BetterAuth: Web Authentication Revisited*

Martin Johns
SAP Research
martin.johns@sap.com

Sebastian Lekies
SAP Research
sebastian.lekies@sap.com

Bastian Braun
University of Passau
bb@sec.uni-passau.de

Benjamin Flesch
SAP Research
benjamin.flesch@sap.com

ABSTRACT

This paper presents "BetterAuth", an authentication protocol for Web applications. Its design is based on the experiences of two decades with the Web. BetterAuth addresses existing attacks on Web authentication, ranging from network attacks to Cross-site Request Forgery up to Phishing. Furthermore, the protocol can be realized completely in standard JavaScript. This allows Web applications an early adoption, even in a situation with limited browser support.

1. INTRODUCTION

1.1 Motivation

The current state of password-based authentication on the Web is a mess. If used in its default configuration without additional protection measures, today's Web authentication almost appears to be an exercise in demonstrating how an authentication process should *not* be realized, showcasing severe flaws, such as, sending the password in cleartext over the wire, allowing untrusted parties to create arbitrary authenticated requests, or exposing the authentication credentials to potentially malicious code. While there have been first stabs in the direction of improving Web-based password authentication, previous approaches expose at least one of the following problems:

Web authentication differs from most other authentication scenarios: It exposes many characteristics that resemble properties from security protocols. However, it lacks a security protocol's rigorous enforcement of message sequence and integrity, resulting, for instance, in enabling the insertion of messages in authenticated workflows via Cross-site Request Forgery. Hence, proposals that approach Web authentication purely from a protocol perspective are in danger of solving only a subset of the problems and missing issues that result from the versatile and fragile nature of Web interaction.

*This work was in parts supported by the EU Projects STREWS (FP7-318097) and WebSand (FP7-256964).

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Furthermore, the vast majority of proposed improvements require fundamental changes both in the browser as well as in the client/server interaction. Hence, without Web browser support Web applications cannot benefit from the potential security benefits. This leads to a chicken/egg problem, as there is no early adopter path for motivated developers, which in turn could encourage the browser vendors to natively implement the mechanism.

In consequence, the basic process of password authentication on the Web has not significantly changed since the day in which the `type="password"` attribute was introduced to HTML.

1.2 Contribution & Organisation

In this paper, we propose BetterAuth, a password-based authentication scheme that is tailored to fit the Web's security requirements and mitigate the flaws of the current scheme. Our approach has the following properties:

- Unlike related approaches [42, 40, 1, 2, 8, 37], BetterAuth spans the full authentication lifecycle, consisting of both the initial authentication process and the ongoing authentication tracking. This allows both a lightweight, consistent design as well as robust, end-to-end security guarantees.

- Furthermore, BetterAuth is secure by default. The developer does not need to enable security properties explicitly. Instead, all security goals are met due to inherent properties of the scheme. In consequence, in its default state, BetterAuth transparently addresses many weaknesses of the established approach, including password sniffing, session credential theft, session fixation, and cross-site request forgery.

- Finally, even while being suited to be adopted as a native capability of Web browsers, BetterAuth can be implemented completely in standard JavaScript. This enables sites to use the scheme today without having to wait for the browser vendors to catch up. This potentially enables a viable, transitional phase, in which only a subset of deployed Web browsers support the scheme natively.

Organisation: The remainder of the paper is structured as follows: First, we summarize the current state of Web-based password authentication, both from the attacker as well as the developer's point of view (Sec. 2). Then, we describe BetterAuth, our improved authentication scheme

(Sec. 3) and report on our experiences in practically implementing the protocol (Sec. 4). An evaluation on security, performance, and limitations is given in Sec. 5. Before we conclude in Sec. 7, we discus related work (Sec. 6).

2. THE CURRENT STATE OF WEB-BASED PASSWORD AUTHENTICATION

The basic process of authenticating against Web applications has not changed significantly since the early days of the Web. In the following sections, we show how the current state of Web authentication came to be. First, we discuss the bare-bones authentication mechanism that is in use by the vast majority of all existing Web applications (see Sec. 2.1). Please note, that in this description, we omit all potential security measures. We simply show how Web authentication would look like, if implemented as is and how little security is provided by default. Then, in Sections 2.2 and 2.3, we revisit attacks on Web authentication and the countermeasures which were introduced to mitigate these threats.

Also, please note, that for the remainder of this paper, we restrict the discussion to password-based authentication, and in this respect, even further to the well established practice of form-based authentication (see Sec. 2.1), as virtually all professional Web applications utilize this method.

2.1 The Basics of Web Authentication and Authentication Tracking

The Web authentication process consists of two steps: First, the initial authentication, in which the user provides his user ID and password to the application's server-side. Then, the authenticated state of the user is maintained over the series of following HTTP request/response pairs. The next two sections will explore these two processes.

Initial authentication: In form-based authentication, the user's ID and password are communicated using HTML forms. After the user has entered his credentials, he submits the form. This causes the Web browser to create an HTTP request, which carries the values in the form of GET or POST parameters. In particular, this implies that the password is sent in clear-text to the server. The server compares the submitted user ID and password with its internal records. If the password and ID match with one of its records, the authentication process succeeds and the user's session is promoted to an authenticated state.

Authentication tracking: HTTP is a stateless protocol. Therefore, there is no protocol-level mechanism to promote a usage session into an authenticated state, as there is no inherent session concept. In consequence, application-layer measures for session and authentication tracking had to be introduced. The dominant method to maintain an authenticated state over a series of HTTP requests is to use HTTP cookies for this purpose. An HTTP cookie is a value that is set by a Web server for the Web server's domain. The value is stored by the browser. From this point on, all further requests that are sent to the server's domain carry the cookie value automatically, via the `Cookie`-header. To implement authentication tracking, the Web server sends a cookie to the browser, which signifies the authenticated state of this client. All further requests which are received by the server carrying this cookie value are regarded as being authenticated under the user's identity. Hence, the cookie value is de facto the user's authentication credential. Again, as with the password, this credential is communicated in cleartext. NB: Instead of setting a new cookie, the server could also promote an already existing session identifier (SID) cookie into an authenticated state, thus, making this SID the user's credential.

