Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web

Michael Dietz
Rice University
mdietz@rice.edu

Alexei Czeskis
University of Washington
alexei@czeskis.com

Dirk Balfanz*
Google Inc.
balfanz@google.com

Dan S. Wallach
Rice University
dwallach@rice.edu

Abstract

Client authentication on the web has remained in the internet-equivalent of the stone ages for the last two decades. Instead of adopting modern public-key-based authentication mechanisms, we seem to be stuck with passwords and cookies.

In this paper, we propose to break this stalemate by presenting a fresh approach to public-key-based client authentication on the web. We describe a simple TLS extension that allows clients to establish strong authenticated channels with servers and to *bind* existing authentication tokens like HTTP cookies to such channels. This allows much of the existing infrastructure of the web to remain unchanged, while at the same time strengthening client authentication considerably against a wide range of attacks.

We implemented our system in Google Chrome and Google's web serving infrastructure, and provide a performance evaluation of this implementation.

1 Introduction

In the summer of 2011, several reports surfaced of attempted man-in-the-middle attacks against Google users who were primarily located in Iran. The Dutch certification authority DigiNotar had apparently issued certificates for *google.com* and other websites to entities not affiliated with the rightful owners of the domains in question[1]. Those entities were then able to pose as Google and other web entities and to eavesdrop on the communication between users' web browsers and the websites they were visiting. One of the pieces of data such eavesdroppers could have conceivably recorded were *authentication cookies*, meaning that the man-in-the-middle could have had full control over user accounts, even after the man-in-the-middle attack itself was over.

This attack should have never been possible: authenticating a client to a server while defeating man-in-the-middle attacks is theoretically a solved problem. Simply put, client and server can use an authenticated key agreement protocol to establish a secure permanent "channel." Once this channel is set up, a man-in-the-middle cannot "pry it open", even with stolen server certificates.

Unfortunately, this is *not* how authentication works on the web. We neither use sophisticated key agreement protocols, nor do we establish authenticated "channels." Instead, we send secrets directly from clients to servers with practically every request. We do this across all layers of the network stack. For example, to authenticate users, passwords are sent from clients to servers; SAML or OpenID assertions are sent from clients to servers in order to extend such user authentication from one website to another; and HTTP cookies are sent with every HTTP request after the initial user authentication in order to authenticate that HTTP request.

We call this pattern *bearer tokens*: the bearer of a token is granted access, regardless of the channel over which the token is presented, or who presented it[2].

Unfortunately, bearer tokens are susceptible to certain classes of attacks. Specifically, an adversary that manages to steal a bearer token from a legitimate user can impersonate that user to web services that require it. For different kinds of bearer tokens these attacks come in different flavors: passwords are usually obtained through phishing or keylogging, while cookie theft happens through man-in-the-browser malware (*e.g.,* Zeus [16]), cross site scripting attacks, or adversaries that manage to sniff the network or insert themselves into the network between the client and server [1, 7]).

The academic community, of course, has known of authentication mechanisms that avoid the weaknesses of

*The opinions expressed here are those of the authors and do not necessarily reflect the position of Google.

[1] It later turned out that the certificates had, in fact, been created fraudulently by attackers that had compromised DigiNotar.

[2] Bearer tokens, originally called "sparse capabilities" [25], were widely used in distributed systems, well before the web.

bearer tokens since before the dawn of the web. These mechanisms usually employ some form of public-key cryptography rather than a shared secret between client and server. Authentication protocols based on public-key cryptography have the benefit of not exposing secrets to the eavesdropper which could be used to impersonate the client to the server. Furthermore, when public/private key pairs are involved, the private key can be moved out of reach of thieving malware on the client, perhaps using a hardware Trusted Platform Module (TPM). While in theory this problem seems solved, in *practice* we have seen attempts to rid the web of bearer tokens gain near-zero traction [10] or fail outright [13].

In this paper, we present a fresh approach to using public-key mechanisms for strong authentication on the web. Faced with an immense global infrastructure of existing software, practices and network equipment, as well as users' expectations of how to interact with the web, we acknowledge that we cannot simply "reboot" the web with better (or simply different) authentication mechanisms. Instead, after engaging with various stakeholders in standards bodies, browser vendors, operators of large website, and the security, privacy and usability communities, we have developed a layered solution to the problem, each layer consisting of minor adjustments to existing mechanisms across the network stack.

The key contributions of this work are:

- We present a slight modification to TLS client authentication, which we call *TLS-OBC*. This new primitive is simple and powerful, allowing us to create strong TLS channels.
- We demonstrate how higher-layer protocols like HTTP, federation protocols, or even application-level user login can be hardened by "binding" tokens at those layers to the authenticated TLS channel.
- We describe our efforts in gaining community support for an IETF draft [2], as well as support from major browser vendors; both Google's Chrome and Mozilla's Firefox have begun to incorporate and test support for TLS-OBC.
- We present a detailed report on our client-side implementation in the open-source Chromium browser, and our server-side implementation inside the serving infrastructure of a large website.
- We give some insight into the process that led to the proposal as presented here, contrasting it with existing work and explaining real-world constraints, ranging from privacy expectations that need to be weighed against security requirements, to deployment issues in large datacenters.

Summary. The main idea of this work is easily explained: browsers use self-signed client certificates within TLS client authentication. These certificates are generated by the browser on-the-fly, as needed, and contain no user-identifying information. They merely serve as a foundation upon which to establish an authenticated *channel* that can be re-established later.

The browser generates a different certificate for every website to which it connects, thus defeating any cross-site user tracking. We therefore call these certificates *origin-bound certificates* (OBCs). This design choice also allows us to completely decouple certificate generation and use from the user interface; TLS-OBC client authentication allows the existing web user experience to remain the same, despite the changes under the hood.

Since the browser will consistently use the same client certificate when establishing a TLS connection with an origin, the website can "bind" authentication tokens (*e.g.,* HTTP cookies) to the OBC, thereby creating an authenticated channel. This is done by simply recording which client certificate should be used at the TLS layer when submitting the token (*i.e.,* cookie) back to the server. It is at *this* layer (in the cookie, not in the TLS certificate) that we establish user identity, just as it is usually done on the web today.

TLS-OBC's channel-binding mechanism prevents stolen tokens (*e.g.,* cookies) from being used over other TLS channels, thereby making them useless to token thieves, solving a large problem in today's web.

2 Threat Model

We consider a fairly broadly-scoped (and what we believe to be a real-world) threat model. Specifically, we assume that attackers are occasionally able to "pry open" TLS sessions and extract the enclosed sensitive data by exploiting a bug in the TLS system [22], mounting a man in the middle (MITM) attack through stolen server TLS certificates [1], or utilizing man-in-the-browser malware [16]. These attacks not only reveal potentially private data, but in today's web will actually allow attackers to impersonate and completely compromise user accounts by capturing and replaying users' authentication credentials (which, as we noted earlier, are usually in the form of bearer tokens). These attacks are neither theoretical nor purely academic; they *are* being employed by adversaries in the wild [24].

