1. Introduction

This document proposes a trusted browser security model intended to secure the mutual automated consent between the end-user and the content provider before allowing a web browser to engage the services of an arbitrary third-party add-on, extension or service. Properly implemented, this proposed security model would create a standardized API across all browsers to further the security of e-commerce and other on-line transactions.
As the world moves away from the traditional desktop computer and becomes more dependent upon web applications, it is critical that the web browser through which these applications may be accessed provide robust security mechanisms which will protect sensitive data and systems from malicious parties [See Wadlow and Gorelik]. Currently there exists a considerable body of work (such as the standards describing SSL/TLS) for client and server authentication as well as the encryption of traffic between client and server. However, there exists no similar standards for securing information prior to its encryption by the SSL/TLS mechanisms. The following pages demonstrate the problems and proposed solution, which if standardized will allow end users and network administrators to establish browser security levels, and further will allow web application administrators and developers to implement server-side security levels. Accordingly, during web application operations, under this proposed model, the browser and web server would be forced to first agree that the two agree on a mutually acceptable security level before the web application is allowed to function. This framework is called "Trusted Browser Security Architecture" (TBSA).

2. The Problem, Defined

AUTHOR’S NOTE:

The problem described in the following section is a well-known attack and vulnerability that has existed for many years. The public discussion of this issue is therefore considered to be of very little value to the computer criminal community. It is the opinion of the author that the following public discussion of the issue is the best and only recourse to ensure the problem is resolved once and for all, as years of silence has not been a source of solutions.

For purposes of illustration, the author will describe a scenario using the Google Chrome web browser (5.0.375.99 beta). However the problem described can be reproduced on all major web browsers, including Internet Explorer, Mozilla Firefox, and Apple Safari. Additionally the problem is demonstrated through a simple manual operation. For ethical and legal reasons, the code for a full exploit is not made public in this document, as the author does not wish to assist malicious parties by providing a "how-to" guide for criminal conduct.

2.a. Browser Keylogging Demonstration

Assume there exists a web application whose login screen is rendered as depicted below in HTML:

```html
<html>
<head>
  <title>Target Site</title>
</head>
<body>
  <div>
    <form>
      Username:<input id="username" type="text" />
      <br />
      Password:<input id="password" type="password" />
      <br />
      <input id="submit" value="submit" type="submit" />
    </form>
  </div>
</body>
</html>
```
The site administrator considers the site to be sensitive and desires the strongest possible security. The web browser requires TLS with client authentication, and a client-side certificate has been installed on the end-user’s computer. This ensures that only the end-user’s computer can connect to the web server for the web application and that all communications between client and server are encrypted.

The end user installs a browser extension for use with another site’s content, and as required by that site, the end-user enables JavaScript. The extension, however, has been compromised and unknown to the end-user begins logging the end-user’s browser key strokes. The next time the end-user accesses the secured ‘target’ site, the extension covertly obtains the user’s credentials. While TLS would prevent the attacker from accessing the secured site from a remote system, the attacker is able to obtain credentials which can then be used to access the secured target site through the end-user’s system via some arbitrary code injected through the trojan browser extension.

Process

1. Open the Google Chrome web browser.
2. Navigate to some web page (see example HTML page provided above).
3. When page loads, right click page and select "Inspect Element".
4. Click "console" in the toolbar.
5. Within the console window, paste the following code:

```javascript
var o=document.getElementById('password');
o.onkeypress=function(e){console.log(e);}
```

6. Press <ENTER>.
7. Return to the form on the target page and select the password field.
8. Begin typing while observing the console window. All keystrokes should appear as captured ASCII codes in the console.

The above scenario demonstrates a simple key logger possible through most browsers. This exploit can be enhanced to use AJAX where the attacker will submit the keystrokes captured during operation to a remote web server. While manual operation requires physical or remote access to the system to inject the code in an open browser prior to the user’s interaction with a pre-loaded web-page, this exploit can be extended through the use of add-on or extension software used with many web browsers.

