

Client Side Encryption for HTML Form Fields

David Moskowitz

Doctoral Student

Nova Southeastern University

Winter 2011

Abstract

Cloud based applications are a growing trend, offering businesses and users the advantages of lower costs, lessened maintenance concerns, and increased scalability. However, such application architectures open up additional security risks. Arguably, the most significant of these risks is the ability of service provider to view private data of which they are entrusted. Storing data in the cloud also opens up that data to potential hackers. This paper addresses these security concerns and proposes a solution using client side encryption. In this approach, data stored on the server that is not needed for server side processing is encrypted prior to submission. A symmetric encryption algorithm is used. The secret key is not shared with the service provider nor transmitted across the network. A prototype JavaScript application implementing these principles is presented. The prototype addresses a primary implementations issue- the maintenance of the client side encryption key. This paper also includes a proposal to incorporate this type of encryption into the HTML standard.

Client Side Encryption for HTML Form Fields

Cloud computing is all the rage, promising to simplify our application deployments, infrastructure provisioning, and data storage frustrations. Running applications from the web, we no longer need to maintain local infrastructure or distribute changes to users. We rely on programs such as Gmail and Google Docs to manage our data. Businesses rely on Salesforce.com to manage their customer relationships. A large segment of the population has divulged a great many personal details to Facebook.

Cloud computing, as defined in this paper, involves a third party hosting service that provides computational resources as needed. The user or organization using the service is generally not involved with the internals of the applications or infrastructure¹.

The popularity of cloud computing is due in large part to the advantages this approach offers. Businessinsider.com describes the following advantages (Rosoff, 2011):

1. Efficiency: lower hardware and IT costs
2. Agility: add capacity fast
3. Flexibility: pay for what you need.

The author also notes several reasons to be concerned about this technology and why some companies are holding off on incorporating cloud technologies. Companies are avoiding the use of cloud computing due to reasons such as (Rosoff, 2010)

1. Security Concerns
2. Complexity of architecture, compared to locally hosted solutions.
3. Proprietary platforms
4. Lack of support services in many cloud providers

¹ This scenario is distinct from infrastructure outsourcing, where only the hardware is maintained by a third party and the organization retains responsibility for application development, deployment, and maintenance.

This paper addresses some of the security concerns regarding cloud computing.

When using the Cloud, we rely on third parties to secure our data. It is often assumed that appropriate levels of security are in place. However, no widely accepted standard for what is "appropriate" exists, especially with new and rapidly changing technologies like cloud computing. In most cases, we simply hope that best practices in data security and protection are applied. Larger companies may have the resources to conduct security audits, but the majority of end users subscribing to the most popular internet services have no such capability.

The ongoing history of security breaches in even the largest and most trustworthy installations imply that complete security can never be assumed. High profile incidents occur frequently, as a quick Google search on the matter will show. One of the largest security breaches yet hit TJX Companies Inc., owner of the T.J.Maxx clothing store. From 2005 to 2007, hackers accessed over 45 million debit and credit card numbers (Vijayan, 2007). Most of these cards were cancelled and reissued, at great cost to the companies and consumers involved. More tech savvy companies are not immune to security concerns either. In 2010, the names and email addresses of over 100,000 Apple iPad owners were compromised due to weaknesses in the device's underlying AT&T network (Heussner, 2010). This list was said to include many CEOs, celebrities, and other high profile early adopters of the trendy new technology.

Current standards of data protection may in fact fall short of what a naive end user may expect. One might assume that Facebook encrypts all the personal secrets you divulge and share them only with persons you authorize, but this is generally not the case. Ignoring the sharing of information with advertisers and the ambiguous privacy policies that are sometimes in place, the simple fact is that most data submitted to the cloud is not encrypted. While password data is generally stored in hashed form (for the major cloud services at least), other data may not be encrypted at all.

Some sensitive information, such as credit card numbers used for purchases, need to be available on the server side for processing. While we can hope good secure practices are used, it is a truism that if the server can read the data, a hacker can too. However, in many cases, that data need not be stored on the server at all. Users want their mail available from any computer. They do not want Google to be able to read that email. Users wish to share information with their friends on social networks. They do not necessarily want Facebook to read that information.

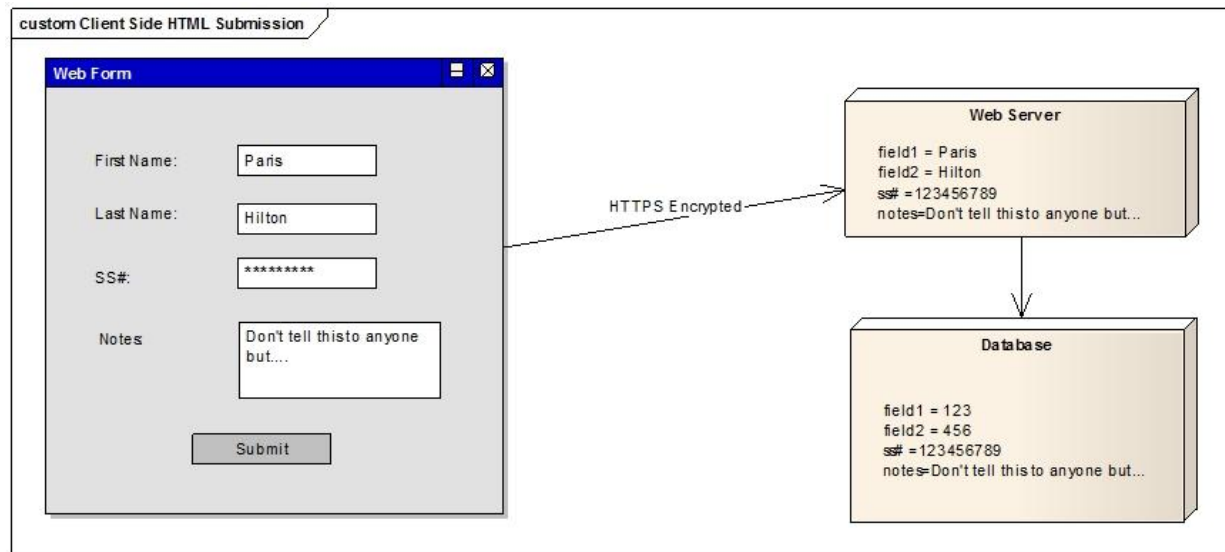
This paper contends that only data the server must act on should be accessible to the server. All other data should be stored in encrypted form. There is a large segment of sensitive data that need not be shared in unencrypted form with third party service providers such as web hosts. Instead, this paper proposes that such data should be encrypted on the client and sent to the server as ciphertext.

(In)secure Application Example

As this paper is not an attack on any one company's business methods, consider a hypothetical web-based contact management application, similar to many used by consumer and business users. In such an application, a user wishes to store, or possibly share, contact information. No matter what server side security methods are used, it is likely that for data other than passwords, which are generally stored hashed, the server or the server's operators can view the data. This situation poses a security risk.

Let us first examine the process of data submission in this hypothetical application in more detail. HTTPS (HTTP over SSL/TLS) is the current standard for encrypting web page form submission. HTTPS can be used in data transmission for server authentication and to protect against eavesdropping. It is assumed, however, that both parties wish to share data with the other. This assumption may not be true. To illustrate this concern, we will look at an example contact

management application². In this scenario, the user enters the information into the application form. Data is submitted and transmitted via secure HTTPS. The web application receives the form data and stores it in a database. The data flow is shown in the following diagram:



This scenario illustrates several best practices in security and application development. Note that the Social security number field is entered using an HTML password field to protect from prying eyes. Data is transmitted securely via HTTPS, protecting against eavesdropping and ensuring server authentication. The Server application can easily read the transmitted data, as decryption is general transparent to the web application.