2.2 Fixing Web Authentication: A History of Band-Aid Solutions and Additive Design

In this section, we briefly revisit documented classes of Web attacks that target either the initial authentication or the authentication tracking process. In addition, we discuss the protective measures that have to be taken by the application developer to mitigate the respective threat.

2.2.1 Network-Based Attacks

As already mentioned in Sec. 2.1, both the user's password as well as the authenticator cookie are communicated in cleartext to the server. This opens the communication to various network-level attacks:

For one, every party that is able to observe the network traffic between the browser and the server can simply sniff the password or cookie value and abuse these credentials under the identity of the user. Furthermore, parties with direct access to the network link can also launch man-in-the-middle attacks, which allows the dynamic modification of HTTP requests and responses.

To counter these threats, the SSL/TLS protocol was introduced, which provides end-to-end confidentiality and integrity guarantees on top of TCP, making the sniffing of authentication credentials infeasible. Furthermore, SSL/TLS provides a PKI-based scheme to prove the server's identity to the user. This way, attempted man-in-the-middle attacks can be mitigated (as long as the user does not choose to ignore the warning dialogues).

SSL Stripping: Most Web applications serve content both encrypted, via HTTPS, as well as unencrypted, via HTTP. Unfortunately, if the user does not explicitly specify the protocol when he accesses a Web page, browsers default to HTTP. In consequence, in the majority of all cases, the first HTTP request to a server is sent via plain HTTP. This opens a loophole for a network-based man-in-the-middle attacker – the so-called SSL Stripping attacks [26].For this first request, an end-to-end SSL/TLS connection has not been established yet. Thus, the attacker can set himself in between the browser and the server and modify the server's responses. This way, even if the server requires HTTPS for certain operations and tries to redirect the browser accordingly, the attacker can simply remove these redirection attempts from the server's responses, before they reach the client. The client is forced to indefinitely communicate unencrypted.

To combat this problem, the `HSTS` HTTP response header [17] was created. This header tells the browser that from now on for a defined time period, all communication with the server shall be conducted using HTTPS. Under the assumption that the first connection to the server has been done using an attacker-free network path, from that point on the browser will reliably and exclusively use HTTPS to communicate with this server. This way, SSL stripping attempts are made impossible.

Further issues with SSL/TLS: The recent past has shown, that the current state of SSL/TLS is not fully bullet

	Transport	HTTP	Cookie	App
SSL/TLS	X			
HSTS		X		
HTTPonly			X	
Anti CSRF		(X^a)		X
Session Fix.	(X^b)		(X^c)	X
Anti Framing		X^d		X^e

a: Origin header, *b*: Origin-Bound Certs (exp.),
c: Origin flag (exp.), *d*: X-Frame options, *e*: JS-framebuster

Table 1: Overview of countermeasures and their respective implementation levels

proof. For one, the security of HTTPS-based communication heavily relies on the security policies and practice of the Certification Authorities (CAs), that issue the root certificates which are included in Web browsers by default. However, issues in that domain have been reported repeatedly, e.g., unlimited RA certificates have been issued [16] and the internal systems of several CA's have been compromised [10, 9]. As the CA system and its security is out of reach of the application's developers and operators, the current approach offers severely limited options to mitigate such threats.

2.2.2 Issues Related to Cookie-Based Authentication Tracking

As discussed above, after the initial authentication process, the cookie value becomes the user's authentication credential. However, HTTP cookies have not been designed with security in mind and were never intended to be used for this purpose.

Session hijacking through cookie theft: For one, based on the fact that the existence of the cookie value in a request suffices that the request is recognized to be authenticated, every party that can obtain this value is able to send arbitrary authenticated requests under the identity of the user. As, by default, the cookie value is sent in cleartext, every party with access to the network can sniff the value for future abuse. While SSL/TLS protects against this threat, many sites only protect the login page with SSL/TLS and then revert back to plain HTTP [19], leaving the cookie exposed.

Even in the existence of an uncompromised SSL/TLS connection, the cookie is readable by default through JavaScript via the `document.cookie` property. Hence, a simple Cross-site Scripting (XSS) vulnerability allows to leak the cookie's value to the adversary. To counter this threat, browser vendors introduced the `HTTPonly`-flag [28], which hides the cookie value from JavaScript. This flag has to be set explicitly by the developer to mark the authentication cookie.

Session Fixation: The `HTTPonly`-flag only prevents read access to the cookie value. However, an attacker is still able to set or overwrite cookie values. Hence, if he is able to set cookies for an attacked domain in the user's browser, he can launch a session fixation attack in which he tricks the application to reuse a value controlled by the attacker as the user's authentication token. Possible scenarios, in which attackers are able to set cookies for foreign domains include XSS, HTTP header injection [23], or insecure subdomains [22].

While this problem is partially addressed with currently experimental browser features [6, 2], the only reliable way for an application to mitigate this attack, is to renew the cookie's value each time the authorization level of the user changes [21].

Cross-site Request Forgery By default, the browser attaches all cookie values that belong to a given origin to every outgoing HTTP request to the corresponding site. However, due to the hypertext background of the Web, several HTTP-tags, such as `img`, `script`, or `iframe`, have the inherent ability to create cross-domain HTTP requests. Regardless of the actual origin of these elements, the browser attaches the target domain's cookies to all HTTP requests that are created this way. This circumstance leads to an attack vector known as Cross-site Request Forgery (CSRF): It is possible for any Web site which is rendered in the user's browser to send authenticated HTTP requests to all other Web sites, which currently maintain an authentication context with the browser.