In this paper we focus on the TLS and HTTP layers of the protocol stack, and on protecting the authentication tokens at those layers—mostly HTTP cookies (but also identity assertions in federation protocols)—by binding them to the underlying authenticated TLS-OBC channel. We have a parallel effort under way to protect the application-layer user logins, but that is mostly outside the scope of this paper. To model this distinction, we consider two classes of attacker. The first class is an attacker that has managed to insert themselves as a MITM *during* the initial authentication step (when the user trades his username/password credentials for a cookie), or an attacker that steals user passwords through a database compromise or phishing attack. The second class of attacker is one that has inserted themself as a MITM *after* the initial user authentication step where credentials are traded for an authentication token. The first class of attacker is strictly stronger than the second class of attacker as a MITM that can directly access a user's credentials can trade them in for an authentication token at his leisure. While the second class of attacker, a MITM that can only steal the authentication token, has a smaller window of opportunity (the duration for which the cookie is valid) for access to the user's private information.

For the purposes of this paper, we choose to focus on the second class of attacker. In short, we assume that the user has already traded their username/password credentials to acquire an authentication token that will persist across subsequent connections. Our threat model allows for attackers to exploit MITM or eavesdropping attacks during any TLS handshake or session subsequent to the initial TLS connection to a given endpoint—including attacks that cause a user to re-authenticate as discussed in Section 4.3. Attacks that target user credentials during the initial TLS connection, rather than authentication tokens during subsequent TLS connections, are dealt with in a forthcoming report.

3 TLS-OBC

We propose a slightly modified version of traditional TLS client certificates, called Origin-Bound Certificates (OBCs), that will enable a number of useful applications (as discussed in Section 4).

3.1 Overview

Fundamentally, an Origin-Bound Certificate is a self-signed certificate that browsers use to perform TLS Client Authentication. Unlike normal certificates, and their use in TLS Client Authentication (see Section 8.1), OBCs do not require any interaction with the user. This property stems from the observation that since the browser generates and stores only one certificate per origin, it's always clear to the browser which certificate it must use; no user input is necessary to make the decision.

On-Demand Certificate Creation If the browser does not have an existing OBC for the origin it's connecting to, a new OBC will be created on-the-fly. This newly generated origin-bound certificate contains *no* user identifying information (*e.g.,* name or email). Instead, the OBC is used only to prove, cryptographically, that subsequent TLS sessions to a given server originate from the same client, thus building a continuous TLS *channel*[3], even across different TLS sessions.

User Experience As noted earlier, there is no user interface for creating or using Origin-Bound Certificates. This is similar to the UI for HTTP cookies; there is typically no UI when a cookie is set nor when it is sent back to the server. Origin-Bound Certificates are similar to cookies in other ways as well:

- Clients uses a different certificate for each origin. Unless the origins collaborate, one origin cannot discover which certificate is used for another.
- Different browser profiles use different Origin-Bound Certificates for the same origin.
- In incognito or private browsing mode, the Origin-Bound Certificates used during the browsing session get destroyed when the user closes the incognito or private browsing session.
- In the same way that browsers provide a UI to inspect and clean out cookies, there should be a UI that allows users to reset their Origin-Bound Certificates.

3.2 The Origin-Bound Certificates TLS Extension

OBCs do not alter the semantics of the TLS handshake and are sent in exactly the same manner as traditional client certificates. However, because they are generated on-the-fly and have no associated UI component, we must differentiate TLS-OBC from TLS client-auth and treat it as a distinct TLS extension. Figure 1 shows, at a high level, how this extension fits in with the normal TLS handshake protocol; the specifics of the extension are explained below.

The first step in the client-server decision to use OBCs occurs when the client advertises acceptance of the TLS-OBC extension in its initial `ClientHello` message. If the server chooses to accept the use of OBCs, it echoes the TLS-OBC extension identifier in its `ServerHello` message. At this point, the client and server are considered to have negotiated to use origin-bound client certificates for the remainder of the TLS session.

After OBCs have been negotiated, the server sends a `CertificateRequest` message to the client that specifies the origin-bound certificate types that it will accept (ECDSA, RSA, or both). Upon a client's receipt of the `CertificateRequest`, if the client has already generated an OBC associated with the server endpoint,

[3]We use the same notion as TAOS [27] does, of a cryptographically strong link between two nodes.

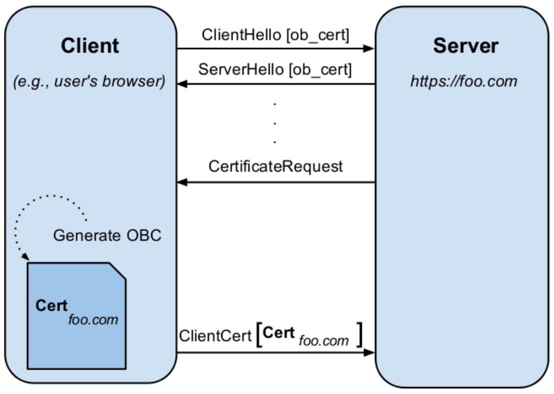

Figure 1: TLS-OBC extension handshake flow.

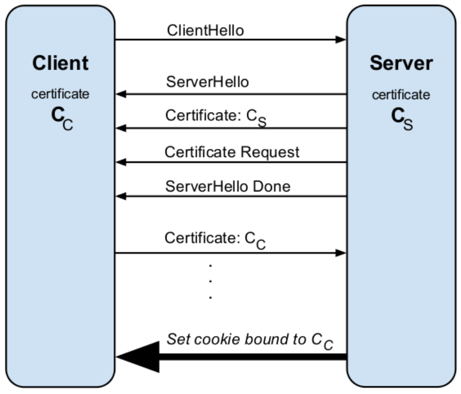

Figure 2: Process of setting an OBC bound cookie

the existing OBC is returned to the server in the client's `ClientCertificate` message. If this is the first connection to the server endpoint or if no acceptable existing OBC can be found, an origin-bound certificate must be generated by the client then delivered to the server in the client's `ClientCertificate` message.

During the OBC generation process, the client creates a self-signed client certificate with common and distinguished names set to "`anonymous.invalid`" and an X509 extension that specifies the origin for which the OBC was generated.

4 Securing Web Authentication Mechanisms with TLS-OBC

We now show how origin-bound certificates can be used to strengthen other parts of the network stack: In Section 4.1 we explain how HTTP cookies can be bound to TLS channels using TLS-OBC. In Section 4.2 we show how federation protocols (such as OpenID or OpenID Connect) can be hardened against attackers, and in Section 4.3 we turn briefly to application-level user authentication protocols.

4.1 Channel-binding cookies

OBCs can be used to strengthen cookie-based authentication by "binding" cookies to OBCs. When issuing cookies for an HTTP session, servers can associate the client's origin-bound certificate with the session (either by unforgeably encoding information about the certificate in the cookie value, or by associating the certificate with the cookie's session through some other means). That way, if and when a cookie gets stolen from a client, it cannot be used to authenticate a user when communicated over a TLS connection initiated by a different client – the cookie thief would also have to steal the private key associated with the client's origin-bound certificate – a task considerably harder to achieve (especially in the presence of Trusted Platform Modules or other Secure Elements that can protect private key material).