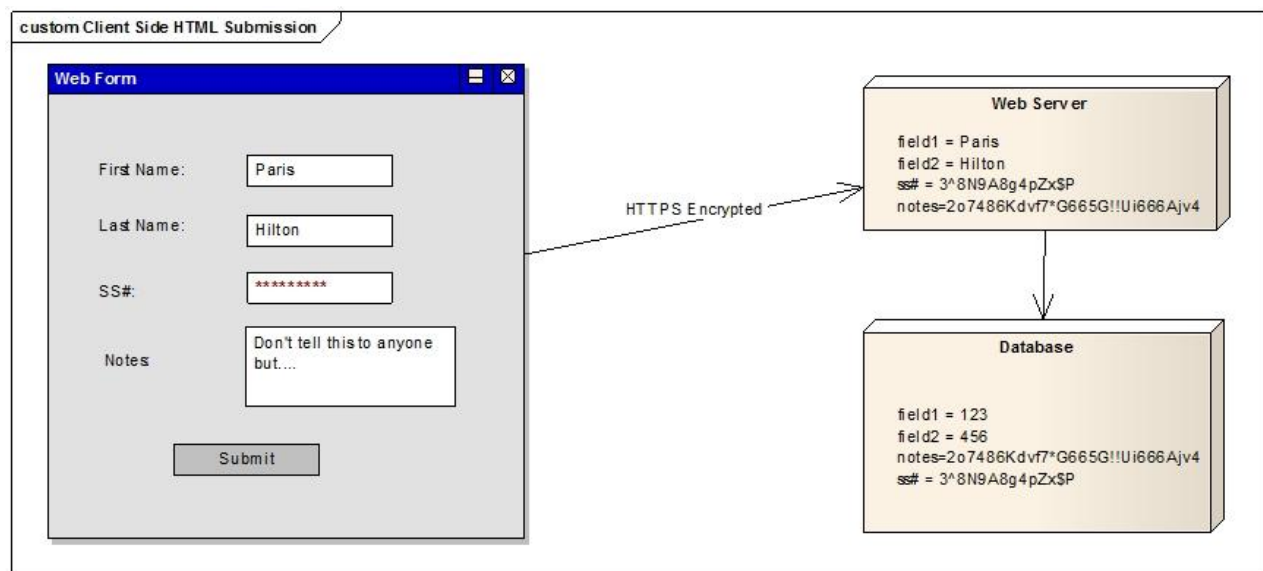
There are potential security issues in this scenario as well. Firstly, the server *can* decrypt the data. As was already mentioned, the server may have no real need to do anything with the data except store it. This issue will be addressed in great extent in the rest of this paper. Another issue is

² Paris Hilton is used in the example as homage to a famous 2005 hack on her T-Mobile sidekick address book, illustrating the problem with transparent storing of information online.

that the data is rarely stored encrypted. Even if the application claims to use encryption, we must take the hosting service's word that they are encrypting and that proper safeguards are being taken.

The use of HTML password fields is not a method of encryption. This HTML element simply protects against over the shoulder eavesdropping. As we can see, the cleartext data is fully available to the server, which generally stores the information in the database as received. Social security numbers, due to their special significance and legal accountability, may be stored encrypted, but for performance reasons, most other fields are not encrypted.

A preferable scenario is the following:



In this diagram, a web page is presented to a user who can view or enter the desired information, just as before. The form contains three fields: two normal textbox fields and one encrypted text box field, the later indicated by color (though an actual implementation would likely have a more unique indicator). Upon submission, the sensitive fields are first encrypted. The entire form is then submitted using HTTPS. When the server receives the form submission, HTTPS decryption will provide field1 and field2 as plain text, but the encrypted fields, SS# and notes, will be available only as ciphertext. If these fields were then stored in a database, the plain text of the non-encrypted fields will be stored but the encrypted field will be stored as ciphertext.

Securing Applications Through Client Side Data Encryption

In the above scenario, key sensitive fields are encrypted. Only those fields the server needs to act on should be stored in unencrypted form. As an example of this principle, consider a bank account transfer. The server certainly needs to see the balance and amounts associated with the transaction. The server may not need access to the notes you enter along with the transfer.

We can classify data into several types as illustrated in the table below:

Data Use \ Data Sensitivity	Sensitive	Non-sensitive
Transparent	<i>Client Side Encryption</i>	No Encryption
Active	Server Side Encryption	No Encryption

The table illustrates recommended encryption methods based on data sensitivity and transparency. We define transparent data as data that the server does not act on that can simply be stored. Active data is that which the server needs to use, either as part of a larger business process or even for simple searching. Encryption may not be feasible for all sensitive data, since search functionality may need access to encrypted data. However, decryption could be done when the data is brought into memory, such as in the case of an in memory database or cache. This paper addresses specifically transparent sensitive and proposals the use of client side encryption

A symmetric encryption algorithm would handle these cryptographic requirements. The browser can perform encryption on the data prior to form submission. In this approach, the entered plaintext would be converted into ciphertext and submitted. The reverse would happen on data retrieval. The ciphertext will be decrypted and cleartext displayed to the user. A symmetric encryption algorithm is therefore the best choice and the encryption key can be kept private and only on the client side.

A straightforward implementation would only need a single encryption key. The encryption key could be generated from a master password. An algorithm such as DES, used for securing passwords in UNIX systems, can be used to generate appropriately sized encryption keys from user passwords. This key could then be used for all client side encryption operations.

This approach is predicated on the security and integrity of the encryption key and therefore of the master password. In our proposal, the master password is never transmitted to the web service provider. The remainder of this paper will address issues related to master password security and proposed implementation of this approach.

At the highest level of abstraction, our proposed solution is the following

1. On application start: user enters a master password
2. On form submission: sensitive html fields are encrypted prior to submission
3. On page load: encrypted html fields are decrypted before display.

First, we will look at several related technologies and tools that provide similar functionality to what we propose. Next, the paper examines possible solutions and implementation choices for client side encryption.

Related Tools

While no implementation of the exact client side encryption we propose exists, other tools that solve similar problems do. The most similar of these tools are from the category of password managers. Products such as Roboform (Siber Systems, Inc., 2011a) and LastPass (LastPass, 2011a) maintain a local encrypted store of online passwords. Both tools have browser plugin functionality to enable on-the-fly password filling, creation, and saving. These tools also support pure JavaScript solutions, called bookmarklets, that store no data client side. Both support online storage of encrypted passwords and synchronization across multiple computers. The encryption key is not

stored online and all encryption occurs client side. Only encrypted data transferred across the network.

The following process, taken from the LastPass web site (LastPass, 2011b) , illustrates how this type of tool work. All encryption is handled client side using a master key. The user first creates a master password. A SHA-256 digest of master password is used to create an encryption key. A digest of this key and the user name is used for login to the back end web site. The login is salted and stored on the server. Local encryption of password data uses AES-256. SSL is used for server communication to enable authorization. Roboform appears to work in the same manner (Siber Systems, Inc., 2011b).

JavaScript and Bookmarklets

A hurdle to implementing secure client side encryption is the difficulties in managing the encryption key or master password. If this password is stored, how can it be stored securely? If this password is not stored, how can it be available for encryption and decryption as needed? Password managers solve this problem in several ways. When using the plug in approach, a memory persistence process runs and monitors browser communications. The user enters the master password when the process starts or when the password is first needed. The plugin can also implement any necessary password expiration rules. JavaScript plugins do not run as a separate process. Instead, these are implemented as bookmarklets - stored pieces of JavaScript code. When a page containing a login form is encountered, the user clicks the bookmarklet (generally saved in the browser toolbar) to execute the JavaScript code. The JavaScript queries the back end server for password information related to the domain of the current page. If such data is available, it is returned in encrypted form. The encryption key is stored in the JavaScript bookmarklet itself. The key is a combination of the user password and a random 256-bit number generated when the user logged in. This random number is also sent to the back end server on login and used for server side

encryption. Bookmarklets are more sensitive to issues revolving around the use of public computers, as the encryption key can be seen by inspecting the JavaScript code, though this code is usually obfuscated to some extent. To lessen the impact of these concerns, logout functionality, either manual or through automatic timeout, and the use of one-time passwords, are often supported. More details are provided on the LastPass Web Site (LastPass, 2011a) .

The JavaScript bookmarklet needed for our proposed encryption introduces distinct problems above those encountered in password manager bookmarklets. An additional problem is the need to maintain a user password across multiple pages in an application. We would prefer not to require the user to enter the password each time an encrypted page is encountered... We also prefer the user not need to click a bookmarklet type link to manually initiate the encryption or decryption of each page.

It is worth noting that the issue of page-to-page navigation is eliminated by using a client side application architecture that uses only a single web page. An example of this type of approach is Google Web Toolkit (GWT) and Ext-JS (Senscha, 2011). Using the single page approach, a password could be entered on initial load of the application and persisted in application memory for the life of the application. Using Ajax techniques, such client side toolkits can simulate multi-page loading without leaving the original html page. This approach makes maintaining application level data much easier.