To prevent third parties abusing this capability to initiate state-changing actions under the user's identity, the developer has to protect all sensitive interfaces of his application. This can be done either using secret nonces [33] or through strict checking of the `origin` request header [3].

Clickjacking: While being only partially related to authentication tracking, Clickjacking [15] (also known as "UI Redressing") is a class of attacks in the CSRF family. Clickjacking exploits the fact, that due to the cookie rules, foreign sites can load authenticated, crossdomain content into `iframes`. Using cascading style sheets, these iframes can be hidden from the user (e.g, by making them completely transparent) and, thus, the user can be tricked to interact with them via clicks or drag'n'drop.

To protect users from such attacks, the developer has to utilize JavaScript framebusting code [35] or the `X-Frame-Options` response header [27].

2.2.3 Phishing

A further serious threat is known by the term "Phishing" [30]. Phishing attacks aim to steal the user's password through simple decoy: A site under the control of the attacker imitates domain name and design of the target Web site. The user is tricked to enter his password into the forged site under the assumption that he interacts with the legitimate application. As HTTP transports the password in cleartext to the communication partner, the attacker is able to obtain and abuse it. A similar approach is followed by "Pharming" [39], a variant of phishing, which utilizes compromised DNS responses.

Due to its social engineering component, there is no straight forward technical solution to combat phishing, as long as the passwords are still sent over the wire. To mitigate the threat, browsers currently check visited URLs for known phishing sites and warn if such a page is accessed [14].

2.3 Summary

To sum the up the previous sections: The current praxis of Web application authentication and authentication tracking is secure if (and only if) the following holds:

- The password is transmitted over an uncompromised SSL/TLS connection in which the authenticity of the Web server has been verified. This requires among others robust defense against SSL-stripping attacks [26], e.g., utilizing the `HSTS` HTTP response header [17].

- All further requests, both belonging to the current session as well as all future sessions, are transmitted over an uncompromised SSL/TLS connection, as long as the authenticated cookie is valid.
- The authenticated cookie is secured against JavaScript read access by the HTTPonly cookie attribute [28].
- The value of the authenticated cookie is changed every time the authorization level of the user changes to combat potential session fixation vulnerabilities [21].
- State-changing interfaces are secured against CSRF using server-side checking of security nonces [33] or strict enforcement of matching `origin` HTTP response header [3].
- UI redressing attacks are avoided by framing prevention [27, 35].

All these measures have to be explicitly introduced and are realized at different positions and abstraction levels within the application architecture, spanning from securing the low-level transport layer via SSL to application layer anti-CSRF prevention (see Table 1 for an overview). Furthermore, even mitigation measures that are positioned at the same level within the application architecture often have to be implemented at separate places in the application's code.

And even if all these factors hold, the basic interaction pattern is still susceptible to phishing attacks, as the current scheme requires sending the password to the server as part of each login process.

3. PROTOCOL DESIGN

As discussed above, the current state exposes numerous security shortcomings. In this section, we present BetterAuth, an improved password scheme which is tailored to the Web's inherent characteristics and addresses the identified problems of the current scheme.

3.1 Design goals

Before we explain the technical aspects of BetterAuth, we briefly state our design goals:

Secure by default: BetterAuth is designed to mitigate the weaknesses of the current approach (see Sec. 2). In particular, these security goals are realized without explicit enabling steps by the developer.

No mandatory reliance on non-existing browser features: BetterAuth is designed in a fashion that allows an implementation for today's browsers. This allows an immediate deployment without the need to wait for browser vendors to implement native support.

No security regression: Regardless of the form of implementation (browser-based or pure JavaScript), BetterAuth has to be at least as secure as the current approach. This means a (re-)introduction of security problems, which are not currently present, is not acceptable.

3.2 High-level overview

Our proposed scheme consists of two steps, implemented as subprotocols:

An *initial mutual authentication protocol* with integrated key negotiation: The browser and the server both prove their knowledge of the password and jointly generate a per-session, shared secret which is used for further authentication tracking.

And an *authentication tracking scheme* which is based on request signing: Every further request from the browser to the server is signed using the freshly generated shared secret, if the request satisfies certain criteria (see Sec. 3.5 for details). Only requests with such a signature are regarded by the server as authenticated.

In the following sections, we give details on the realization of the two subprotocols.

3.3 Initial mutual authentication

As motivated above, one of BetterAuth's pillars is a mutual authentication step, which results in a shared cryptographic session key. Such mutual authentication schemes have received considerable attention in the past. In the given scenario both parties already share a textual secret (i.e., the password). Hence, a suiting choice for this task is a member of the password authenticated key exchange (PAKE) family [12]. PAKE protocols utilize well established cryptographic building blocks, such as the Diffie-Hellman key creation, and protect the communication against active network attackers using the pre-shared password.

While various protocols match our requirements, we selected [32] for our implementation, a scheme which is currently under active standardization by the IETF and, thus, has the potential for future adoption by the browser vendors. The protocol works as follows (see Figs. 1 and 2):

1. *Initial Handshake:* The browser sends a request targeted at the restricted resource. Along with this request, it sends the user's ID (*UID*, e.g., the user name). This causes the server to create the server-side partial key (*SPK*) for the Diffie-Hellmann key generation. The value is encrypted with the password[1], which has been set for the given UID.
2. *Key exchange:* The encrypted *SPK* is sent back to the browser as part of a 401 response. The browser creates the client-side Diffie-Hellmann partial key (*BPK*). The browser is now able to calculate the session key *SSK* using *SPK* and *BPK*. In addition *BPK* is encrypted with the password and added to the next request to the server.
3. *Mutual authentication:* The browser signs (see Sec. 3.4) the *BPK* carrying request using *SSK*. The server receives the request, calculates *SSK* himself and verifies the request signature. As the browser can only correctly compute *SSK*, if it knows the password, the correctness of the signature is used as authentication proof by the server. Hence, the server sends the restricted resource to the browser. Furthermore, the server also signs the response using *SSK*, to let the browser verify the server's knowledge of the password.