2.b. Discussion

Given the absence of a security mechanism preventing malicious activity by browser add-ons or extensions, and given the lack of code signing or other strategies to ensure that any previously trusted browser add-on or extension software has not been tainted by malicious code, it is possible for an attacker to first introduce malicious code into an existing browser add-on or extension or to introduce a Trojan add-on/extension from which the attack can occur.

Under the Trojan add-on/extension theory, the author constructed a simple browser extension for Mozilla Firefox. This extension was intended to synchronize passwords between a user’s many distributed computers, providing a centralized mechanism for password storage (i.e. the author’s own web server). This test-scenario resulted in an extension crafted to secure the end-user’s consent to capture and store user names and passwords, possibly blurring the lines of legality if put into actual practice. The author then conducted testing of this
extension using his own personal accounts on many popular websites, demonstrating the ability of a browser extension to inject the above key logging code into the target pages and to capture keystrokes. While the code injected by the test extension was much more complex than the above demonstration, it painted a picture that defeats any SSL/TLS or operating system level security measures and makes the SAAS model of web applications a foolhardy effort without further mechanisms within the web browser.

3. Trusted Browser Security Architecture (TBSA)

Simply blocking any and all browser add-ons and extensions is not a valid long-term solution to this problem. Extensions and add-ons include Adobe’s Reader and Flash Player as well as other media players and even those open source gizmos end-users find indispensable. While blocking extensions and add-ons is feasible in the corporate environment, where IT policy can over-ride end-user demand, in the home and small business arena, this option simply does not exist. Often through a lack of technical education or complacency, systems are left vulnerable and site developers for banking and other institutions are left without a valid recourse.

One strategy, proposed by Adam Barth, Adrienne Porter Felt, Prateek Saxena and Aron Boodman in "Protecting Browsers from Extension Vulnerabilities" (University of California at Berkley Technical Report No. UCB/EECS-2009-185) suggests a stronger browser security strategy that restricts the privileges of a browser extension, isolates extensions, separates privileges and otherwise increases the security of extensions in general. However, such strategies does nothing to allow the legitimate content-provider to be aware of or to consent to an otherwise user-approved extension running with his or her web application. Thus, where an end-user may for whatever reason believe his or her choice of browser extensions is ‘safe,’ or where the user may intentionally choose an extension in order to reverse engineer a web application, the site operator has absolutely no control to protect his or her intellectual property or the other users who rely upon this system.

The above issue can be resolved by implementing an API at the browser which is accessible from client-side programming languages, such as JavaScript, vbScript, etc. This API would allow a TBSA-aware web application to identify and negotiate the security level of the client-server relationship before delivering privileged content. Part of this strategy would also involve a browser integrity check, completing the integration of the client (i.e. web browser) and the application or web server.

-----------------------------------------------------------------
NOTE
TBSA is implemented as an API accessible via client-side programming rather than HTTP headers sent to the server. This is consistent with the current trend in web application development, where an initial application ‘seed’ can be sent to the client and expand dynamically based on client-side demand. This strategy is a more scalable and robust approach which gives web developers the greatest amount of freedom with the lowest learning curve while not compromising on security.
-----------------------------------------------------------------

3.b. Browser Integrity Check (BIC)

No security strategy will succeed without the verification of the platform’s integrity. Loss of integrity could mean that the ‘trusted platform’ is lying about its current state, rendering any further work absolutely worthless.
Because the web browser’s integrity is in question, it cannot be self-determined. However, because the server most likely exists outside the network perimeter and may not own or have rights to access lower-level operating system functionality (which in its own right would create other security issues), the browser integrity check must consist of both a client and server component, as described below:

3.b.1 Client-side Browser-Integrity Check (BIC)

The TBSA-compliant browser must provide a tbsa object whose methods are used for TBSA operations. These standardized methods ensure platform independence and a lower industry learning curve—two factors that increase the likelihood this standard will be adopted. The browser-integrity check is an example of this.