Several examples of single web page application that specifically address this type of client side encryption problems do exist. ThreeTags is an online notebook supporting client side encryption of data (ThreeTags Inc.). UserEncrypted.com is a simple prototype of a similar system, based on an article written by the site's creator (Beanizer.org). These examples further illustrate the need and potential of client side data encryption.

The single page approach, though it makes client side encryption simpler, is not the approach taken for our client side encryption. We wish to support a more typical page-to-page navigation style application, as the majority of existing and new applications still use this paradigm.

Cryptographic Tools for JavaScript

The architecture described above lends itself to a bevy of standard encryption tools and techniques. Open source JavaScript libraries exist for client side browser based cryptography, such as the JSBN library (Wu, 2009) developed at Stanford. This library includes JavaScript implementations of RSA and Elliptic Curve cryptography. AES algorithms are implemented in (AES Advanced Encryption Standard). This tool is used on userencrypted.com. A simple example of client side encryption is provided as well in (Beanizer.org).

jCryption is a tool for encryption of form data prior to submission. In this respect, it is strikingly similar to our proposal. As decryption still happens on the server, the main use for jCryption is enabling encrypted form submission for servers that do not support SSL, although authentication is not supported. The tool also does not support full client side only encryption, as there must be a public key exchange with the server. While originally intended to encrypt entire form, the latest version (1.1) supports string encryption. jCryption is implemented as a jQuery plug in which makes incorporation into existing and new JavaScript code quite easy.

Client Side Encryption Implementation

The tools discussed in the previous section are all examples of large applications or services incorporating client side encryption into their overall solution. Instead of just *implementing* an application that supports client side encryption, such as ThreeTags, we wish to develop a toolkit that easily *enables* applications to support client side encryption. The rest of the paper will describe possible options for such an implementation. A single option is selected and an initial prototype described.

Implementation Option 1 - Native HTML

Ideally, client side encryption could be implemented as native HTML tags.

```
<form method="post" action="">
  <label>SS#:</label>
  <input type="password" name="ssn" encrypted="true"/>
  <label>Notes:</label>
  <textarea name="notes" encrypted="true" />
</form>
```

In the sample HTML above, the developer only need add an additional attribute, "encrypted" on each appropriate form element and the browser would handle the rest, similarly how the browser can handle the details of HTTP form submission. The encryption and decryption of data could be handled as described in the prior section. Of course, this capability does not currently exist in HTML and no indication exists that it is under consideration. A draft proposal for this capability is included in the Appendix A.

Firefox 4, released in 2011, includes password synchronization features (Firefox Sync for Mobile). Additional browser data, such as history and bookmarks can also be synchronized. Client side AES 256 encryption is uses along with a user entered master password. This approach is the same as was seen in most of the other solutions examined in this paper and is identical to the dedicated password managers discussed.

The inclusion of this type of client side encryption in a major browser bodes well for the possibility of additional features being implemented in Firefox and other browsers. This should be a first step to implementing similar encryption for HTML for fields as proposed in this paper.

Implementation Option 2 - Native Browser Plugin

Many enhancements to HTML and web browsers are implemented as browser plugins. The plugin approach is potentially the best implementation of the type of encryption proposed in this paper. The plugin would need to examine web content for each page and apply

encryption/decryption as necessary. Though outside the scope of this paper, this potential solution should be examined as an alternative to the methodology proposed here.

Implementation Option 3 - JavaScript Plugin

Although browser plugins have proved the preferred solution for password management and form filling, the kind of form submission and interception required for more generalized client side encryption might require a more integrated JavaScript-based approach to handle encryption prior to SSL submission. In addition, choosing a plug in solution would force the development of separate plugins for each supported browser. Likely, certain browsers would remain unsupported, due to lack of a suitable plug in infrastructure or lack of sufficient resources to build the plugin. A JavaScript based approach built on web standards is preferable for a system meant to read HTML off multiple sites and run across all popular web browsers

Several possible implementations using JavaScript are possible. One possibility is the use of frames. A hidden frame could remain open and accessible at all times the application is active. The password could easily be stored in the Document Object Model (DOM) of the hidden frame or as JavaScript Data. Even if a hidden frame were used, such a requirement would be extremely limiting, as framesets do not confirm to modern web standards. Such a change would also make retrofitting existing especially difficult.

The approach that will be addressed in this paper is the implementation of client side encryption using a JavaScript plugin. JavaScript can intercept page loads and submissions and perform the necessary cryptographic operations using widely available and accepted libraries.

Incorporating a Trusted Third Party

As password management tools typically include a server component, we could incorporate a back end password server into our proposed process. Under this architecture, the client would authenticate to the password server and the master key, encrypted using SSL, would be returned.

This communication would need to happen on every page where encryption is applied and would likely add performance some overhead. Even with the potential performance penalty and additional infrastructure requirements, this approach is very feasible.

The server in this case, takes the form of a trusted third party. Ideally, this party would not gain access to the full encryption key used. This can be accomplished by using the domain as part of a dynamically calculated encryption key. While the trusted third party could potentially determine the key of any one domain, additional server side authentication (i.e. site passwords) for each application would likely be in place independent of the client side encryption and further prevent unauthorized data access. Since this solution does not store password, only encrypted form data, this should provide a reasonable level of security.

It turns out that a third party key server is not needed due to advances introduced in HTML5. A trusted third party might still be required to support older browsers.

HTML 5 Support

As was mentioned earlier, the main hurdle of client side encryption is the persistent storage of the master password. If this problem cannot be solved, the user could be forced to enter the master password on every page load in the application. We addressed this concern using a trusted key server. This issue is also lessened if a single page web design is used.

Another potential technology that could address this situation is the client storage capabilities introduced in HTML 5 (W3C, 2011). These features are similar to cookies of earlier HTML versions. Unlike cookies, the client-stored data is not transmitted to the server with each web request.

Several versions of HTML 5 client storage are available, included disk based storage that persists across browser sessions, and memory based storage that persists only as long as a browsing

session is active. The data is available only to the originating web site and persists in memory until the browsing session ends

The use of local storage is potentially more secure and capable of storing greater volumes of information than what was available using cookies. More importantly, session storage could be used to store the master password once it is entered. The user would still need to enter password data for each web site, but only once for that site. The master password should therefore be available to each page in the web application after it is initially entered by the user.