Service Cookie Hardening One way of unforgeably encoding an OBC into a cookie is as follows. If a traditional cookie is set with value v, a channel bound cookie may take the form of:

$$\langle v,\ \mathrm{HMAC}_k(v+f)\rangle$$

where v is the value, f is a fingerprint of the client OBC, k is a secret key (known only to the server), and $\mathrm{HMAC}_k(v+f)$ is a keyed message authentication code computed over v concatenated to f with key k. This information is all that is required to create and verify a channel bound cookie. The general procedure for setting a hardened cookie is illustrated in Figure 2. Care must be taken not to allow downgrade attacks: if both v and $\langle v, \mathrm{HMAC}_k(v+f)\rangle$ are considered valid cookies, a man-in-the-middle might be able to strip the signature and simply present v to the server. Therefore, the protected cookie *always* has to take the form of $\langle v, \mathrm{HMAC}_k(v+f)\rangle$, even if the client doesn't support TLS-OBC.

Cookie Hardening for TLS Terminators The technique for hardening cookies, as discussed above, assumes that the cookie-issuing service knows the OBC of the connecting client. While this is a fair assumption to make for most standalone services, it is not true for many large-scale services running in datacenters. In fact, for optimization and security reasons, some web services have TLS "terminators". That is, all TLS requests to and from an application are first passed through the TLS terminator node to be "unwrapped" on their way in and are "wrapped" on their way out.

There are two potential approaches to cookie hardening with TLS terminators. First, TLS terminators could extract a client's OBC and pass it, along with other information about the HTTP request (such as cookies sent by

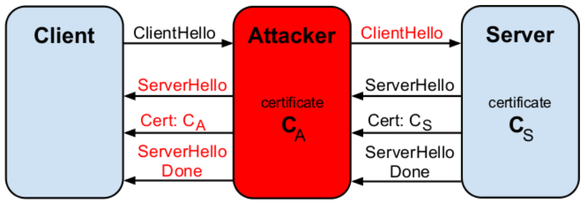

Figure 3: MITM attack during a TLS handshake

the client) to the backend service. The backend service can then create and verify channel-bound cookies using the general procedure in the previous section.

The second approach involves using the TLS terminator to channel-bind the cookies of legacy services that cannot or will not be modified to deal with OBC information sent to them by the TLS terminator. Using this approach, TLS terminators must receive a list of cookie names to harden for each service to which they cater. When receiving an outbound HTTP response with a `Set-Cookie` header for a protected cookie, the TLS terminator must compute the hardened value using the OBC fingerprint, rewrite the cookie value in the `Set-Cookie` header, and only then wrap the request in a TLS stream. Similarly, the TLS terminator must inspect incoming requests for `Cookie` headers bearing a protected cookie, validate them, and rewrite them to only have the raw value. Any inbound request with a channel-bound cookie that fails verification must be dropped by the TLS verifier.

Channel-Bound Cookies Protect Against MITM

As mentioned earlier, TLS MITM attacks happen and some can go undetected (see Figure 3 for a depiction of a conventional MITM attack). Channel-bound cookies can be used to bring protection against MITM attacks to web users.

Recall that our threat model assumes that at some time in the past, the user's client was able to successfully authenticate with the server. At that point, the server would have set a cookie on the client and would have bound that cookie to the client's legitimate origin-bound certificate. This process is shown in Figure 2. Observe that on a subsequent visit, the client will send its cookie (bound to the client's OBC). However, the MITM lacks the ability to forge the client's OBC and must substitute a new OBC in its handshake with the server. Therefore, when the MITM forwards the user's cookie on to the server, the server will recognize that the cookie was bound to a different OBC and will drop the request. This process is shown in Figure 4. The careful reader will observe that a MITM attacker may strip the request of any bearer tokens completely and force the user to provide his username/password once more or fabricate a new cookie and

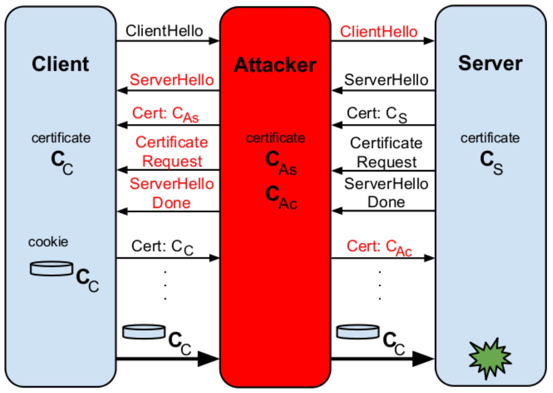

Figure 4: Using OBCs and bound cookies to protect against MITM. The server recognizes a mismatch between the OBC to which the cookie is bound and the cert of the client (attacker) with who it is communicating.

log the user in as another identity. We cover this more in Section 4.3 and in an upcoming report.

4.2 Hardening Federation Protocols

Channel-binding cookies with OBCs allows a single entity to protect the authentication information of its users, but modern web users have a plethora of other login credentials and session tokens that make up their digital identity. Federation protocols like OpenID [20], OpenID Connect [23], and BrowserID [14] have been proposed as a way to manage this explosion of user identity state. At a high level, these federation protocols allow the user to maintain a single account with an identity provider (IdP). This IdP can then generate an *identity assertion* that demonstrates to relying parties that the user controls the identity established with the identity provider. While these federation techniques reduce the number of credentials a user is responsible for remembering, they make the remaining credentials much more valuable. It is therefore critical to protect the authentication credentials for the identity provider as well as the mechanism used to establish the identity assertion between identity provider and relying party. Towards that end, we explore using TLS-OBC and channel-binding to harden a generic federation system against attack.

PostKey API The first step towards hardening a federation protocol is to provide a way for an identity provider and *relying party* to communicate in a secure, MITM resistant manner. We introduce a new browser API called the *PostKey* API to facilitate this secure communication. This new API is conceptually very similar to the *PostMessage* [11] communication mechanism that allows distinct windows within the browser to send messages to each other using inter-process communication

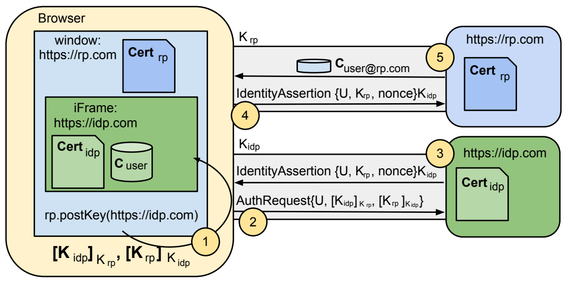
Browser
window: https://rp.com
Cert rp
iFrame: https://idp.com
Cert idp
C user
rp.postKey(https://idp.com)
[K idp] K rp, [K rp] K idp
1
K rp
5
C user@rp.com
IdentityAssertion {U, Krp, nonce}Kidp
4
https://rp.com
Cert rp
Kidp
3
IdentityAssertion {U, Krp, nonce}Kidp
AuthRequest{U, [Kidp] K rp, [Krp]Kidp}
2
https://idp.com
Cert idp

Figure 5: Simplified federation protocol authorization flow using PostKey and OBCs.

rather than the network. The goal of *PostKey* extends beyond a simple communication mechanism to encompass the secure establishment of a "proof key" that communicates the public key of an OBC to a different origin within the browser by exposing a new browser window function:

```
otherWindow.postKey(message, targetOrigin)
```

This `postKey` call works like the existing `postMessage` call but additional `cert` and `crossCert` parameters are added to the event received by the recipient window's message handler. The `cert` parameter contains a certificate that is signed by the receiver's origin-bound key and includes: the sender's origin, the sender's OBC public key, the receiver's origin, and an X509 extension that includes a random nonce. The `crossCert` has the sender and receiver's roles reversed (*i.e.,* it contains the receiver's key, signed by the sender's key) and includes the same random nonce as in `cert`.