3.4 Request Signing

After the first protocol step has concluded successfully, both parties share a fresh symmetric key *SSK*, which from now on will serve as the basis for authentication tracking. Our authentication tracking mechanism is realized by HMACs [24], a well established Message Authentication Code scheme which utilizes cryptographic hash functions.

The client attaches an HMAC-based signature to all further requests to the server which satisfy the criteria given in Sec. 3.5, closely mimicking the current practice of automatically adding cookie headers to outgoing requests. For GET requests, the URL in a normalized form and selected request

[1]NB: This step can also be done with salted passwords.

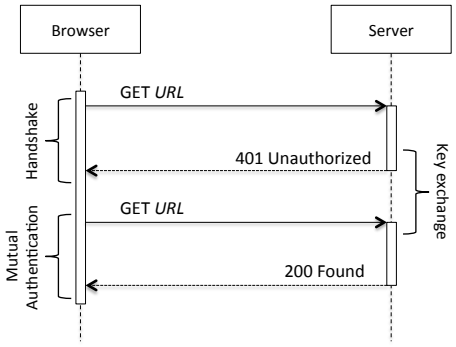

Figure 1: Initial auth. (HTTP communication)

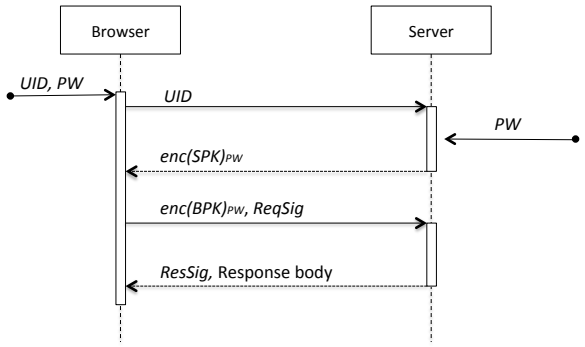

Figure 2: Initial auth. (cryptographic values)

headers are signed, for POST requests, also the POST parameters are included in the signature. Only requests, for which the server can successfully validate the correctness of the HMAC are recognized to be properly authenticated. This way, both the authenticity as well as the integrity of the received requests are ensured.

3.5 Context-Dependent Authentication

As discussed in Sec. 2.2.2, several security problems - most notably Cross-Site Request Forgery - are caused by the fact that currently all requests that originate from an authenticated browser are automatically equipped with the authentication credentials, i.e., the authentication cookies.

Our approach breaks from this troublesome behavior and instead only signs outgoing requests if the request's origin, i.e. the Web page which initiated the request, is already in an authenticated state with the server. Hence, we enforce *in-application authentication tracking*. All requests that are generated in the browser from outside of the Web application, i.e., from third party Web sites, are not signed and, in consequence, not treated as authenticated by the server.

3.6 Public Interfaces

While a strict enforcement of context-dependent authentication would provide robust security guarantees, it is too inflexible to cater to all existing usage patterns of the Web. For example, social Web bookmarking services, such as `delicious.com` provide one-click interfaces to add bookmarks from external pages. Such requests need to be processed in the user's authentication context, as they commit state changing actions to the user's data. However, as they are generated from outside of the Web application's authentication context, they would not receive a signature. Therefore, to enable such scenarios, our approach supports the declaration of *public interfaces*. Such a public interface is a URL for which the server opts in to receive authenticated requests, even if they originate from outside of the application's authentication context. A Web application's public interfaces, if they exist, are communicated to the browser during the initial key exchange using a simple policy format.

3.7 Resulting Authentication Tracking Logic

In consequence, the decision process which requests to sign works as follows:

1. *Test:* Check that the target URL of the request points to a domain, for which currently a valid BetterAuth authentication context exists. Such a context exists, if in the key storage a valid SSK_{app} key could be found, which is assigned to the domain value and that has not yet expired.
2. *Test:* Verify that the request is entitled to be signed. This means, check:
   - Was the request generated *within* the application? This means that the HTML element which was responsible for creating the request (e.g. hyperlink-navigation, form submission, or JavaScript action) is rendered within the browser in the origin of the authenticated application.
   - Or, is the target of the request contained in the applications's list of public interfaces?
3. *Action:* Normalize the request data (Method, URL, selected HTTP headers, request body) and create an HMAC signature using SSK_{app} as signature key.
4. *Action:* Attach the resulting request signature in an `Authorization` header to the request.

4. IMPLEMENTATION

In this section, we present our experiences on practically implementing BetterAuth. We created two different client-side implementations: For one, we built a Firefox browser extension in order to be able to assess how applications would behave, if the BetterAuth-protocol was implemented as a native part of the Web browser (see Sec. 4.1). Furthermore, we implemented BetterAuthcompletely in standard JavaScript (see Sec. 4.2). Using this implementation, Web applications could utilize the protocol during a transitional phase, in which only a subset of browsers support the approach natively.

4.1 Native Implementation

As mentioned above, we approximated a native browser implementation by realizing our approach in the form of a Firefox extension. The extension hooks itself as an observer into the browser's rendering process and monitors the outgoing HTTP requests. Whenever an authentication with a BetterAuth-enabled site is initiated or a request is sent to a domain for which an established BetterAuth authentication context exist, the extension becomes active.

4.1.1 Initial Authentication

If an HTML form is processed during rendering, which is marked with the custom attribute `data-purpose= "better-`

`auth"` the extension becomes active and the submission process of this form is intercepted: Before submitting the form, the username and password data is retrieved from the request data and used to initiate the BetterAuth-authentication handshake. After receiving the 401 response, the extension removes the password value from the request's data and submits the form.