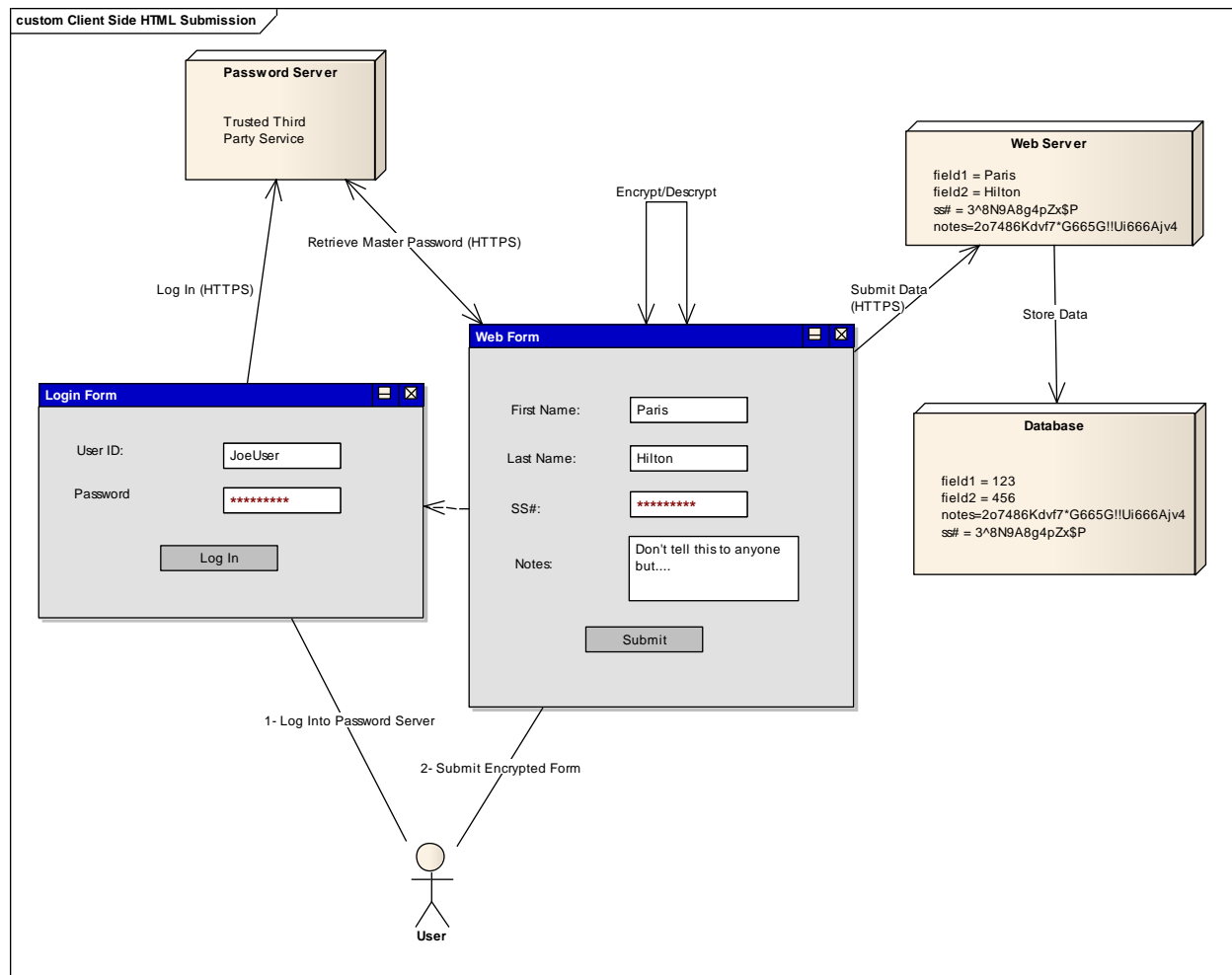
There is nothing in this approach that limits the user to a single master password. The user could just as easily create a separate password for each site. As storing a separate password for each site this could lead to a password proliferation and be difficult for the user to manage, the client side passwords could be maintained through a password management tool such as RoboForm. Having a different password for each side avoids the issues of a compromised password on one site leading to security breaches on other sites.

While not directly related to the requirements of this proposal, disk based local storage could be used to store encrypted password information in a manner similar to browser plugin implementation of popular password managers. However, as no encryption capabilities exist in the HTML 5 specification, any stored data would be available in clear text unless additional encryption is applied. JavaScript writer Nicholas Zakas has proposed SecureStorage to address the lack of encryption in HTML 5 local storage (Zakas). This specification defines an API for disk based local storage and encryption of key value pairs.

Client Side Encryption Architecture

We can now put all the pieces together and describe our proposal for client side encryption. The proposed architecture is illustrated in the following diagram. We present two approaches. The

first uses a key server and supports pre-HTML5 browsers. The second uses the additional features of HTML5 to eliminate the need for the key server.



Key server architecture.

1. User navigates to a client encryption enabled application, such as our sample contact manager.
2. When an encrypted page is accessed, a web service call is made from the client browser to the password server. The first time, a response will be returned requiring login. The application will then display a dialog box and the user will enter his user id and password. Upon submission, the master password will be returned and can be used for cryptographic operations on that page.

3. When the user navigates to another page, a web service requesting the master password will again be made. As the request is coming from the same browser, no login will be needed.

The master password will be returned and be available for cryptographic operations on that page.

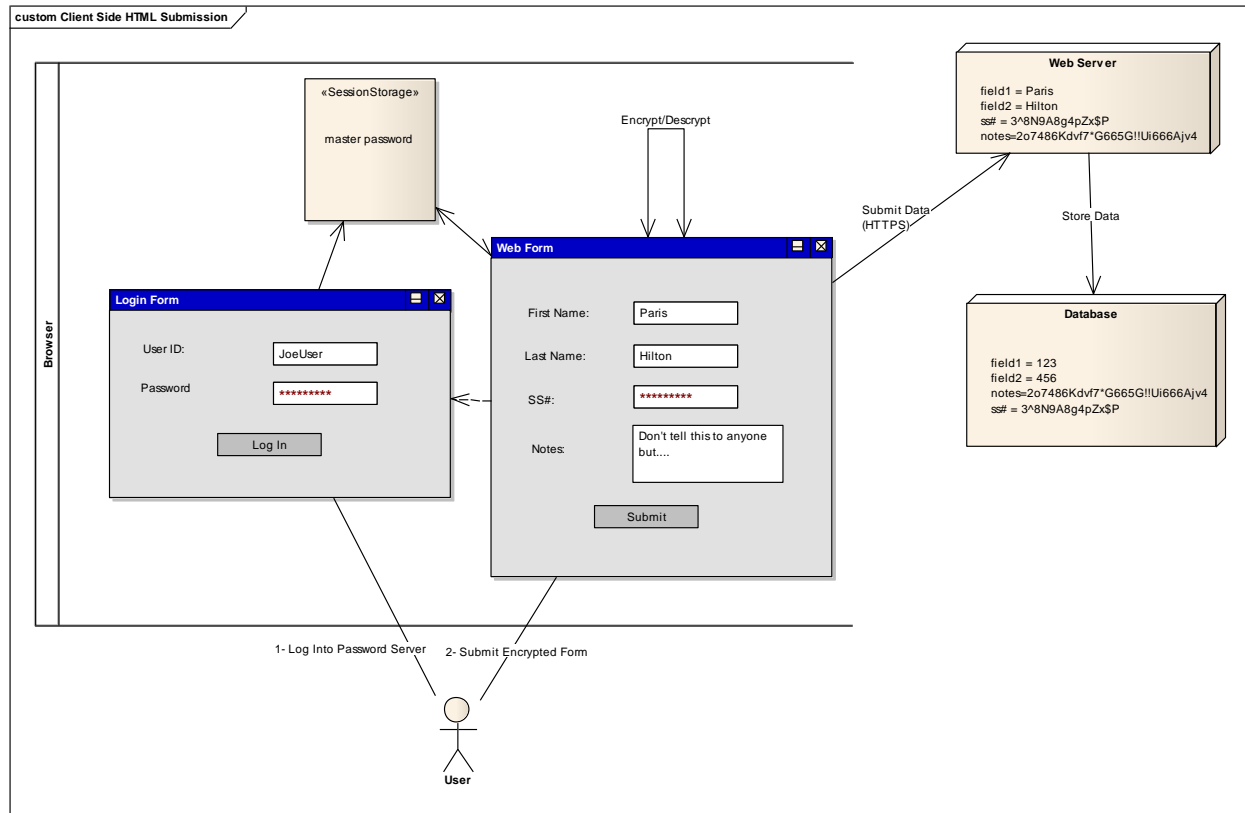
The password server functionality could be distributed as a JavaScript library, to easily integrate into web applications.

It may be argued that this process places too much trust in both websites. It is certainly possible for the contact management website to capture and transmit back the master password. However, such concerns are similar to the trust we give to native programs we install on our machines all the time. As JavaScript is open and observable, any attempt to steal user information would likely be uncovered. Additional research is needed in this area to further safeguard this process.

It should be noted that JavaScript imposes a limitation that code can only communicate with the machine that served the code. However, JSONP is a standard technique to get around this limitation, allowing JavaScript code to call another web server that complies with the protocol (jQuery Community Experts, 2009, pp. 407-409).

Session storage architecture.

The second architectural approach uses the session storage capabilities of HTML 5 to persist the user password for the entire browsing session.



In this approach, the key server is replaced by HTML 5 SessionStorage. This diagram further specifies the elements running in the user browser through the demarcated box on the left.

