark-based Authentication. In FragToken bookmark-based authentication, an attacker that wants to steal Alice's credentials must obtain both her password and her secret token embedded in her bookmark. A social-engineering attack is unlikely to succeed, as Alice does not know her own secret token and thus cannot type it in by mistake, even if she is fooled by the web page. If Alice clicks her bookmark on a page that is not her expected login page, all modern browsers will send her to her expected login page, away from the attacker. If Alice forgets to click her bookmark, the adversary will only obtain her password, never her secret token.

A passive network adversary might see a challenge-response exchange if the web site does not use SSL. In that case, he can obtain the secret token only if he can break the security of the HMAC (this is true even if the adversary has already performed a social engineering attack to extract Alice's plaintext password.)

Importantly, an active network adversary can be mostly defeated by a combination of SSL and FragToken bookmark. The bookmark directs Alice to her login page at an SSL URL, and, if the active adversary spoofs the IP address or DNS name, the browser will warn Alice that the certificate does not match. In browsers like Internet Explorer 7, the user won't even be allowed to see the resulting page.

6.3 Security in the Web Application Stack

With browsers installed on hundreds of millions of computers, and browser upgrades a fairly rare occurrence with typically conservative goals, it may become increasingly useful to think about implementing security in the web application stack, where the web site developer can innovate. It will be interesting to think about what small changes can be made to the browser platform to enable more innovation in the web application stack, so that the browser need not commit to one security solution, only to becoming a better platform for additional security.

6.4 Impact

Single sign-on is a growing use case which stands to benefit the most from our proposals. With FragToken bookmark-based authentication, OpenID identity providers, Yahoo BBAuth, and single sign-on providers can significantly reduce the threat of phishing without deploying client-side software. In addition, given

the many small web applications that are beginning to rely on these identity providers, FragToken URLs can provide a more secure mechanism for transferring the secure token from an SSL identity provider to a non-SSL relying party.

7 Conclusion

Using only existing features of HTTP and modern web browsers, we've designed and implemented FragToken, a set of secure authentication techniques for various degrees of security. For small web sites that cannot implement SSL, we propose secure capability URLs. For high-value web sites that aim to fight phishing attacks, in particular single sign-on sites, we proposed FragToken bookmark-based authentication, which achieves two-factor authentication without a browser upgrade. All of these features exploit the URL Fragment Identifier and its unusual properties: it is never sent over the network, and changing it does not trigger a page reload.

More generally, we suspect that the web platform is now generative enough for application-layer security in Javascript and HTML. The flexibility of this approach is particularly enticing: new security features can be tested and deployed rapidly, on a per-web-application basis. It will be interesting to see if other existing features can be cajoled to yield additional security properties.

References

- [1] Alan O. Freier and Philip Karlton and Paul C. Kocher. The SSL Protocol, Version 3.0, November 1996. <http://wp.netscape.com/eng/ssl3/draft302.txt>.
- [2] Bank Of America. Sitekey. <http://www.bankofamerica.com/privacy/sitekey/>.
- [3] Blake Ross and Collin Jackson and Nicholas Miyake and Dan Boneh and John C. Mitchell. Stronger password authentication using browser extensions. In P. McDaniel, editor, *14th USENIX Security Symposium*, 2005.
- [4] Kim Cameron. As simple as possible – but no simpler. <http://www.identityblog.com/?p=649>, last visited on February 3rd 2007.
- [5] Kim Cameron and Michael B. Jones. Design rationale behind the identity metasytem architecture, 2006. http://www.identityblog.com/wp-content/resources/design_rationale.pdf.
- [6] D. Recordon and B. Fitzpatrick. OpenID Authentication 1.1, May 2006.
- [7] Remi Delon. Cherry py http framework. <http://cherry.py.org>, last viewed on February 3rd 2007.
- [8] Rachna Dhamija, Doug Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM Special Interest Group on Computer-Human Interaction, January 2006.
- [9] Rachna Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, New York, NY, USA, 2005. ACM Press.