4.1.2 Authentication Tracking

As discussed in Sec. 3.4, the authentication tracking mechanism mimics the behavior of Web browsers in respect to automatically adding cookie headers to requests that are targeted to the cookie's domain. The extension keeps track of currently active authentication contexts. Whenever a request is targeted towards a domain, for which such an authentication context exists, the extension verifies that the request originated from within this authenticated context or whether the target URL is listed in the application's set of public interfaces (see Sec. 3.5). If one of these conditions is satisfied, the extension transparently signs the outgoing request.

4.2 JavaScript Implementation

Our solution is designed in a fashion that allows to create a pure JavaScript fallback for browsers which do not support our authentication scheme natively. This way, a transitional phase can be supported, which allows developers to already use the mechanism without requiring to provide a separate authentication scheme for legacy browsers. In this section, we document the design of the JavaScript implementation of BetterAuth.

4.2.1 General Approach

The core of the transitional implementation is the replacement of native navigation operations, such as form submissions and page transitions, with a JavaScript initiated loading mechanism. This way, the initial authentication handshake can be executed and all further outgoing requests can be signed by JavaScript before they are sent to the server.

This approach is realized using four distinct elements: A dedicated form handling for the initial authentication (see Sec. 4.2.2), a request signing component (see Sec. 4.2.4), and a dedicated page loader object for pure page transitions (see Sec. 4.2.5). Furthermore, we utilize domain isolation to keep the key material out of reach of potentially untrusted JavaScript code (see Sec. 4.2.3).

4.2.2 Initial Authentication

Implementing the actual initial authentication handshake is straightforward: The BetterAuth-enabled HTML form executes a JavaScript function on form submission which conducts the key exchange handshake. For this purpose, the username and password values are read from the DOM elements. Using the XMLHttpRequest object, the script creates the OPTIONS request to the server's authentication interface. After receiving the server's encrypted Diffie-Hellman key and the optional password salt in the 401 response, JavaScript calculates the browser's Diffie-Hellman key and encrypts it with the password. In addition, after sending the key to the server, the script calculates the session signing key using the two key fragments.

4.2.3 Isolating the Secure Key Storage

As in Sec. 3.1 stated: It is unacceptable for any aspect of our technique to introduce security flaws which are not present in the current state. For this reason, we have to take measures to separate the key material from potentially untrusted JavaScript code.

An implementation of the authentication tracking process requires that the session signing key is handled by standard JavaScript functions. In consequence, a careless implementation would lead to a situation in which an XSS-attack could be used to steal this key and leak it to the adversary. Such an attack would be comparable to XSS-based cookie stealing, which can effectively be mitigated using the HTTPonly cookie flag. Hence, to avoid the introduction of security regression, we have to ensure that the key material is kept out of reach of untrusted parties.

To achieve this, we leverage the guarantees provided by the same-origin policy [34] and the postMessage API [38]: First, we introduce a separate subdomain, which is responsible to handle and store the signing key. This domain only contains static JavaScript dedicated to this task and nothing else. Based on this, we consider it to be feasible that the code running in this origin is well audited and XSS-free. An HTML document hosted on this subdomain which contains all necessary scripts, is included in the main application's pages using an invisible iframe.

The main application communicates with the key handling scripts on the secure subdomain using the postMessage API [38]: The postMessage API is a mechanism by which two browser documents are able to communicate across domain boundaries in a secure manner. A postMessage can be sent by calling the method `postMessage(message, targetOrigin)`. While the `message` attribute takes a string message, the `targetOrigin` represents the origin of the receiving page. In order to receive such a message the receiving page hast to register an event handler function for the `message` event. When receiving a message via the event handler function, the browser passes additional metadata to the receiving page. This data includes the origin of the sender. Hence, the postMessage API can be used to verify the authenticity of the sending page.

After a successful key exchange, the component responsible for the initial handshake passes the session signing key via postMessage to the secure subdomain. The receiving script stores the key, depending on its configured lifespan, either via the subdomain's sessionStorage or localStorage mechanism [11].

4.2.4 JavaScript-Based Request Signing

Following the initial authentication, all further requests have to carry a correct HMAC signature to be recognized as authenticated. In consequence, all outgoing requests have to be initiated via JavaScript. This is done by replacing hyperlink targets and form actions with JavaScript event handlers, which pass the target URL to the signing component of our implementation. This component normalizes the request's data and then passes it, using the browser's post-message API, to the secure iframe (see Lst. 1).

As mentioned above, a central feature of the post-message API is, that the origin domain of the incoming requests is communicated in an unspoofable fashion. Hence, the request signing script can verify that the call to the signing function was created within an authenticated context (see Sec. 3.5),

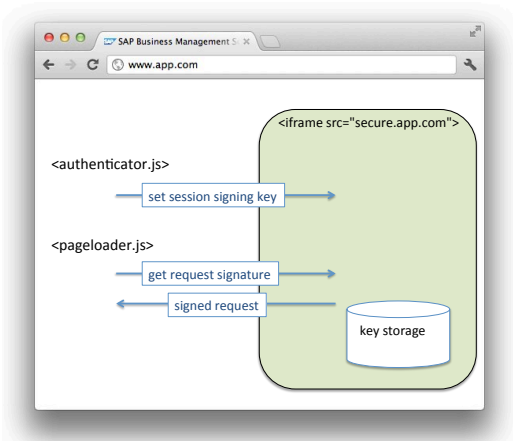

Figure 3: Domain isolated key handling

Code Listing 1 Request initation (simplified sketch)

```
<a href="#" onclick=
    "initAuthRequest('submod.jsp?a=b')">

<script>
window.addEventListener
    ("message", handleSignedRequest);
// Get request signature
function initAuthRequest(requestData){
    var rq = normalize(requestData);
    window.postMessage(rq,
                "http://secure.app.com")
   return false;
}
// receive signed request
function handleSignedRequest(event){
    if(event.origin ===
        "http://secure.app.com"){
 [attach request signature to request]
    }
}
</script>
```

and not by an untrusted third party which tries to abuse the functionality. Then, the signing component retrieves the signing key from localStorage, conducts the signing process, and passes the resulting values back to the main application, again using the postMessage functionality (see Lst. 2).