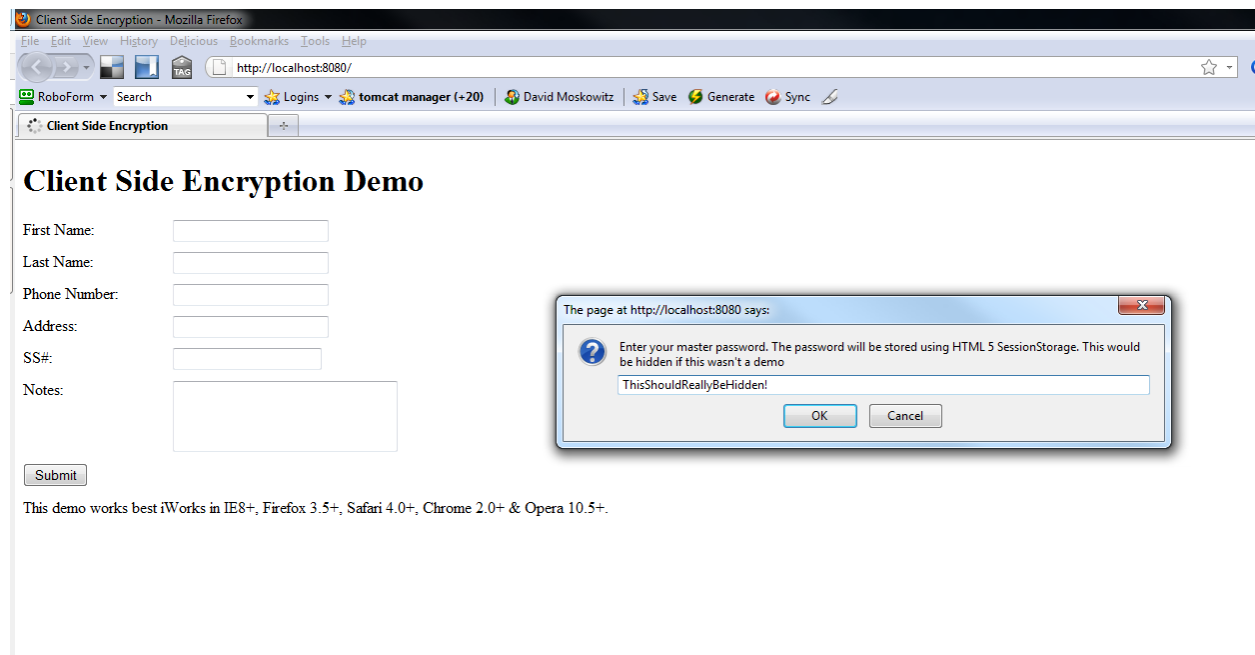
The scenario is a slight modification to the first key server scenario.

1. User navigates to a client encryption enabled application, such as our sample contact manager.
2. The application checks SessionStorage for the master password variable.
3. If the master password is not found, the application will then display a dialog box and the user will enter this password. Upon entry, the master password will be stored in Session Storage
4. The master password will be returned from SessionStorage and be available for cryptographic operations on that page.

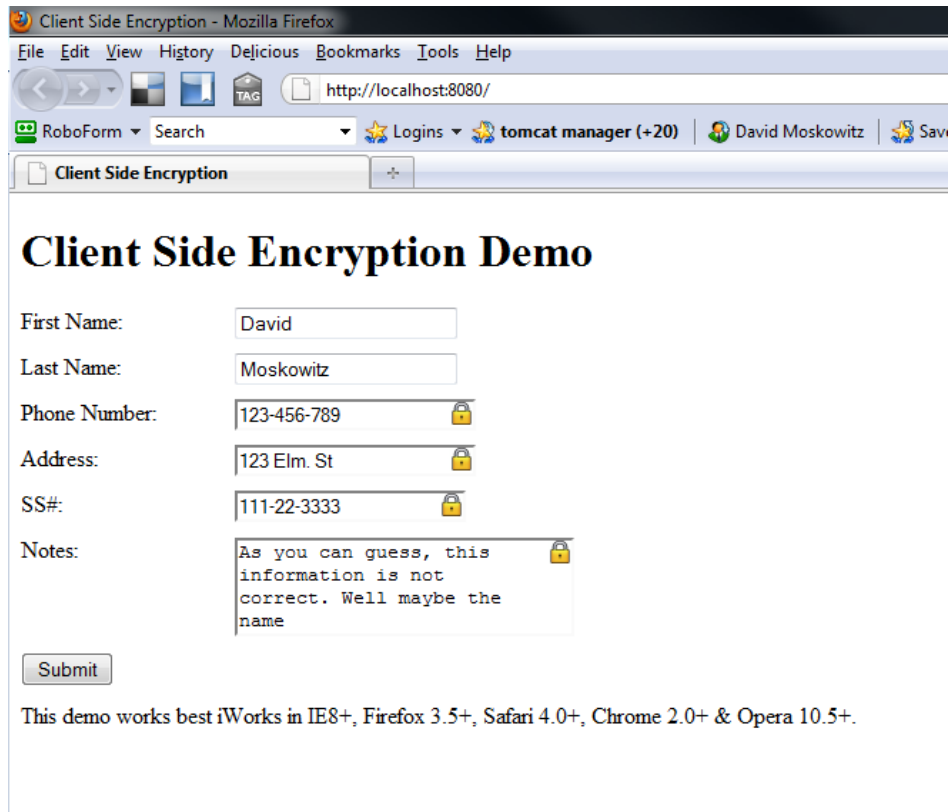
Prototype Implementation using SessionStorage

A proof of concept was created using the AES cryptographic tools from (AES Advanced Encryption Standard). Implementation proved quite simple using these tools along with jQuery. Session storage proved easy to use for browsers that supported it. The full application source code is provided in Appendix B and distributed along with this paper.

The following screen captures demonstrate the functionality of the demo application1- User is presented with the initial contact entry web page. As a password has not yet been entered, an input box is displayed. In a production application, a HTML dialog window should be displayed so the password can be hidden as it is entered on screen.



2- After password entry, the user is free to enter data values into the form fields. Note the indication of encrypted fields with the lock icon.



Client Side Encryption - Mozilla Firefox

File Edit View History Delicious Bookmarks Tools Help

http://localhost:8080/

RoboForm Search Logins tomcat manager (+20) David Moskowitz

Client Side Encryption

Client Side Encryption Demo

First Name:

Last Name:

Phone Number:

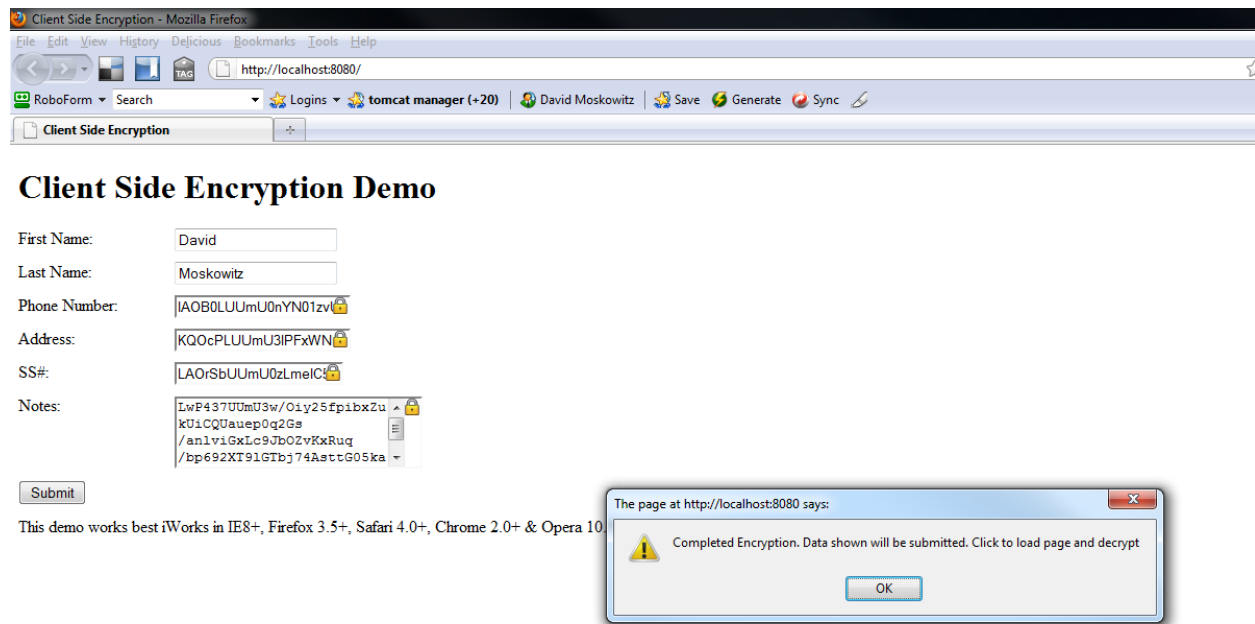
Address:

SS#:

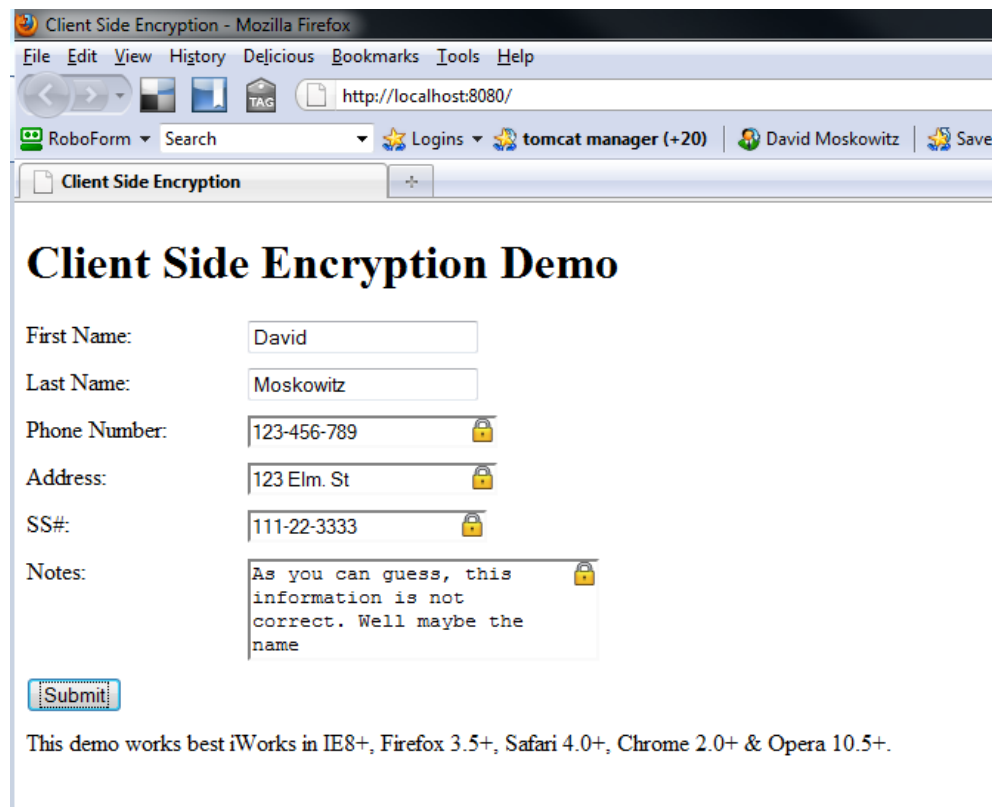
Notes:

This demo works best iWorks in IE8+, Firefox 3.5+, Safari 4.0+, Chrome 2.0+ & Opera 10.5+.

3- Upon form submission, the indicated fields are encrypted and the form is submitted. The demo application pauses to allow inspection of the encrypted fields.



4- Following form submission, the encrypted data is decrypted and the form displayed using plaintext values.



As the master password is already stored in SessionStorage, the password dialog box is not displayed on subsequent page loads. The master password will remain accessible as long as the session is active on the given domain.

The full code listing for this demonstrating is provided in Appendix B. However, much of this code is "wrapper" code used to simulate submission and retrieval of data from a back end server. The actual encryption and decryption methods are only a few lines each. The following code excerpt isolates the pieces relevant to client side encryption. Some code is modified slightly from the actual application code to simplify explanation.

```
/* add encryption method to page submit event. Call loadPage function, described below*/
$(document).ready(function() {
    loadPage();
    $('form').submit(function() {
        encryptValues();
    });
});

/* called on page load. Check to see if a password is already set in SessionStorage.
Format encrypted fields with appropriate icon.
decrypt encrypted fields for display an cleartext.*/

function loadPage() {
    checkPassword();
    formatEncrypted();
    decryptValues()
}

/* Convenience function to return password.*/
function getPassword() {
    return sessionStorage.password;
}

/* Store the password in sessionstorage. this password will be available as long as the session
with the server is active */
function setPassword(pwd) {
    sessionStorage.setItem('password', pwd);
}

/* Check if a password is already set. This check needs to occur on every page load. prompt the
user for a password if not already set. This function uses a simple input box. An actual
implementation should use a modal html form so the password can be hidden from screen display */

function checkPassword() {
    if (getPassword() == null || getPassword() == '') {
        var msg = "";
        if (sessionStorage != null) {
            msg = "Enter your master password. The password will be stored using HTML 5
                SessionStorage. This would be hidden if this wasn't a demo";
        } else {
            msg = "Enter your master password. The password will be using a local
                variable. This would be hidden if this wasn't a demo";
        }
        setPassword(prompt(msg, ""));
    }
}
```

```
/*Perform decryption on all fields contained the "encrypted" attribute. The actual value of the
attribute is ignored for this demo */
```

```
function decryptValues() {
    if (getPassword() != null && getPassword() != '') {
        $('*[encrypted]').each(function() {
            var ciphertext = $(this).val();
            var origtext = Aes.Ctr.decrypt(ciphertext, getPassword(), 256);
            $(this).val(origtext);
        });
    }
}
```

```
/*Perform encryption on all fields contained the "encrypted" attribute. The actual value of the
attribute is ignored for this demo */
```

```
function encryptValues() {
    if (getPassword() != null && getPassword() != '') {
        $('*[encrypted]').each(function() {
            var ciphertext = Aes.Ctr.encrypt($(this).val(), getPassword(), 256);
            $(this).val(ciphertext);
        });
    }
}
```

```
/* Though not absolutely necessary, this method visually format encrypted fields. In this demo,
we add a lock icon */
```

```
function formatEncrypted() {
    $('*[encrypted]').css({'background-image':'url("lock.png")',
        'background-repeat':'no-repeat',
        'background-position':'right top',
        'padding-right':'15px'
    });
}
```

As this demo is built to run as a local HTML page without a web server, it should be noted that only Opera 11 supports Session Storage running in this mode. Internet Explorer 8 and Firefox 3.6, the other two browsers tested, will only support session storage when running off an actual web server. Firefox will actually throw an error when run locally. It is recommended that Opera 11 be used to run the demo prototype.

Related Research

Other researchers have looked at using encryption to keep cloud based service providers from accessing the data they store. Much of the recent interest in this issue has occurred in area of social networking. This area is certainly associated with privacy concerns, due to the type of data shared and the interrelations that can be inferred due to large volume of users.

Anderson et. al. define a generic security architecture for privacy in a social network environment (Anderson, Diaz, Bonneau, & Stajano, 2009). Their prototype uses a Java Web Start

based application to server HTML and JavaScript code to the user's browser. This "smart client" acts as an intermediary, or proxy, between the user and the social network. This architecture will only work with applications specifically designed for their API. Existing applications, specifically those using SSL, will not be able to communicate with the proxy.

Frikken and Srinivas describe a key distribution scheme to enable trusted relationships between users without the use of a trusted third party (2009). Keys are derived based on the social networks graph of friends and permissions set based on distance (in nodes) in the graph. This methodology eliminates the need for a handshake since the graph alone is sufficient to generate the necessary keys.

One research project devoted specifically to Facebook privacy is FlyByNight (Lucas & Borisov, 2008). The authors address privacy risks in social networks "the foremost of which being that social network service providers are able to observe and accumulate the information that users transmit through the network" (Ibid.). The FlyByNight solution incorporates a Facebook application, allowing users to maintain their existing Facebook friend links while encrypting data so that is only visible to intended parties.

FlyByNight maintains a database of encrypted private keys. Public and private keys are used in FlyByNight since the system provides secure communications between two parties. The application utilizes Facebook as an intermediary transport medium. Our system could utilize such techniques if we wished to share encrypted information with others users. This is a likely future enhancement. The Lucas & Borisov paper also discusses, but does not implement, additional techniques to protect against the service modifying the JavaScript libraries used in the encryption scheme. Such a technique could be applied to our implementation as well, as protection against altered JavaScript was not implemented.

One risk with utilizing a cloud service in ways that the service provider does not approve is that users utilizing FlyByNight or similar tools can be banned from the service. If Facebook was to see a large amount of apparently encrypted data going through the network, they could deny service to those users. Ideally, the user community would demand that Facebook allow such technology, or implement it themselves, but that would require a mass movement. Otherwise, it is in Facebook's interest to accumulate, utilize, and potentially market this data. Our proposal does not attempt to bypass a service provider's implementation. The client side encryption methods proposed in this paper are meant as tools for the service providers themselves to offer a more secure service to their user base and perhaps differentiate their services from other less secure offerings.