These certificates form what is called a cross certification, where the recipient of the certification can establish that the sender's public key is K_S because K_S has been signed, by the browser, with the receiver's private key K_R. Additionally, the caller's public key cross-certifies the receiver's public key to establish that both keys belong to the same browser.

It's important to note that the sender does not get to choose the keys used in this cross certification process. Instead, the browser selects the OBCs associated with the origins of the sender and receiver and automatically performs the cross certification using the keys associated with the found OBCs.

Putting it all together The combination of the *PostKey* API and origin-bound certificates can be used to improve upon several federation protocols.

Figure 5 shows the steps required to federate a user's identity in a generic federation protocol that had been modified to work with the *PostKey* API and OBCs. In step 1 the *relying party* issues a *PostKey* javascript request to the IdP's iFrame and the IdP receives a cross certification from the web browser. In step 2, an Authorization Request is issued to the IdP. Since the request is sent over the TLS channel authenticated with K_{IdP} the server associates the incoming request with the user U associated with K_{IdP}. The authorization request contains the cross certification that asserts that K_{RP} and K_{IdP} belong to the same user's browser so upon user consent, the IdP can respond (in step 3) with a single use Identity Assertion that asserts that K_{RP} is also associated with user U. The IdP's iFrame then passes the Identity Assertion to the RP's frame where, in step 4, the Identity Assertion is forwarded to the relying party's server. The relying party verifies that the Identity Assertion was delivered over a channel authenticated with K_{RP}, has been properly signed by the IdP, and has not been used yet. If this verification succeeds the RP can now associate user U with key K_{RP} by setting a cookie in the user's browser as shown in step 5.

4.3 Protecting user authentication

We've largely considered the initial user-authentication phase, when the user submits his credentials (*e.g.,* username/password) in return for an authenticated session, to be out of scope for this paper. However, we now briefly outline how TLS-OBC can be leveraged in order to secure this tricky phase of the authentication flow.

As a promising direction where TLS-OBC can make a significant impact, we explore the ideas put forth by a recent workshop paper by Czeskis *et al.* [8], where the authors frame authentication in terms of protected and unprotected login. They define unprotected login as an authentication during which all of the submitted credentials are user-supplied and are therefore vulnerable to phishing attacks. For example, these types of logins occur when users first sign in from a new device or after having cleared all browser state (*i.e.,* cleared cookies). The authors observe that to combat the threats to unprotected login, many websites are moving towards protected login, whereby user-supplied credentials are accompanied by supplementary, "unphishable" credentials such as cookies or other similar tokens. For example, websites may set long-lived cookies for users the first time they log in from a new device (an unprotected login), which will not be cleared when a user logs out or his session expires. On subsequent logins, the user's credentials (*i.e.,* username/password) will be accompanied by the previously set cookie, allowing websites to have some confidence that the login is coming from a user that has already had some interaction with the website rather than a phisher. The authors argue that websites should move all possible authentications to protected login, minimize unprotected login, and then *alert* users when unprotected logins occur. The paper argues that this approach is meaningful because phishers are not able to produce protected logins and will be forced to initiate unprotected logins instead. Given that unprotected logins should occur rarely for legitimate users, alerting users during an unprotected login will make it significantly harder for password thieves to phish for user credentials.

It's important to note that websites can't fully trust protected logins because they are vulnerable to MITM attacks. However, with TLS-OBC, websites can protect themselves by channel-binding the long-lived cookie that enables the protected login. Combining TLS-OBC with the protected login paradigm allows us to build systems which are resilient to more types of attacks. For example, when describing the attack in Figure 4, we mentioned that attackers could deliver the user cookie, but that would alert the server to the presence of a MITM. We also mentioned that attackers could drop the channel-bound cookie altogether and force the user to re-authenticate, but that this attack was out of scope. However, using TLS-OBC along with the protected/unprotected paradigm, if the attacker forced the user to re-authenticate, the server could force an unprotected login to be initiated and an alert would be sent to the user, notifying him of a possible attack in progress. Hence, channel-bound cookies along with TLS-OBC would protect the user against this type of attack as well.

The careful reader will observe that protecting first logins from new devices (an initial unprotected login) is difficult since the device and server have no pre-established trust. We are currently in the beginning stages of building a system to handle this case and leave further discussion as future work.

5 Implementation

In order to demonstrate the feasibility of TLS origin-bound certificates for channel-binding HTTP cookies, we implemented the extensions discussed in Section 3. The changes made while implementing origin-bound certificates span many disparate systems, but the major modifications were made to OpenSSL, Mozilla's Network Security Services (used in Firefox and Chrome), the Google TLS terminator, and the open-source Chromium browser.

5.1 TLS Extension Support

We added support for TLS origin-bound certificates to OpenSSL and Mozilla's Network Security Stack by implementing the new TLS-OBC extensions, following the appropriate guidelines [5]. We summarize each of these changes below.

NSS Client Modifications Mozilla's Network Security Stack (NSS) was modified to publish its acceptance of the TLS-OBC extension when issuing a `ClientHello` message to a TLS endpoint. Upon receipt of a `ServerHello` message that demonstrated that the communicating TLS endpoint also understands and accepts the TLS-OBC extension, a new X509 certificate is generated on-the-fly by the browser for use over the negotiated TLS channel. These NSS modifications required 108 modified or added lines across 6 files in the NSS source code.

OpenSSL Server Modifications The OpenSSL TLS server code was modified to publish its acceptance of the TLS-OBC extension in its `ServerHello` message. Furthermore, if during the TLS handshake the client and server agree to use origin bound certificates, the normal client certificate verification is disabled and the OBC verification process is used instead.

The new verification process attempts to establish that the certificate delivered by the client is an OBC rather than a traditional client authentication certificate. The check is performed by confirming that the certificate is self-signed and checking for the presence of the X509 OBC extension. With these two constraints satisfied, the certificate is attached to the TLS session for later use by higher levels of the software stack.

An upstream patch of these changes is pending and has preliminary support from members of the OpenSSL community. The proposed patch requires 316 lines of modification to the OpenSSL source code where most of the changes focus on the TLS handshake and client certificate verification submodules.

5.2 Browser Modifications

In addition to the NSS client modifications discussed above, Chromium's cookie storage infrastructure was adapted to handle the creation and storage of TLS origin-bound certificates. The modifications required to generate the OBCs resulted in a 712 line patch (across 8 files) to the Chromium source code. Storage of OBCs in the existing Chromium cookie infrastructure required an additional 1,164 lines added across 15 files. These changes have been upstreamed as an experimental feature of Chromium since version 16.