- [10] Apache Foundation. Ssl/tls faq for the apache web server. http://httpd.apache.org/docs/2.0/ssl/ssl_faq.html#vhosts.
- [11] Apache Software Foundation. Apache http server project. <http://httpd.apache.org>, last viewed on February 3rd 2007.
- [12] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. <http://www.ietf.org/rfc/rfc2617.txt>, last viewed on February 3rd 2007.
- [13] 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>.
- [14] Fritz Schneider. Rijndael in Javascript, 2002. <http://www-cse.ucsd.edu/~fritz/rijndael.html>.
- [15] Simson L. Garfinkel. Email-Based Identification and Authentication: An Alternative to PKI? *IEEE Security & Privacy*, 1(6):20–26, November 2003.
- [16] Anti-Phishing Working Group. Phishing activity trends, November 2006. http://www.antiphishing.org/reports/apwg_report_november_2006.pdf.
- [17] Guido van Rossum. The Python Programming Language. <http://python.org>, last viewed on October 26th, 2006.
- [18] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
- [19] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. Internet Mail Consortium RFC. <http://www.faqs.org/rfcs/rfc3207.html>.
- [20] Collin Jackson and Helen Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of WWW 2007*, 2007. to appear.
- [21] Jesse James Garrett. Ajax: A New Approach to Web Applications, February 2005. <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [22] JotSpot. DojoDotBook. <http://manual.dojotoolkit.org/WikiHome/DojoDotBook/Book0>.
- [23] 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.
- [24] Brian Krebs. Microsoft releases windows malware stats, June 2006. http://blog.washingtonpost.com/securityfix/2006/06/microsoft_releases_malware_sta.html.
- [25] Ben Laurie. Openid: Phishing heaven. <http://www.links.org/?p=187>, last visited on February 3rd 2007.
- [26] Eric A. Meyer. S5: A Simple Standards-Based Slide Show System. <http://meyerweb.com/eric/tools/s5/>, last viewed on October 26th, 2006.

- [27] Kevin Miller. Cardspace extension for firefox. <http://www.fearthecowboy.com/2006/12/cardspace-extension-for-firefox.html>, last visited on February 3rd 2007.
- [28] Paul Johnston. A JavaScript implementation of the Secure Hash Algorithm. <http://pajhome.org.uk/crypt/md5>.
- [29] R. Fielding and J. Gettys and J. Mogul and H. Frystyk and L. Masinter and P. Leach and T. Berners-Lee. Hypertext Transfer Protocol, Version 1.1, June 1999. <http://wp.netscape.com/eng/ssl3/draft302.txt>.
- [30] Ed Rice. Passwords in the clear, 2006. <http://www.w3.org/2001/tag/doc/passwordsInTheClear-52>, last viewed on February 3rd 2007.
- [31] Simson Garfinkel. Fingerprinting Your Files. *MIT Technology Review*, August 2004. http://www.technologyreview.com/read_article.aspx?id=13718&ch=infotech.
- [32] Inc. SixApart. Typepad blogging service. <http://typepad.com>, last viewed on February 3rd 2007.
- [33] T. Berners-Lee and R. Fielding and L. Masinter. Uniform resource identifier (uri): General syntax, January 2005. <http://www.ietf.org/rfc/rfc3986.txt>.
- [34] T. Dierks and C. Allen. The TLS Protocol, Version 1.0, January 1999. <http://www.ietf.org/rfc/rfc2246.txt>.
- [35] Tim O'Reilly. What is Web 2.0. <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>.
- [36] Harvard University. Harvard university pin system. <http://pin.harvard.edu/>, last viewed on February 3rd 2007.
- [37] Stanford University. Stanford webauth. <http://www.stanford.edu/services/webauth/>, last viewed on February 3rd 2007.
- [38] 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.
- [39] Wikipedia. Usage share of web browser. http://en.wikipedia.org/wiki/Usage_share_of_web_browsers, last visited on February 3rd 2007.
- [40] Min Wu, Simson L. Garfinkel, and Robert Miller. Secure web authentication with cell phones. <http://groups.csail.mit.edu/uid/projects/cellphone-auth/>.
- [41] Yahoo. Browser-Based Authentication. <http://developer.yahoo.com/auth/>, last viewed on October 26th, 2006.