Earlier research in the area of client side encryption dealt with issues other than the privacy of social networks (as those networks had not yet achieved the popularity they have today). Many of these papers examine the issue of lack of trust in web service providers and how to implement client side encryption

Hassinen and Mussalo propose the user of a Java applet to handle client side encryption (Hassinen & Mussalo, 2005). Encryption keys are stored on mobile media (such as a USB thumb drive) or through a key server from a trusted third party. The only potential advantage of the Java applet approach is the ability to interface with computer hardware, a feature not currently possible with JavaScript. Encryption functionality may also be faster using a Java applet than in JavaScript. The use of a Java applet still does not allow transparent cryptography for a multiple page application unless features such as frames are included in the (re)design. Loading the applet for each page would likely add great overhead to the load time of each page compared to a JavaScript call to a key server. This overhead would likely outweigh any advantage in cryptography speed.

W3bcrypt is a system for applying client side encryption using CSS-type style sheets (Stavrou, Locasto, & Keromytis, 2006). The system handles point-to-point encryption using a peer

based PGP approach. Many of the ideas of our client side encryption solution are similar to those discussed in this paper. The motivation for their research is also identical to ours. They write, there are "plenty of circumstances when even the web server cannot be trusted". While existing security solutions address the transport layer (SSL) or network layer (IPSEC), "neither can protect from a compromised or malicious application server because their confidentiality and integrity protections do not reach up through the application layer" (ibid.). The application layer in this case is the web and application server. Another motivation of W3bencrypt is targeted encryption - to allow only those parties needing a piece of information to decrypt it. Many of the examples discussed in the paper deal with larger data flow scenarios, like credit card purchases in an ecommerce system. In such a use case, the merchant does not need to know the credit card information; it can pass through encrypted to the bank. Likewise, the bank does not necessarily need to know the details of what was purchased, an important privacy consideration due to potential aggregation of user information by a single party (the bank). Our proposal can be seen as a small subset of this scenario, generally involving only two parties, the end user and the service provider. As mentioned earlier, our proposal could extend to multiple parties in the case where we want to enable other users to view our encrypted data.

In particular, W3bencrypt enables the automatic encryption of html forms marked with a specific tag, such as

```
<div class="w3bencrypt">
...encrypted content here...
</div>
```

This feature is analogous to our encrypted="true" tag. HTML attributes like this are used in practice to target both JavaScript and CSS. W3bencrypt encryption is handled through a browser plugin menu. The desired text must be selected and encryption initiated. It is unclear how transparent encryption and decryption could be designed into the application. The W3bencrypt web site mentioned in the

paper is no longer active and no additional examples of transparent encryption are provided.

However, this research seems the most similar to our proposal and the most promising for future enhancements. Further research could prove useful to modernize this approach and take advantage of advances in web technologies in the past 5 years, as client side functionality and security have become even more important

While not directly related, PwdHash, attempts to increase the strength of user passwords through hashing techniques (Ross, Jackson, Miyake, Boneh, & Mitchell, 2005). The paper is useful as an illustration of some of the attacks and defenses that must be considered in browser based cryptographic solutions. The idea of PwdHash is strikingly simple and useful: transparently strengthen a user-entered password using a cryptographic hash using the site domain as the salt. This technique lessens the risks of using the same password for multiple sites, as the password is transparently altered and strengthened. A paper also describes the difficulties encountered in implementing such as simple concept.

PwdHash does not need to solve the problem of storing a user password once it is entered, as the existing password prompts do not change. However, even recognizing what is a password is an issue. Even tools such as RoboForm cannot always distinguish actual passwords from fields using a password type to hide the display. Another issue addressed in the paper is protection from malicious JavaScript that grabs the cleartext password before it can be hashed. Browser extensions are suggested as a way to prevent this and secure the password. Our proposal and prototype implementation does not address non-trustworthy sites. However, techniques such as implemented by PwdHash could be useful and should be examined

Future Work

As mentioned earlier in this paper, the most promising and transparent implementations of client side encryption would be done by the browser creators themselves, either with or without the

support of the HTML standard. Short of that, the use of a native browser plug in should be examined and compared with the preliminary results found in building our JavaScript prototype.

Our client side encryption demonstration program proved to be a simple and successful implementation of the concepts proposed. This tool would be more usable, and go further towards the goal of creating a reusable client side encryption toolkit, if it was packaged as a jQuery plugin. Currently, the implementation is a collection of JavaScript methods that make use of jQuery functionality.

A jQuery plugin would likely have functionality applied to certain web page events similar to the following example.

On application load:

```
$('form input[encrypted=true]').encrypted();
```

This method would locate all elements that require encryption, load the indicator icon in the field display area.

On Form display:

```
$('form input[encrypted=true]').decrypt();
```

Decrypt any encrypted data prior to display

On form Submit:

```
$('form input[encrypted=true]').encrypt();
```

This method would encrypt all relevant fields prior to form submission.

Notice that the code from our sample application can be packaged in three short jQuery methods.

This approach would make implementation easier.

The sample application did not address many security risks around the use of sensitive data such as passwords, such as protection from malicious JavaScript code or related attacks. These security risks should be considered. Specifically, possible attacks should be listed and, if possible, addressed.

The methodology described in this paper offers no means for users to share data with other users. Encrypted data is only available to users with the master password. While sharing of passwords is certainly feasible and a key feature of multi party symmetric encryption, such an approach would not work well for the type of applications addressed. Sharing of keys also presents its own technical challenges, as a secure key exchange algorithm must be implemented.

A better approach is public key encryption. Several research efforts address this approach, including the previously mentioned Facebook application FlyByNight. Future work should address sharing of encrypted data using public key encryption. Research should also examine methods to define more granular permissions, such as at the field level and by user, similar to conventional access control methods. The implementation proposed in this paper only address encryption at the site level.

Another research question concerns how to allow the client to control the encryption process. Does the developer need to build encryption into the application (also opening up a risk to the developer intentionally compromising the security) as the current proposal specifies, or can the client choose when and where to apply encryption? Can client side encryption be applied to *any* web application? Such capabilities should be possible as long as the client can manage the various encryption applied. This capability could be built into a client side browser plugin, enabling arbitrary encryption of web data. This is potentially the most powerful enabler of client side encryption for HTML form fields.

Conclusion

In a cloud computing environments, the service provider may have full access to the application data. This paper has shown the potential role for client side encryption in cloud based web applications in order to secure sensitive data from service providers and server side hackers. While similar technologies have been implemented in various products, this paper is the first to

present a client side architecture designed for widespread usage and incorporation into existing and new applications. The sample application presented shows how the approach can be easily incorporated into web applications today using open source tools. Next steps for further standardization of this approach were also proposed.

References

- AES Advanced Encryption Standard*. (n.d.). Retrieved from Movable Type Web Site:
<http://www.movable-type.co.uk/scripts/aes.html>
- Anderson, J., Diaz, C., Bonneau, J., & Stajano, F. (2009). Privacy-enabling social networking over untrusted networks. *Proceedings of the 2nd ACM workshop on Online social networks (WOSN '09)* (pp. 1-6). New York, NY, USA: ACM.
- Beanizer.org. (n.d.). *Practical uses of client side encryption*. Retrieved April 3, 2011, from Beanizer.org:
<http://www.beanizer.org/site/index.php/en/Articles/Practical-uses-of-client-side-encryption.html>
- Firefox Sync for Mobile*. (n.d.). Retrieved from Mozilla web site: <http://www.mozilla.com/en-US/mobile/sync/>
- Frikken, K. B., & Srinivas, P. (2009). Key allocation schemes for private social networks. *Proceedings of the 8th ACM workshop on Privacy in the electronic society (WPES '09)* (pp. 11-20). New York, NY, USA: ACM.
- Hassinen, M., & Mussalo, P. (2005). Client controlled security for web applications. *Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)* (pp. 810-816). IEEE Computer Society.
- Heussner, K. M. (2010, June 9). *Hackers: Data Breach Exposed iPad Owners' Personal Info*. Retrieved 2011, from ABC News Web Site: <http://abcnews.go.com/Technology/Broadcast/hackers-data-breach-exposed-ipad-owners-personal-info/story?id=10871229>
- jQuery Community Experts. (2009). Using jQuery and JSONP. In C. Lindley (Ed.), *jQuery Cookbook*. Sebastopol, CA: O'Reilly Media.
- LastPass. (2011a). Retrieved from LastPass Web Site: <http://lastpass.com/>

LastPass. (2011b). *Common Questions*. Retrieved April 3, 2011, from LastPass Web Site:

http://lastpass.com/support_faqs.php

Lucas, M. M., & Borisov, N. (2008). FlyByNight: mitigating the privacy risks of social networking.