6 Performance Evaluation

We have conducted extensive testing of our modifications to TLS and have found them to perform well, even at a significant scale. We report on these results below.

6.1 Chromium TLS-OBC Performance

Experimental methodology In order to demonstrate that the performance impact of adding origin-bound certificates to TLS connections is minimal, we evaluated the performance of TLS-OBCs in the open-source Chromium browser using industry standard benchmarks. All experiments were performed with Chromium version 19.0.1040.0 running on an Ubuntu (version 10.04) Linux system with a 2.0GHz Core 2 Duo CPU and 4GB of RAM.

All tests were performed against the TLS secured version of a Google's home page. During the tests JavaScript was disabled in the browser to minimize the impact of the JavaScript engine on any observed results. Additionally, SPDY connection pooling was disabled, the browser cache was cleared, and all HTTP connections were reset between each measured test run in order to eliminate any saved state that would skew the experimental results. The Chromium benchmark results discussed in section 6.1.1 were gathered with the Chromium benchmarking extension [12] and the HTML5 Navigation Timing [19] JavaScript interface.

6.1.1 Effects on Chromium TLS Connection Setup

We first analyzed the slowdown resulting the TLS-OBC extension for all connections bound for our website's *HTTPS* endpoints. The two use-cases considered by these tests were the first visit, which requires the client-side generation of a fresh origin-bound certificate, and subsequent visits where a cached origin-bound certificate is used instead.

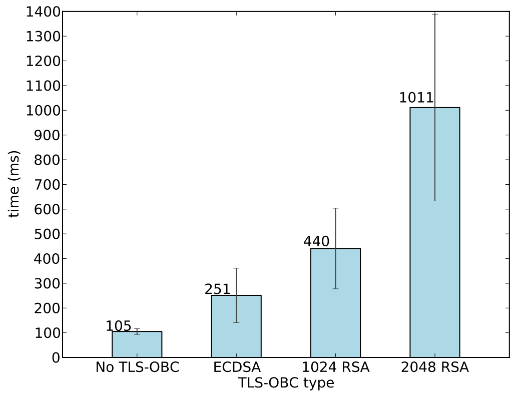

Figure 6: Observed Chromium network latency (ms) with TLS-OBC certificate generation.

The first test shown in Figure 6 shows the total network latency in establishing a connection to our web site and retrieving the homepage on the user's first visit. We measured the total network latency from the Navigation Timing *fetchStart* event to the *responseEnd* event, encapsulating TLS handshake time as well as network communication latency.

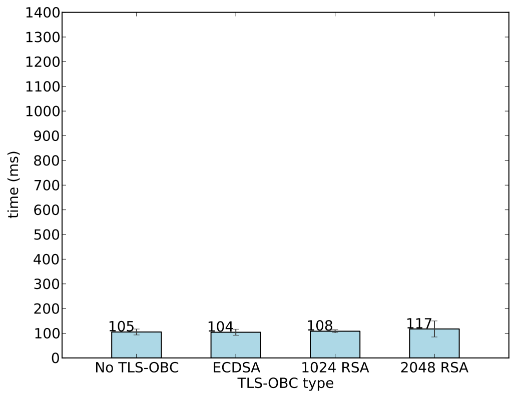

Figure 7: Observed Chromium network latency (ms), TLS-OBC certificate pre-generated.

The results shown in Figure 7 represent subsequent requests to our web site where there is a cache hit for a pre-generated origin-bound certificate. We observed no meaningful impact of the additional *CertificateRequest* and *Certificate* messages required in the TLS handshake on the overall network latency.

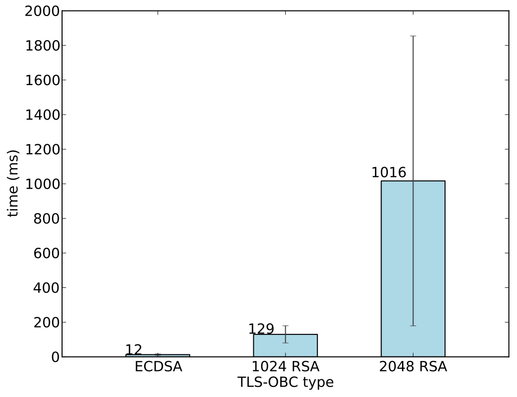

Figure 8: NSS certificate generate times (ms).

The differences between the latencies observed in Figures 6 and 7 imply that origin-bound certificate generation is the contributing factor in the slowdown observed when first visiting an origin that requires a new origin bound certificate. We measured the performance of the origin-bound certificate generation routine, as shown in Figure 8, and found that the certificate generation does seem to be the contributing factor in the higher latencies seen when first connecting to an origin with an origin-bound certificate.

Client Performance Analysis These observations demonstrate that certificate generation is the main source of slowdown that a client using origin-bound certificates will experience. The selection of public key algorithm has a significant impact on the fresh connection case, and an insignificant impact on subsequent connections. This suggests that production TLS-OBC browsers should speculatively use spare CPU cycles to precompute public/private key pairs, although fresh connections will still need to sign origin-bound certificates, which cannot be done speculatively.

6.2 TLS Terminator Performance

We also measured the impact of TLS-OBC on Google's high-performance TLS terminator used inside the datacenter of our large-scale web service. To test our system, we use a corpus of HTTP requests that model real-world traffic and send that traffic through a TLS terminator to a backend that simulates real-world responses, *i.e.,* it varies both response delays (forcing the TLS terminator to keep state about the HTTP connection in memory for the duration of the backend's "processing" of the request) as well as response sizes according to a real-world distribution. Mirroring real-world traffic patterns, about 80% of the HTTP requests are sent over resumed TLS sessions, while 20% of requests are sent through freshly-negotiated TLS sessions.

We subjected the TLS terminator to 5 minutes of 3000 requests-per-second TLS-only traffic and periodically measured memory and CPU utilization of the TLS terminator during that period.

We ran four different tests: One without origin-bound certificates, one with a 1024-bit RSA client key pair, one with a 2048-bit RSA client key pair, and one with a 163-bit client key pair on the *sect163k1* elliptic curve (used for ECDSA). We also measure the latency introduced by the TLS terminator for each request (total server-side latency minus backend "processing" time).

Figure 9 shows the impact on memory. Compared to the baseline (without client certificates) of about 1.85GB, the 2048-bit RSA client certs require about 12% more memory, whereas the 1024-bit RSA and ECDSA keys increase the memory consumption by less than 1%.

Figure 10 shows the impact on CPU utilization. Compared to the baseline (without client certificates) of saturating about 4.3 CPU cores, we observed the biggest increase in CPU utilization (of about 7%) in the case of the ECDSA client certificates.

Finally, Figure 11 through Figure 14 show latency histograms. While we see an increase in higher-latency responses when using client-side certificates, the majority of requests are serviced in under one millisecond in all four cases.