A TBSA-compliant browser must expose the following method for its most basic operation:

```python
<string> tbsa.browserIntegrityCheck(<string> serverCallbackURL);
```

When a web browser connects to a TBSA-aware web server, the web application calls `tbsa.browserIntegrityCheck()` through using some client-side programming language (e.g. JavaScript). This call must specify some server-side facility for the TBSA Browser-Integrity Check (i.e. `serverCallbackURL`). The browser will execute the browser-integrity check by performing an xmlHTTPrequest to the `serverCallbackURL` where it will submit its digital signature to the server:

```
+-----------------------------------------------+
|           Browser-Integrity Check             |
|                   HTTP/POST                   |
+-----------------------------------------------+
    POST <some_serverCallback_url> HTTP/1.0
    From: <user_specification>
    User-Agent: <browser-specfied agent>
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 32
    browserName=<some_browser_name>&
    digitalSignature=<browser_digital_signature>
```

When `tbsa.browserIntegrityCheck()` submits its digital signature to the `serverCallbackURL`, the server-side arbitrary tbsa module will evaluate the digital signature to determine if the browser is to be trusted. The browser must at no time know the outcome of this operation. Thus, the server-side browser-integrity check routine will return some TOKEN only the server can interpret. This token is returned in response to the xmlHTTPrequest, and the browser, unable to process or interpret the token, is expected to return the same to the code which first invoked `tbsa.browserIntegrityCheck()`.

The TOKEN received upon termination of `tbsa.browserIntegrityCheck()` must be cross-verified with the server. To do this, the code which first initiates the browser-integrity check must submit the TOKEN to its server-side component for cross-validation. In response, the server may reply with a simple boolean result or it may disconnect the session altogether.

The entire Browser-Integrity Check process is illustrated below:

```
+-------------------------------------------------------------+
```
TBSA Browser-Integrity Check
Client-side Example

<script type="text/javascript">
  if(tbsa){
    var bicToken=tbsa.browserIntegrityCheck("http://foo.com/tbsa-bic.php");
    if(myApp.tbsaBicXverify(bicToken)){
      /* Browser-Integrity Check token is verified */
    } else {
      /*Browser Integrity Check failed!* /
    }
  } else{
    /*Browser is not TBSA-compliant*/
  }
</script>

A NOTE ON BIC TOKEN CONSTRUCTION
The construction of BIC TOKENS is completely arbitrary, but any strategy should ensure the TOKEN is of reasonable complexity to ensure such a token cannot be easily guessed. For this reason the BIC TOKEN must not be some value the browser might determine by analyzing the web application. It is suggested that this value should be a sha512 hash of some random string of no less than 8192 bits (possibly including a timestamp.

3.b.2 The Server-side of Browser-Integrity Check (BIC)
The bulk of TBSA must be implemented at the client side. However, to realize the benefit of TBSA, some server-side body is needed. This server-side TBSA library must be platform-independent, and it must be easily applied in any of the several common server-side programming languages (e.g. ASP.NET, PHP, Python, Perl).

To implement the browser-integrity check at the server-side requires any TBSA server-side library to have two components. The first is tbsa.bicValidate() and the second is tbsa.bicXvalidate().

TBSA.bicValidate() is the module which is exposed as the serverCallBackURL used at the client side to obtain the BIC TOKEN.

This method must be aware of X.509 digital certificates and capable of validating the same. This may mean that tbsa.bicValidate could use an existing project such as openSSL to validate the certificate or it could apply some proprietary strategy.