Proceedings of the 7th ACM workshop on Privacy in the electronic society (WPES '08) (pp. 1-8). New York, NY, USA.: ACM.

Rosoff, M. (2011, March 29). *Here's Why Cloud Computing Is So Hot Right Now*. Retrieved from

Business Insider: <http://www.businessinsider.com/why-businesses-are-embracing-cloud-computing-efficiency-agility-and-cost-2011-3>

Rosoff, M. (2010, November 17). *Seven Reasons Why Companies Are Sitting Out Cloud Computing*.

Retrieved from Business Insider: <http://www.businessinsider.com/seven-reasons-why-companies-are-sitting-out-cloud-computing-2010-11>

Ross, B., Jackson, C., Miyake, N., Boneh, D., & Mitchell, J. (2005). Stronger Password

Authentication Using Browser Extensions. *Proceedings of the 14th Usenix Security Symposium (SSYM'05)*. Berkeley, CA: USENIX Association.

Senscha. (2011). *Ext JS Cross-Browser Rich Internet Application Framework*. Retrieved from Senscha Web

Site: <http://www.sencha.com/products/extjs/>

Siber Systems, Inc. (2011a). Retrieved from Roboform Web Site: <http://www.roboform.com/>

Siber Systems, Inc. (2011b). *RoboForm Online Security*. Retrieved from Roboform Web Site:

<https://online.roboform.com/security>

Stavrou, A., Locasto, M. E., & Keromytis, A. D. (2006). W3bcrpt: Encryption as a stylesheet. (J.

Zhou, M. Yung, & F. Bao, Eds.) *ACNS 2006. LNCS, vol. 3989*, 349-364.

ThreeTags Inc. (n.d.). Retrieved from Three Tags Web Site: <http://www.threetags.com/>

Vijayan, J. (2007, March 28). *TJX data breach: At 45.6M card numbers, it's the biggest ever*. Retrieved from

Computerworld:

http://www.computerworld.com/s/article/9014782/TJX_data_breach_At_45.6M_card_numbers_it_s_the_biggest_ever

W3C. (2011, February 28). *Web Storage*. Retrieved April 3, 2011, from W3C Web Site:

<http://dev.w3.org/html5/webstorage/>

Wu, T. (2009). *RSA and ECC in JavaScript*. Retrieved April 3, 2011, from [http://www-cs-](http://www-cs-students.stanford.edu/~tjw/jsbn/)

[students.stanford.edu/~tjw/jsbn/](http://www-cs-students.stanford.edu/~tjw/jsbn/)

Zakas, N. C. (n.d.). *SecureStore 1.0 Proposal*. Retrieved April 3, 2011, from NCZOnline:

<http://www.nczonline.net/blog/securestore-proposal/>

Appendix A - Proposal for Client Side Encryption in HTML

This section outlines a brief proposal for including client side encryption of HTML forms in a future HTML standard. This methodology can also be supported by browser manufacturers prior to standardization...

1 - "encrypted" Attribute

1a- Browsers will recognize the attribute "encrypted" for input elements of type "text" and "password" and textarea elements.

1b- Allowable values for the "encrypted" attribute are the following:

Value	Description
Client	Client side encryption

Other values are reserved for future use.

2- Master Password

2a- Upon first encountering the "encrypted" attribute, the browser should display a dialog box allowing the user to enter his master password. Browser will have the option of allowing a single master password, a master password per domain, or a choice of either.

2b- Once a master password is entered for the appropriate context (global or domain), its entry should not be needed again until the appropriate timeout value.

3- Client side encryption

3a- Fields marked with encrypted ="client" should be encrypted prior to form submission.

Client side encrypted should use AES-256 algorithm.

3b - When a web page is displayed, fields marked with encrypted ="client" should be decrypted using the master password specific to that domain and an AES-256 algorithm. Decryption should occur before the page Document Object Model (DOM) is populated.

Appendix B - Demo Application Source Code

```

<!doctype html>
<html>
<head>
  <title>Client Side Encryption</title>
  <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"></script>
//following script available from http://www.movable-type.co.uk/scripts/aes.html
  <script type="text/javascript" src="aes.js"></script>
  <style>
    div {float: left;clear: left;margin-bottom: 10px;}
    label {width: 150px;float: left;}
    input[type=text], input[type=password] {width: 150px;float: left;}
  </style>

  <script type="text/javascript">
    //Store some local variables to simulate server side persistence
    var firstName = ""; var lastName = ""; var phone = ""; var address = "";
    var ssnun = ""; var notes = "";
    var password = ""; //used if SessionStorage is not supported

    $(document).ready(function() {
    //jQuery Page init function
      loadPage();
      $('form').submit(function() {
        //Overwrite form submit event
        encryptValues();
        storeValues();
        loadPage();
        return false;

      });
    });
    function loadPage() {
      checkPassword();
      formatEncrypted();
      loadFormValues();
      decryptValues()
    }

    function getPassword() {
      if (sessionStorage == null) {
        return password
      }else {
        return sessionStorage.password;
      }
    }

    function setPassword(pwd) {
      if (sessionStorage == null) {
        password = pwd;
      }else {
        sessionStorage.setItem('password', pwd);
      }
    }

    function checkPassword() {
      if (getPassword() == null || getPassword() == '') {
        var msg = "";
        if (sessionStorage != null) {
          msg = "Enter your master password. The password will be stored using HTML 5
SessionStorage. This would be hidden if this wasn't a demo";
        }else {
          msg = "Enter your master password. The password will be using a local
variable. This would be hidden if this wasn't a demo";
        }
        setPassword(prompt(msg, ""));
      }
    }
  </script>

```

```

function storeValues() {
    firstName = $('#firstName').val();
    lastName = $('#lastName').val();
    phone = $('#phone').val();
    address = $('#address').val();
    ssn = $('#ssn').val();
    notes = $('#notes').val();
}

function loadFormValues() {
    $('#firstName').val(firstName);
    $('#lastName').val(lastName);
    $('#phone').val(phone);
    $('#address').val(address);
    $('#ssn').val(ssn);
    $('#notes').val(notes);
}

function decryptValues() {
    if (getPassword() != null && getPassword() != '') {
        $('*[encrypted]').each(function() {
            var ciphertext = $(this).val();
            var origtext = Aes.Ctr.decrypt(ciphertext, getPassword(), 256);
            $(this).val(origtext);
        });
    }
}

function encryptValues() {
    if (getPassword() != null && getPassword() != '') {
        $('*[encrypted]').each(function() {
            var ciphertext = Aes.Ctr.encrypt($(this).val(), getPassword(), 256);
            $(this).val(ciphertext);
        });
        alert('Completed Encryption. Data shown will be submitted. Click to load page and
decrypt');
    }
}

function formatEncrypted() {
    $('*[encrypted]').css({'background-image':'url("lock.png")',
        'background-repeat':'no-repeat',
        'background-position':'right top',
        'padding-right':'15px'
    });
}
</script>

</head>
<body>
<h1>Client Side Encryption Demo</h1>
<form method="post" action="">
    <div><label>First Name:</label><input type="text" id="firstName" name="firstName"/></div>
    <div><label>Last Name:</label><input type="text" id="lastName" name="lastName"/></div>
    <div><label>Phone Number:</label><input type="text" name="phone" id="phone"
encrypted="client"/></div>
    <div><label>Address:</label><input type="text" encrypted="client" id="address"
name="address"/></div>
    <div><label>SS#:</label><input name="ssn" id="ssn" encrypted="client"/></div>
    <div><label>Notes:</label><textarea name="notes" id="notes" rows="3" cols="25"
encrypted="client"></textarea></div>
    <div><input type="submit" id="submitButton" value="Submit"/></div>
</form>
<div>
    This demo works best in IE8+, Firefox 3.5+, Safari 4.0+, Chrome 2.0+ & Opera 10.5+.
</div>
</body>
</html>

```