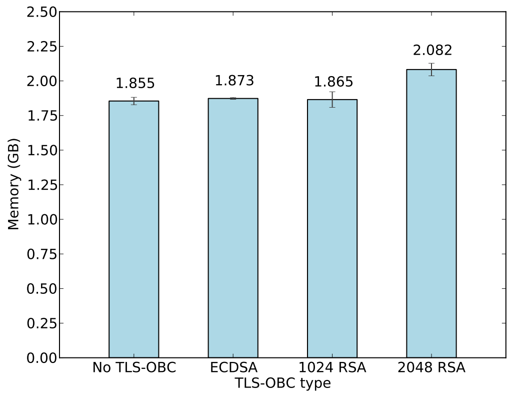

Figure 9: Server-side memory footprint of various client-side key sizes.

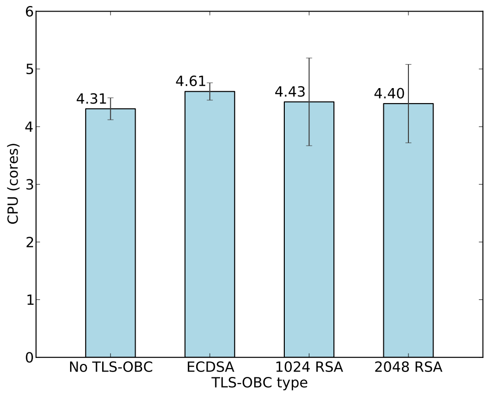

Figure 10: Server-side CPU utilization for various client-side key sizes.

Server Performance Analysis If we cared purely about minimizing the memory and CPU load on our TLS terminator systems, our measurements clearly indicate that we should use 1024-bit RSA. As 1024-bit RSA and 163-bit ECDSA are offer equivalent security [4], however the ECDSA server costs might be worth the client-side benefits.

7 Discussion – Practical Realities

We now discuss a variety of interesting details, challenges, and tensions that we encountered while dealing with the actual nature of how applications are developed and maintained on the web.

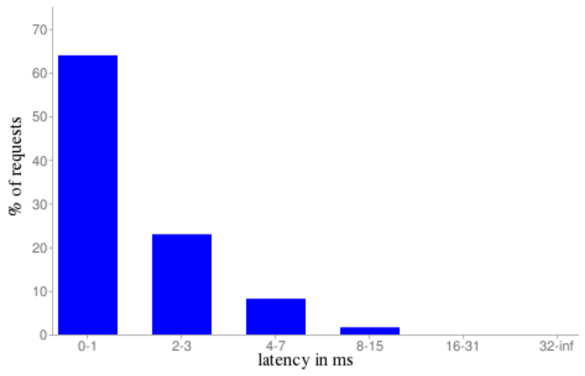

Figure 11: Latency without client certificates.

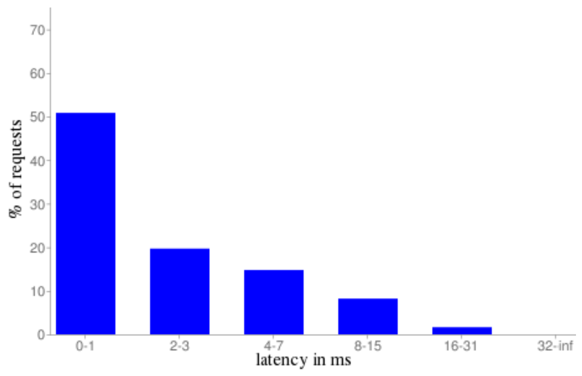

Figure 13: Latency with 2048-bit RSA certificate.

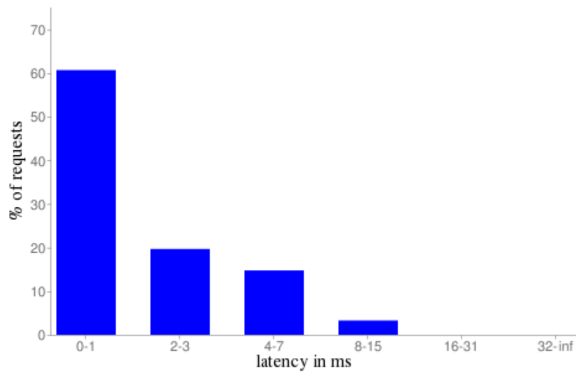

Figure 12: Latency with 1024-bit RSA certificate.

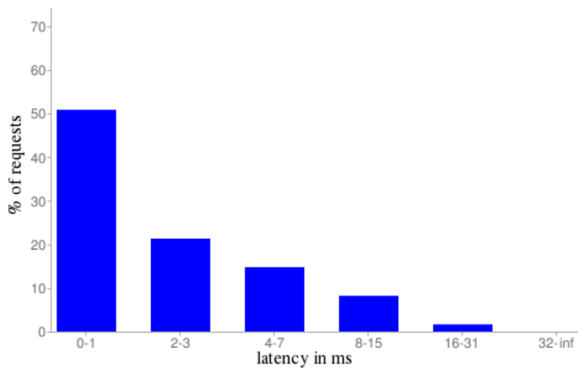

Figure 14: Latency with 163-bit ECDSA certificate.

7.1 Domain Cookies and TLS-OBC

In Section 4 we explained how cookies can be *channel-bound* using TLS-OBC, hardening them against theft. However, this works only as long as the cookie is not set across multiple origins. For example: when a cookie is set by origin *foo.example.com* for domain *example.com*, then clients will send the cookie with requests to (among others) *bar.example.com*. Presumably, however, the client will use a different client certificate when talking to *bar.example.com* than it used when talking to *foo.example.com*. Thus, the channel-binding will break.

Bortz *et al.* [6] make a convincing argument that domain cookies are a poor choice from a security point-of-view, and we agree that in the long run, domain cookies should be replaced with a mix of origin cookies and high-performance federation protocols.

In the meantime, however, we would like to address the issue of domain cookies. In particular, we would like to be able to channel-bind domain cookies just as we're able to channel-bind origin cookies.

To that end, we are currently considering a "legacy mode" of TLS-OBC, in which the client uses whole domains (based on eTLDs), rather than web origins, as the granularity for which it uses client-side certificates. Note that this coarser granularity of client certificate scopes does *not* increase the client's exposure to credential theft. All the protocols presented in this paper maintain their security properties against men-in-the-middle, *etc.* The only difference between origin-scoped client certificates and (more broadly-scoped) domain-scoped client certificates is that in the latter case, related domains (*e.g., foo.example.com* and *bar.example.com*) will be able to see the same OBC for a given browser.

It is also worth noting that even coarse-grained domain-bound client certificates alleviate many of the problems of domain cookies, if those cookies are channel-bound – including additional attacks from the Bortz *et al.* paper.

In balance, we feel that the added protection afforded to widely-used domain cookies outweighs the slight risk of "leaking" client identity across related domains, and are therefore planning to support the above-mentioned "legacy mode" of TLS-OBC.

7.2 Privacy

The TLS specification [9] indicates that both client and server certificates should be sent in the clear during the handshake process. While OBCs do not bear any information that could be used to identify the user, a single OBC is meant to be reused when setting up subsequent connections to an origin. This certificate reuse enables an eavesdropper to track users by correlating the OBCs used to setup TLS sessions to a particular user and track a users browsing habits across multiple sessions.