The tbsa.bicValidate() method must also be capable of generating the random BIC TOKEN for return to the calling entity. This is
a simple matter of generating a random string of bits then performing a SHA512 hash operation on the same. However, this TOKEN must be cross-verified through tbsa.bicXvalidate(). Thus, the TOKEN must be stored for a reasonable amount of time in some server-side repository. It must also be able to age-out old TOKENS.

tbsa.bicXvalidate() is the counterpart to tbsa.bicValidate(). This method must be able to receive a TOKEN from the client-side and compare the TOKEN to the TOKENS registered by tbsa.bicValidate(). Further, to minimize the attack surface of registered TOKENS, the tbsa.bicXvalidate() method must remove any TOKEN it cross-validates. Removing cross-validated TOKENS conserves server-side storage also resources, but the primary purpose is to avoid increasing the probability that some browser may ‘guess’ a valid TOKEN and spoof the TBSA framework.

3.c. Signed Third-Party Extensions and Add-on Software

TBSA requires any ‘trusted’ browser, third-part extensions, or add-on software be digitally signed. While it should remain possible to install and execute unsigned extensions and add-on software for backward compatibility, only those software packages which are digitally signed should be considered ‘trusted.’

3.d. TBSA Extension Privilege Levels (tbsa.extensionPrivileges)

Browser extensions are the largest security risk in the current web application environment. This risk arises largely from the lack of any boundaries or limits on the extension-space. Web servers are unable to limit the browser add-on/extension or to verify their integrity if they are needed for the operation of the web application.

The TBSA client-side API provides tbsa.extensionPrivileges() and four security levels for the control of any browser extension:

```
+----------------------------------+
|        extensionSecurityLevel     |
+---+----------------------------------+
| 0 | Allow All Extensions             |
| 1 | Allow Signed Extensions          |
| 2 | Allow Specific Signed Extensions  |
| 3 | Disallow ALL extensions          |
+---+----------------------------------+
```

When a web browser connects to a web server, the browser extensions are assumed to operate at some privilege level set by browser design and user permissions. Where the web application is TBSA-aware and the browser is TBSA-compliant, the application may invoke the tbsa.extensionPrivileges() API call to either determine the current security level or to negotiate a higher security level.

```
<int> tbsa.extensionPrivileges( <int> intLevel,
<array> arrAllowedExtensions=NULL );
```

WARNING

Caution must be exercised when implementing the browser TBSA API. Calls to tbsa.extensionPrivileges must NEVER be allowed to negotiate a LOWER extension privilege level than the user’s minimum (default) privilege level.
Assume a web browser connects to some TBSA-aware web server and requests a web application. This web application would, presumably first perform a browser-integrity check, as described in the preceding sections, and then, if the browser’s integrity is verified, evaluate and negotiate any browser extension security level. This second security check may look something like the example below:

```html
<script type="text/javascript">
  if(tbsa.extensionPrivileges(1){
    /*We have passed! The extension privilege level is set to our minimum specification*/
  }else{
    alert('Browser cannot provide tbsa security level');
  }
</script>
```

In the above example, the web application instructs the browser to set a minimum browser extension security level of 1 ("Allow Signed Extensions"). The browser evaluates the request against its internal preference (i.e. the internal minimum privilege level). If possible the browser will elevate its privilege level to the requested level.

Regardless of the outcome of any internal operations, the browser will return its CURRENT browser extension security level. Thus, in the above example, where the browser is NOT TBSA-compliant, the call to the tbsa.extensionPrivileges() method would result in a zero or null value which would cause an alert to be raised informing the end user that the web application cannot be run on his/her platform.

However, assuming a browser processing the above script is TBSA-compliant and the browser’s internal default privilege level is equal to or higher than the input value (1), the above example would allow the web application to load. This presents a weakness for the above example that can be best addressed using a switch() statement and fall-through to dynamically negotiate content rendered based on TBSA compliance:

```html
<script type="text/javascript">
  switch(tbsa.extensionPrivileges(3)){
    case 0:
      /*Render security notification page.*/
      break;
    case 1:
      /*Allow only non-sensitive content.*/
      /*Fall-through*/
    case 2:
      /*Allow minimal content.*/
      /*Fall-through*/
    case 3:
      /*Allow sensitive content.*/
      break;
    default:
      /*Display error*/
  }
</script>
```

In the above example, the web application could respond dynamically to the browser’s TBSA compliance level. Assume, for example, that we have an travelling end-user who must remain connected to some corporate web application. The user most often uses a corporate laptop computer to engage the web application, but on one given trip the user’s only access is through a public computer at, say, some coffee shop. While on a trusted TBSA-compliant computer, the user logs in using only a username and password. But when the web application detects that the browser at
the coffee shop is not TBSA compliant, it executes a different user login mechanism whereby the user must provide not only username and password but a one-time passcode provided via automated telephone message or other means.