For apparent reasons, all page transitions and related request initiating actions of the main application have to utilize the request signing functionality. While for newly written applications, this won't cause a lot of effort, legacy applications have to be adapted to support the novel functionality. However, as discussed in [1], many applications can easily be adapted by traversing the application's pages DOM on load and patching the encountered links and forms to use the request signing functions. Alternatively, server-side rewriting of outgoing HTML could be utilized, modifying hyperlinks and form-actions to utilize JavaScript page navigation (see Lst. 1). Finally, for applications that mainly rely on AJAX driven client/server interaction, the request signing functionality can be introduced transparently replacing the XMLHttpRequest object with an object wrapper which implements the necessary actions.

Code Listing 2 Request signing code on the secure subdomain (simplified sketch)

```
window.addEventListener
        ("message", handleSignOrder);
// Create signed request
function handleSignOrder(event){
    if(authContext(event.origin)){
      var key = getSSK(event.origin)
      var sig = signReq(event.data, key)
      event.source.postMessage(sig,
                               event.origin)
    }
}
```

4.2.5 *Accessing Public Interfaces*

The final puzzle piece in the transitional implementation is a facility that enables external sites to navigate to the application's public interfaces (see Sec. 3.5). To recall, a public interface is a URL to which external sites are allowed to navigate in an authenticated state (e.g., for posting to social sharing sites).

For this purpose, we utilize a *pageloader* object: The page loader is a small JavaScript that is delivered by the application in case an unauthenticated request has been received for a URL which requires authentication and is contained in the application's set of public interfaces. The script is carried in the body of the initial 401 response during the key exchange handshake. In consequence, if such a response is received during a standard Web navigation process (opposed to the explicit authentication handshake executed by the native or transitional implementation), the page loader is executed in an otherwise blank HTML document.

The pageloader's source code is created dynamically by the server to contain the request's data which needs to be signed, in most cases mainly consisting of the original request's URL. The page loader dynamically includes the iframe to the secure subdomain and utilizes the standard request signing functionality of the implementation (see Lst. 1) to create a second, now authenticated request. The strict origin checking mechanism of the subdomain's signing interface robustly prevents potential abuse.

5. EVALUATION

5.1 Security Evaluation

In this section, we examine how capable BetterAuth is in mitigating the security threats (see Sec. 2).

Network-based attacks: At no point, passwords nor authentication tokens are transmitted over the network. Therefore, sniffing attacks are powerless. Also, due to the mutual authentication properties of the initial authentication, man-in-the-middle attacks are mitigated. However, please note, that BetterAuth only proves that the server indeed possesses the password. Furthermore, the security properties of BetterAuthdo not rely on the security of an underlying SSL/TSL connection. In consequence, SSL stripping attacks or CA breaches have no effect.

Issues related to cookie-based authentication tracking: There is no authentication cookie anymore, which could be stolen or manipulated. Hence, session hijacking and fixation attacks do not apply. Furthermore, CSRF attacks are

mitigated, as only *in-application* requests receive a signature, leading to a situation in which crossdomain requests are treated as unauthenticated by default. Only, if URLs are explicitly added to the list of public interfaces, the developer has to ensure, that crossdomain request to these URLs do not cause unwanted side effects. Finally, as we will discuss further in Sec. 5.3, Clickjacking attacks are partially addressed but still might occur.

Phishing: The password never leaves the browser. Hence, phishing attacks are bound to fail. However, this property only holds, if the password is entered only in BetterAuth-enabled input fields (see Sec. 5.3 for a further discussion of this limitation).

Limitations of the JavaScript implementation: Unlike a native implementation, the transitional implementation is susceptible to active man-in-the-middle attackers. The reason for this is, that the cryptographic components, which are executed in the secure subdomain's iframe are transported over the compromised network connection. Hence, the adversary could alter the transmitted source code in a fashion that leaks the session signing key or the user's password to the outside. Hence, at least the secure subdomain's content should be communicated via HTTPS.

5.2 Performance Evaluation

We don't expect a native implementation to cause considerable overhead. The utilized algorithms are in similar form already highly efficiently implemented both in browsers and servers as part of the SSL/TLS suite. Hence, the introduced overhead will be at most in the same range as overhead introduced by HTTPS communication. However, for the transitional implementation, the client-side component is implemented in pure JavaScript. Thus, a potential for noticeable overhead is given. Fortunately, in the last couple of years the browser vendors were inclined in an arms race on rapidly improving the performance of their JavaScript interpreters. To evaluate, how a JavaScript realization of the initial authentication would perform under realistic circumstances, we implemented the protocol as outlined in Sec. 4.2. For the cryptographic operations, we utilized the "Big Integer Library"[2] and the "Stanford Javascript Crypto Library (SJCL)"[3]. We benchmarked our implementation on three different machines running different operating systems each (Linux, Mac Os X, and Windows 7) and in total six browsers (see Tab. 2 for further details). The results of our benchmarking efforts can be obtained in Tab. 2. Among all configurations, the best performance could be observed with the Chrome browser, which reliably stayed below 300 ms, using a reasonable key length of 1024 Bit. The worst performance was exposed by Internet Explorer 9, which consumed in average 1314 ms for the same operations. Please keep in mind, that this overhead occurs only once during the whole process. The HMAC based authentication tracking can be implemented highly efficient and, thus, causes negligible performance effects.