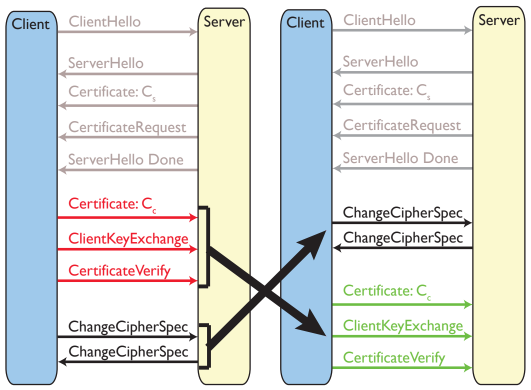

Figure 15: TLS encrypted client certificates

Towards rectifying this issue, we propose to combine TLS-OBC with an encrypted client certificate TLS extension. This extension modifies the ordering of TLS handshake messages so that the client certificate is sent over an encrypted channel rather than in the clear. Figure 15 shows the effect this extension has on TLS message ordering.

7.3 SPDY and TLS-OBC

The SPDY [26] protocol multiplexes several HTTP requests over the same TLS connection, thus achieving higher throughput and lower latency. SPDY has been implemented in Google Chrome for some time, and will be supported in Firefox 11. SPDY always runs over TLS.

One feature of SPDY is *IP pooling*, which allows HTTP sessions from the same client to different web origins to be carried over the same TLS connection if: the web origins in question resolve to the same IP address, *and* the server in the original TLS handshake presented a certificate for all the web origins in question.

For example, if *a.com* and *b.com* resolved to the same IP address, and the server at that IP address presented a valid certificate for *a.com* and *b.com* (presumably through wildcard subject alternative names), then a SPDY client would send requests to *a.com* and *b.com* through the same SPDY (and, hence, TLS) connection.

Remember that with TLS-OBC, the client uses a different client TLS certificate with *a.com* than with *b.com*. This presents a problem. The client needs to be able to present different client certificates for different origins.

In fact, this is not a problem unique to TLS-OBC, but applies to TLS client authentication in general: theoretically speaking, a client might want to use different non-OBC TLS certificates for different origins, even if those origins qualify for SPDY IP pooling.

One solution to would be to disallow SPDY IP pooling whenever the client uses a TLS client certificate. Instead, the client would have to open a new SPDY connection to the host to which it wishes to present a client certificate. This solution works well when client certificates are rare: most of the time (when no client certificates are involved), users will benefit from the performance improvements of SPDY IP pooling. When TLS client certificates become ubiquitous, however (as we expect it to be the case through TLS-OBC), most of the time the client *would not* be able to take advantage of SPDY IP pooling if this remained the solution to the problem.

Therefore, SPDY needs to address the problem of client certificates and IP pooling. From version 3 onward, it does this by adding a new CREDENTIAL control frame type. The client sends a CREDENTIAL frame whenever it needs to present a new client certificate to the server (for example, when talking to a new web origin over an IP-pooled SPDY connection). A CREDENTIAL frame allows the client to prove ownership of a public-key certificate without a new TLS handshake by signing a TLS extractor value [21] with the private key corresponding to the public-key certificate.

7.4 Other Designs We Considered

Before settling on TLS-OBC, we considered, and rejected, a number of alternative designs. We share these rejected ideas below to further motivate the choice for TLS-OBC.

Application-Level Crypto API In this design, web client applications would be able to use a crypto API (similar to a PKCS#11 API, but accessible by JavaScript in the browser). JavaScript would be able to generate key pairs, have them certified (or leave the certificates self-signed), use the private key to sign arbitrary data, *etc.*, all without ever touching the private key material itself (again, similar to PKCS#11 or similar crypto APIs).

Every web origin would have separate crypto key containers, meaning that keys generated in one web origin would not be accessible by Javascript running in other web origins. It would be up to individual applications to sign relevant (and application-specific) authentication tokens used in HTTP requests (*e.g.,* special URL query parameters) with keys from that web origin. The application could further design its authentication tokens in such a way that they don't grant ambient authority to a user's account, but rather authorize specific actions on a user's account (*e.g.,* to send an email whose contents hashes to a certain value, *etc.*).

Such a system would give some protection against a TLS MITM: being unable to mint authentication tokens itself, the attacker could only eavesdrop on a connection. Also, this approach doesn't require changes in the TLS or HTTP layers, and is therefore "standards committee neutral", except for the need for a standardized JavaScript crypto API, which presumably would be useful in other contexts (than authentication) as well.

Note, however, that TLS-OBC with channel-bound cookies provides strictly more protection, preventing men-in-the-middle from eavesdropping. This approach is also vulnerable to XSS attacks and requires applications to be re-written to use these application-level authentication tokens (instead of existing cookies).

We didn't consider the advantages mentioned above strong enough to outweigh the disadvantages of this approach.

Signed HTTP Requests We also explored designs where the client would sign HTTP requests at the HTTP layer. For example, imagine an HTTP request header "X-Request-Signature" that contained a signature of the HTTP request. The key used to sign requests would be client-generated, per-origin, *etc.*, just like for TLS-OBC. Unlike TLS-OBC, this would not require a change in TLS, or HTTP for that matter. This design, however, quickly morphed into a re-implementation of TLS at the HTTP layer. For example, protection against replay attacks leads to timestamps, counters, synchronization issues, and extra round trips. Another example is session renegotiation, questions of renegotiation protocols, and the resulting induced latency.

TLS solves all these issues for us: it protects against replay attacks, allow session renegotiation to be multiplexed with data packages, and many other issues that would have to be addressed at the HTTP layer. We felt that the TLS extension we're proposing was far less complex than the additions to the HTTP layer that would have been necessary to get to comparable security, hence our focus on TLS.

8 Related Work

Origin-bound certificates are closely related to traditional client certificates; we take this opportunity to explain why traditional client certificates don't work in today's web. We also briefly mention various similar efforts to remedy the security issues with authentication on the web, and explain why they stop short of a complete solution.

8.1 Traditional TLS Client Certificates

While TLS *server* authentication is widely used across the web, the *client* authentication aspect of TLS is used much less frequently. Just like TLS server authentication identifies a web server to a client (*i.e.,* browser), TLS client authentication uses public key cryptography to authenticate a client to a web server; this process is an optional part of the TLS handshake.

While effective in small, managed systems such as enterprise networks, the flaws of TLS client authentication begin to emerge as we examine them at web scale:

Bad User Experience One issue that prevents conventional TLS client authentication from becoming the standard for web authentication is the cumbersome, complicated, and onerous interface that a user must wade through in order to use a client certificate. Typically, when web servers request that browsers generate a TLS client certificate, browsers display a dialog where the user must choose the certificate cipher and key length. Even worse, when web servers request that the browser provide a certificate, the user is prompted to select the client certificate to use with the site they are attempting to visit. This "login action" happens during the TLS handshake, before the user can inspect any content of the website (which presumably would help her decide whether or not she wanted to authenticate to the site in the first place).