Another use of the above example is to disable specific features and content (such as a banking site which may not allow wire transfers or other operations without a TBSA-compliant browser).

The tbsa.extensionPrivileges() API call can request a browser to allow ANY extension to operate. It may also require that ONLY signed extensions be allowed to operate. At the opposite end of the spectrum, the web application may require that all signed or unsigned extensions be blocked from executing on the web site. This leverages existing mechanisms in most web browsers and will impose a small implementation overhead. The interesting feature of this API call is its ability to specify a set of known extensions that may run, banning all others. Thus, where the web application includes a structure such as the following, the browser may identify those extensions which it knows and trusts:

```javascript
<script type="text/javascript">
  var arrAllowableExt=[
    {name:'booExtension'},
    {name:'cooExtension'},
    {name:'dooExtension'},
    {name:'fooExtension'},
    {name:'gooExtension'},
    {name:'mooExtension'},
  ];
  switch(tbsa.extensionPrivileges( 2, arrAllowableExt)){
    case 3:
      /*
        Oops!
        We may have needed those trusted extensions to run our web application but the browser restricts all extensions. The system should notify someone.
      */
      break;
    case 2:
      /*
        Success!
        We requested specific extensions be allowed, and they were allowed. At this point we can proceed to verify our extension integrity and load the application.
      */
      break;
    default:
      /*
        Error!
        The browser declined our request. We are NOT trustworthy.
      */
      break;
  }
</script>
```

---

NOTE
When the web browser engages a TBSA-aware web application, it is presumed that the web developer will have designed the system to first implement all TBSA security checks before proceeding to render sensitive information to the browser. However, nothing should preclude the web
application from re-engaging the browser’s TBSA API to
reset or re-evaluate the TBSA privilege levels or integrity
checks. Thus, it is possible to elevate the required
security when performing login or other sensitive
operations, while relaxing these controls at other times.

3.e. Extension-Integrity Check (EIC)

TBSA integrity verification starts with the browser integrity check
described earlier and continues to the extensions themselves. Any
browser-extension which is TBSA compliant must provide a valid
digital signature. This signature may be used by browsers, the
operating system and by engaged web applications to verify the
integrity of the extension, using the TBSA API.

3.e.1. Client-Side Extension-Integrity Check (EIC)