5.3 Open Issues

The password entry field: BetterAuth provides strong protection against phishing attacks on the protocol level. However, this protection can be circumvented by the attacker on the GUI-level: As duly observed in [36, 12], if the user can be tricked into entering his password in non-BetterAuth form field, the attacker is still able to steal it. What is needed to close this hole is a visually unspoofable "trusted path" [43] from the user to a well isolated password handler within the browser. If such functionality is provided, further security guarantees in respect to the handling of the password data can be robustly introduced. Merely implementing such an approach is not a hard task on the engineering level, in parts it has already been done with the authentication dialogues for HTTP basic and digest authentication as well in various research prototypes. However, it is a major challenge in UI design. A right balance has to be found between the needs of Web UI designers and the ability of users to reliably recognize such "secure" entry forms.

Browser	D-H key length 768 Bit	1024 Bit	1536 Bit
Chromium/Linux[1]	116.7	261.1	876.6
Firefox/Linux[1]	182.6	426.8	1476.6
Chrome/Mac[2]	113.9	257.9	862.6
Safari/Mac[2]	405.6	942.5	3069.7
Chrome/Win 7[3]	127.2	281.9	932.7
IE 9/Win 7[3]	592.2	1314.6	4511.2

1: Acer Aspire, Ubuntu 11.10, Core i5, 2.53 GHz, 4GB RAM
2: MacBook Pro, Os X 10.7.2, Core i7, 2,2 GHz, 8GB RAM
3: ThinkPad T410s, Win7 Pro x64 SP1, Core i5 2,6 GHz, 4GB RAM

Table 2: JavaScript implementation performance (times in ms, averaged over ten runs)

Limited protection against Clickjacking: As motivated in Sec. 2.2.2, Clickjacking can be regarded as a class of vulnerabilities rooted in the current practice of authentication tracking. More precisely, Clickjacking is based on the adversary's capability to load cross-domain, authenticated GUI interfaces into the `iframe`. If BetterAuth is used without any configured public interfaces (see Sec. 3.5), this attack pattern would be infeasible, as no entry point into the application logic can be accessed from outside of an authenticated context. However, this protection ends as soon as public interfaces are added: In this case, the application probably offers a navigation path from the public interface to the targeted GUI interface. By tricking the user into multiple click-interactions with the disguised iframe, the attacker may be able to trick the user into unknowingly conduct this navigation. As all requests which originate from the public interface come from an authenticated context, they transparently receive request signatures, resulting in potential access of the attacker to the targeted Web GUI. Therefore, public interfaces should still be protected with anti-framing measures. Nonetheless, BetterAuth raises the bar of difficulty for Clickjacking attacks and with the set of public interfaces being limited and explicitly configured, full anti-framing protection is a straight forward task.

Replay attacks: If implemented in the from it is described in Sec. 3, the communication between browser and server would be susceptible to replay attacks from network attackers. We left handling of this issue out of the description for brevity and clarity reasons. However, adding replay protection to the authentication tracking process is straightforward, using a sliding window of monotonous growing nonces in the requests and limited state-keeping on the server-side.

[2] `http://leemon.com/crypto/BigInt.html`
[3] `http://crypto.stanford.edu/sjcl/`

6. RELATED WORK

Isolated security aspects of the Web authentication have received considerable attention, foremost the areas of phishing [8, 37, 41],cross-site scripting [25, 29, 20, 31]and CSRF [3, 33]. Due to the narrow focus of these works, we omit a detailled discussion. For the remainder of this section, we focus on password protocols and approaches that target Web authentication.

Password protocols: Bellovin and Merritt proposed the *Encrypted Key Exchange (EKE)* protocol that is based on pre-shared secretes, i.e. passwords, and secure against dictionary attacks [4, 5]. They put emphasis on considering which messages should be encrypted with the password without increasing the risk of offline, brute-force attacks. One drawback of this approach in modern web scenarios lies in the fact that the password has to be stored in cleartext on server side. Jablon proposed an improved approach that eliminates this need [18]. Wu proposed a modified version of EKE, called *Asymmetric Key Exchange (AKE)*, which is finally used to derive the *Secure Remote Password (SRP)* protocol [42]. It ceases to use symmetric cryptography and focuses on strong security properties with respect to leakage of server's user database or session keys. Steiner et al. describe the integration of a slightly modified version of Bellovin's and Merritt's approach [4], named *DH-EKE*, into TLS [40]. This way, they eliminate the need for a public key infrastructure. Due to mutual authentication, certificates become obsolete. Their *Secure Password-Based Cipher Suite for TLS* implements confidentiality and authenticity.

Web authentication: *SessionLock* [1] is closely related to our transitional JavaScript implementation of BetterAuth. The paper demonstrates how standard JavaScript can be used to substitute cookie-based authentication tracking with a browser-driven HMAC scheme. SessionLock does not protect against CSRF problems and does not handle the initial authentication step. [2] introduces browser authentication without user authentication: The browser, generates a self-signed certificate. This certificate must not contain any user-related information. A new certificate is issued by the client for every single server domain ("origin"). Session tracking can be secured by relating session cookies to the respective client certificate, hence, mitigating several of the cookie related threats, such as SID theft or session fixation. Finally, the most recent draft of HTTP/1.1 specification provides an extension of long-known HTTP Basic and Digest Authentication based on Challenge-Response Authentication [13], which prevents known eavesdropping attacks on the former HTTP authentication standards.

Chen et al. address the problem of cross-site attacks that occur while surfing sensitive and non-trustworthy websites at the same time in one browser [7]. Therefore, they isolate browser sessions which prevents cross-domain attacks. Same-domain attacks are out of scope of this approach. This feature is comparable with our context dependent authentication and public interfaces. The security level of *app isolation* is equivalent to surfing different websites using different browsers.