At extension privilege level one (1) or higher, as described in the
above section, a browser extension may be considered ‘trusted’ and
therefore must be subject to verification processes. Unsigned
extensions are not subject to verification as they are ineligible
for ‘trusted’ status. The integrity of a ‘trusted’ extension may be
verified by a web application using the tbsa.validateExtension() API
call:

```c
<bool> tbsa.validateExtension( <string>strExtensionName,
<string>strServerCallbackURL

);```

As with tbsa.browserIntegrityCheck(), the browser cannot be trusted
to perform any client-side security operation under TBSA. Otherwise
it would be feasible for a malicious party to construct a client-
side exploit of the TBSA model. Good security principles dictates
that the perimeter must be well established between client and
server and all sensitive operations must be performed at the server
until the integrity of the COMPLETE platform (including extensions)
is verified. For this reason, a call to tbsa.validateExtension() must
specify a server callback URL with which the browser will call
the server to submit the digital certificate of a named extension.

The name of any extension passed to tbsa.validateExtension() must
match the name of the extension identified in the certificate, and
the server must be able to validate the certificate chain for the
certificate. Once the server-side module (identified by the call-
back URL) has verified the extension, the server will generate an
integrity-check verification TOKEN and return the same to the
browser. The browser will then return the TOKEN to the calling
structure, which must cross-verify the extension’s integrity by
submitting the TOKEN to the server’s TBSA cross-validation API. At
this point, the server may either respond with a boolean ‘true’
value indicating "extension valid," or the server may respond with a
boolean ‘false’ indicating the extension’s integrity check failed. A
web application’s server-side could also discretely flag the session
for termination and block any further communication with the client
browser, at the discretion of the web application’s developers.

3.e.2. Server-Side Extension Integrity Check (EIC)

The server side of an extension integrity check is facilitated by
two methods (tbsa.eicValidate() and tbsa.eicXvalidate()). The first
method is exposed indirectly via the server call back URL to
validate the digital signature and generate the validation TOKEN the
server will later cross-validate. When the TOKEN is returned to the
browser and the browser returns the TOKEN to its arbitrary calling structure, the calling structure must make the server-side tbsa.eicXvalidate() API call in order to complete the validation process.

At the server-side, when tbsa.eicValidate() is invoked, a TOKEN must be generated regardless of the validation outcome. The browser must not be able to discern from the TOKEN whether or not the integrity check has passed or failed. This protects the validation process from tipping its hand to the browser which (being suspect) could craft a passing validation token for cross validation or otherwise spoof a lesser strategy.

When tbsa.eicValidate() generates the validation TOKEN, the token must be returned to the browser as well as stored server-side for validation. In order to reduce attack surface on the TOKEN space, the eicValidate() method should also be expected to age-off any old TOKENs in its store. No TOKEN, therefore, should be allowed to remain in a token store for more than a few seconds, as cross-validation should be a relatively immediate process.

At the other side of the server-side validation process, the tbsa.eicXvalidate() method must be able to look up and compare the TOKEN received from the web application’s client-side code to any TOKENs located in the token store and to return a boolean ‘true’ if the TOKEN exists or otherwise respond as appropriate to the web application’s design if no match is found. Further, the cross-validation method must remove any TOKENs it has found matching a given value. This too reduces the attack surface and prevents a replay attack against the system.

3.f. Ajax Privileges

Ajax is a powerful tool which is essential in web application development. However, this also allows for many vulnerabilities the web application developer cannot control. Thus, though there may be some extension which meets the standard for ‘trusted’ status, once this extension has access to the DOM, it is possible for the extension to share information in an unexpected and unauthorized manner with some third party.

To avoid the abuse of Ajax by extensions or other code-injection strategy, TBSA implements Ajax privileges:

```c
<int> tbsa.ajaxPrivileges( <int> intLevel,
    <array> arrUrlList=NULL
    );
```

```
+-------------------------------+
|       ajaxPrivilegeLevels     |
+-------------------------------+
| 0 | Allow All                 |
| 1 | Allow Specific HTTP URLs  |
| 2 | Allow Specific HTTPS URLs |
| 3 | Block All URLs            |
```

The default ajax privilege level must be assigned by the end-user or browser design. No web application should be allowed to reduce this default privilege level. Web applications may only increase or identify the current privilege level.

When a TBSA-compliant browser engages a web application, the application may query the browser for or request a specific Ajax privilege level in
the same way the application may do so for the extension privilege level.

For example, a web application may include a block of code similar to the following:

```<script type="text/javascript">
    switch(tbsa.ajaxPrivileges(3)) {
        case 3:
            /*
            SUCCESS: We have blocked all third-party Ajax calls. This would allow the application to only talk to its own server(s), as determined by FQDN.
            */
            case 2:
                /*
                ALLOW SPECIFIC HTTPS:
                The browser has indicated it cannot block all third-party URLs for some reason, but it can limit access to a few specific URLs over HTTPS. If none were identified, this may be the same as the requested level 3, but this should not be trusted to be the case.
                */
            case 1:
                /*
                ALLOW SPECIFIC HTTP:
                The browser has indicated that it can only restrict Ajax to specific URLs using either HTTP or HTTPS. If no URLs were identified, this may have the same effect as level 3, but this should not be assumed to be the case.
                */
            default:
                /*
                OOPS!
                The browser has declined to honor the TBSA Ajax privileges request or it is unable to do so. In either case, the effect is the same. No restrictions on Ajax are in place at this time.
                */
    }
</script>`
```

In the above example, we use a switch() ladder at the client side to process the TBSA Ajax negotiation. Assuming we wish to lock down all third-party Ajax transactions, and the browser responds with any value less than our requested level 3 lock-down on Ajax, our web application should respond by refusing to proceed.

Alternatively, if the web application is accessed by a browser but refuses to lock-down the ajax privileges as we have requested, the application could show an error message or warning that notifies the
end user of the problem and asks that they agree to proceed (thereby making an informed decision).

Content could be restricted based on the above script given a browser’s response in order to minimize the risk to security while not completely disabling a customer/user base.

When restricting Ajax privileges to specific URLs, a call to tbsa.ajaxPrivileges() must be given a one-dimensional array of strings that describes URL of every allowed site. TBSA must also allow the dynamic reconfiguration of the Ajax privileges. Thus, a web application could begin with a complete ajax lock-down and then slowly introduce allowed URLs as the site requires, minimizing the attack surface at any given time.

3.g. Password Encryption

The example in the earlier problem statement involved stealing a value from some password field on a web page using JavaScript code injection. While the other measures specified in this document could prevent many attacks of this nature, password information is nonetheless stored in cleartext. A TBSA-compliant browser should encrypt this information as it is input by the end-user.

To facilitate inline password encryption, TBSA-compliant browsers must provide the following API calls:

<bool> tbsa.passwordEncryption(<string> publicKey);

The server can, therefore, configure the means by which the browser will protect password fields. The server may set an arbitrary public key for use with the AES encryption algorithm using the above method. When a field is specified in some website using the HTML <input /> tag with a field type of "password," a TBSA-compliant browser should ensure the field value is encrypted as key strokes are encountered.

Because the server is specifying the public key of an encryption scheme the browser may only encrypt the password information without any risk of the cryptographic system being compromised by some third party. Also because the server is establishing the public key cryptography parameters, the browser must only perform the encrypt operation then submit the information to the server, where it can be decrypted.

4. Security Considerations

There do not appear to be any security considerations associated with the Trusted Browser Security Architecture at this time, other than those expressed elsewhere in this document. The TBSA is intended as a loosely defined framework of client-server interactions, which when implemented by specific web browser developers/vendors may reveal implementation-specific vulnerabilities or security concerns that are best addressed on a case-by-case basis.

5. IANA Considerations

There are no anticipated IANA Considerations. The subject discussed in this document do not alter the specifications for any internet identifiers. The proposed TBSA is intended to operate at the application layer of the OSI reference model, and has no impact on the other layers of said model.

6. Conclusion

The solution proposed herein is intended to be backward compatible and to impose the minimal administrative overhead and industry investment
to address the described problem. For this reason, no new protocols are needed. Nor are any server-side changes required, other than the implementation of TBSA-aware code within the web applications. The key to addressing the described vulnerability is to implement TBSA within all browsers and to enforce the TBSA at the server-side through the use of TBSA-aware code within those applications which require the additional security.

7. Contributors

8. Acknowledgements

The following persons are acknowledged for their feedback and peer-review of this document prior to submission:

Tristan Slominski
Daniel Molina Wegener(dmw@coder.cl)
Phil Salemi (phil@firelytics.com)
Keith Chadwick (keith.chadwick@magma.ca)

9. Appendicies

There are no appendicies at this time.

10. References


11. Author’s Address

Sam Caldwell
POBox 2401
Round Rock, Texas 78680
512.963.7805
mail@samcaldwell.net
http://samcaldwell.net
skype: x684867

12. IPR Boilerplate