7. CONCLUSION

In this paper we presented BetterAuth. BetterAuth is a mutual Web authentication protocol that was designed to be secure by default, thus, freeing the developers and operators of Web applications from the need to counter potential threats at various heterogeneous places in the application's architecture, as it is required by the currently established approach. BetterAuth significantly improves the susceptibility of the authentication process to known threats, ranging from network attacks, over Cross-site Request Forgery, up to Phishing. Furthermore, the protocol was designed in a fashion that allows an implementation in standard JavaScript, enabling its deployment even in situation in which no widespread native browser support is present yet.

8. REFERENCES

[1] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 517–524, New York, NY, USA, 2008. ACM.

[2] D. Balfanz, D. Smetters, M. Upadhyay, and A. Barth. TLS Origin-Bound Certificates. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-obc-01`, Version 01, November 2011.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS'09*, 2009.

[4] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Oakland, CA, May 1992.

[5] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, Fairfax, VA, November 1993.

[6] A. Bortz, A. Barth, and A. Czeskis. Origin Cookies: Session Integrity for Web Applications. In *W2SP 2011*, 2011.

[7] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App Isolation: Get the Security of Multiple Browsers with Just One. In *18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[8] R. Dhamija and J. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Symposium On Usable Privacy and Security (SOUPS) 2005*, July 2005.

[9] P. Eckersley. How secure is HTTPS today? How often is it attacked? [online], `https://www.eff.org/deeplinks/2011/10/how-secure-https-today`, October 2011.

[10] P. Eckersley and J. Burns. The (Decentralized) SSL Observatory. Invited Talk, Usenix Security 2011, `http://static.usenix.org/events/sec11/tech/slides/eckersley.pdf`, August 2011.

[11] I. H. (Ed. Web Storage. W3C Candidate Recommendation, `http://www.w3.org/TR/webstorage/`, December 2011.

[12] J. Engler, C. Karlof, E. Shi, and D. Song. Is it too late for PAKE? In *Proceedings of W2SP*, 2009.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Y. Lafon, and

J. Reschke. HTTP/1.1, part 7: Authentication. [IETF draft], http://tools.ietf.org/html/draft-ietf-httpbis-p7-auth-18, Version 18, January 2012.

[14] Google. Safe Browsing for Firefox. [application], http://www.google.com/tools/firefox/safebrowsing/, (03/20/06), 2006.

[15] R. Hansen and J. Grossman. Clickjacking. [online], http://www.sectheory.com/clickjacking.htm, last accessed 02/13/12, August 2008.

[16] E. Henning. Trustwave issued a man-in-the-middle certificate. [online], http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate- 1429982.html, February 2012.

[17] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). [IETF draft], http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-04, Version 04, January 2012.

[18] D. Jablon. Extended Password Key Exchange Protocols Immune to Dictionary Attacks. *Enabling Technologies, IEEE International Workshops on*, 0:0248, 1997.

[19] C. Jackson and A. Barth. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *WWW 2008*, 2008.

[20] M. Johns. SessionSafe: Implementing XSS Immune Session Handling. In *European Symposium on Research in Computer Security (ESORICS 2006)*. Springer, September 2006.

[21] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable Protection Against Session Fixation Attacks. In *26th ACM Symposium on Applied Computing (SAC 2011)*. ACM, March 2011.

[22] D. Kaminsky. h0h0h0h0. Talk at the ToorCon Seattle Conference, http://seattle.toorcon.org/2008/conference.php?id=42, April 2008.

[23] A. Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Whitepaper, Sanctum Inc., http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf, March 2004.

[24] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, http://tools.ietf.org/html/rfc2104, February 1997.

[25] M. T. Louw and V. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland'09)*, May 2009.

[26] M. Marlinspike. New Tricks For Defeating SSL In Practice. Talk at the Black Hat DC conference, 2009.

[27] Microsoft. Ie8 security part vii: Clickjacking defenses, 2009.

[28] MSDN. Mitigating Cross-site Scripting With HTTP-only Cookies. [online], http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp, (01/23/06).

[29] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS 2009)*, 2009.

[30] J. Nelson and D. Jeske. Limits to Anti Phishing. In *Proceedings of the W3C Security and Usability Workshop*, 2006.

[31] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *3rd International Symposium on Engineering Secure Software and Systems (ESSoS '11)*, LNCS. Springer, February 2011.

[32] Y. Oiwa, H. Watanabe, H. Takagi, B. Kihara, T. Hayashi, and Y. Ioku. Mutual Authentication Protocol for HTTP. [IETF draft], http://tools.ietf.org/html/draft-oiwa-http-mutualauth-10, Version 10, October 2011.

[33] Open Web Application Security Project. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. [online], https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, accessed November 2011, 2010.

[34] J. Ruderman. The Same Origin Policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001.

[35] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[36] D. Sandler and D. S. Wallach. <input type="password"> must die! In *Web 2.0 Security and Privacy (W2SP)*. IEEE, May 2008.

[37] M. Sharifi, A. Saberi, M. Vahidi, and M. Zoroufi. A Zero Knowledge Password Proof Mutual Authentication Technique Against Real-Time Phishing Attacks. In P. D. McDaniel and S. K. Gupta, editors, *ICISS*, volume 4812 of *Lecture Notes in Computer Science*, pages 254–258. Springer, 2007.

[38] E. Shepherd. window.postmessage. [online], https://developer.mozilla.org/en/DOM/window.postMessage, last accessed 02/12/12, October 2011.

[39] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by Pharming. In *In Proceedings of Information and Communications Security (ICICS '07)*, number 4861 in LNCS, December 2007.

[40] M. Steiner, P. Buhler, T. Eirich, and M. Waidner. Secure Password-Based Cipher Suite for TLS. In *NDSS*, pages 134–157, 2001.

[41] M. Wu, R. C. Miller, and G. Little. Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In *Proceedings of the second symposium on Usable privacy and security (SOUPS 06)*, 2006.

[42] T. Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

[43] K.-P. Yee. User interaction design for secure systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, ICICS '02, pages 278–290, London, UK, UK, 2002. Springer-Verlag.