Layer Confusion Arguably, TLS client authentication puts user identity at the wrong layer in the network stack. An example that reveals this layer confusion is multi-login: Google has implemented a feature in which multiple accounts can be logged into the website at the same time (multiple user identities are encoded in the cookie). This makes it easy to quickly switch between accounts on the site, and even opens up the potential to show a "mashup" of several users' accounts on one page (*e.g.,* show calendars of all the logged-in accounts). With TLS client authentication, the user identity is established at the TLS layer, and is "inherited" from there by the HTTP and application layers. However, client certificates usually contain exactly one user identity, thus forcing the application layer to also only see this one use identity.

Privacy Once a user has obtained a certificate, any site on the web can request TLS client authentication with that certificate. The user can now choose to not be logged in at all, or use the same identity at the new site that they use with other sites on the web. That is a poor choice. Creating different certificates for different sites makes the user experience worse: Now the user is presented with a list of certificates every time they visit a website requiring TLS client authentication.

Portability Since certificates ideally are related to a private key that can't be extracted from the underlying platform, by definition, they can't be moved from one device to another. So any solution that involves TLS client authentication also has to address and solve the user credential portability problem. Potential solutions include re-obtaining certificates from the CA for different devices, extracting private keys (against best security practices) and copying them from one device to another, or cross-certifying certificates from different devices. So far we have not been able to come up with good user interfaces for any of these solutions.

Trusted Computing Base in Datacenters Large datacenters often terminate TLS connections at the datacenter boundary [3], perhaps even using specialized hardware for this relatively expensive part of the connection setup between client and server. If the TLS client certificate is what authenticates the user, then the source of that authentication is lost at the datacenter boundary.

This means that the TLS terminators become part of the trusted computing base – they simply report to the backends who the user is that was authenticated during the TLS handshake. A compromised TLS terminator would in this case essentially become "root" with respect to the applications running in the datacenter.

Contrast this with a cookie-based authentication system, in which the TLS terminator forwards the cookie that the browser sends to the app frontend. In such a system, the cookies are minted and authenticated by the app frontend, and the TLS terminator would not be able to fabricate arbitrary authentic cookies. Put another way, in a cookie-based authentication system a compromised TLS terminator can modify an incoming request before it is delivered to the backend service, but cannot forge a completely new request from an arbitrary user.

In summary, TLS client authentication presents a range of issues, ranging from privacy to usability to deployment problems that make it unsuitable as an authentication mechanism on the web.

8.2 Other Related Efforts

CardSpace Microsoft's CardSpace [13] authentication system attacked two of the problems mentioned so far: First, it replaced passwords with a public-key based protocol, thus eliminating one kind of bearer tokens. Second, it moved user identity from the TLS layer to the application layer.

It allowed users to manage multiple digital identities from a single user interface. CardSpace stored user identities in the form of identity "cards". When visiting a website that implemented the CardSpace protocol, users could choose which card, and hence which identity, to use to authenticate with that website. Instead of a username/password pair, a cookie, or a TLS client certificate, CardSpace would authenticate users by sending cryptographic tokens that encoded the user identity. There is no consensus on why CardSpace did not become an industry standard; however, we believe the same complexity that gave CardSpace a wide variety of features, also contributed to its demise by unnecessarily complicating the user interface, interaction, and development models.

CardSpace by itself was also agnostic to the use of bearer tokens in lower layers of the protocol stack once the user was logged in. In this paper we approach the problem from the opposite direction: we build a strong foundation at the TLS layer that allows us to harden other protocols (HTTP, application-specific login, *etc.*), so theoretically origin-bound certificates and CardSpace are more complementary than competing proposals – in particular one could imagine a "channel-bound" CardSpace token that results in a channel-bound cookie (see Section 4). However, we strive to learn from CardSpace's failure in the market and carefully designed our system to not alter the user experience (and burden developers) too much from what users (and developers) are already used to.

BrowserID Mozilla has recently developed a prototype of an authentication mechanism called BrowserID [14], which abstracts identity to the level of email addresses. BrowserID is aimed at the password bearer token, at least for websites that choose to become relying parties to email providers. For those, instead of using a password, users authenticate by providing a cryptographic proof of email ownership. Similarly to CardSpace, the browser maintains a cache of emails (identities) and generates the respective proofs (tokens) for the user. Unlike CardSpace, BrowserID is based on both a simpler model of identity (email addresses vs. a variety of claims) and a simpler implementation platform (JWTs vs. WS-Trust).

BrowserID is complementary to the ideas put forth in this paper. Since it mostly plays at the application layer, it is agnostic to the use of bearer tokens at lower layers (*e.g.,* HTTP cookies). It could easily be adjusted by binding BrowserID identity assertions to the underlying TLS channel if the browser supports origin-bound certificates.

TLS-SA As another approach, Opplinger *et al.* address the disconnect between user authentication and TLS channels in their proposed TLS Session Aware (TLS-SA) User Authentication scheme [17, 18]. TLS-SA is intended to solve the man-in-the-middle (MITM) problem by providing the server side of a TLS connection with the information necessary to determine if a user's credentials have been sent over a different TLS session than the session that the client thought the credentials were being sent over. However, these protections apply only to the initial user credentials and not to the subsequent bearer tokens. To our knowledge TLS-SA has neither been implemented nor tested on a mass, web scale.

Hardening Cookies Some work has also focused on hardening the information stored in HTTP cookies. For example, Murdoch presented a method for toughening cookies by encoding values not only based on on a secret server key, but also on a hash of the user's password [15]. This approach has the benefit of making it harder for attackers to fabricate fake cookies (even if the secret server key has been compromised), but does not protect the user if the cookie is ever stolen.

9 Conclusion

In this paper we presented TLS origin-bound certificates as a new approach to TLS client certificates. TLS-OBCs act as a foundational layer on which the notion of an authenticated channel for the web can be established.

We showed how TLS-OBCs can be used to harden existing HTTP layer authentication mechanisms like cookies, federated login protocols, and user authentication.

We implemented TLS-OBCs as an extension to the OpenSSL and NSS TLS implementations and deployed TLS-OBC to the Chromium open source browser as well as the TLS terminator of a major website.

Finally, we demonstrated that the performance overhead imparted by using TLS-OBC is small in terms of CPU and memory load on the TLS server and observed latency on the TLS client.

We see origin-bound certificates as a first step towards enabling more secure web protocols and applications.

10 Acknowledgements

A great number of individuals have contributed to the work presented in this paper. We would like to thank the team at Google, including Mayank Upadhyay, Adam Langley, Wan-Teh Chang, Matt Mueller, Ryan Hamilton, Diana Smetters, Adam Barth and Warren Zhang for helping us develop the ideas presented in this paper, and for implementing and testing them. Our thanks go out to Ben Adida, Mike Hanson and Brian Smith from Mozilla, as well as the members of the IETF TLS Working Group for sanity-checking and improving our proposals. We would also like to thank Tadayoshi Kohno for his support.

Finally, we would like to thank the anonymous reviewers of our manuscript for helping us make this a better paper.

References

[1] H. Adkins. An update on attempted man-in-the-middle attacks. http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html, Aug 2011.

[2] D. Balfanz. TLS Origin-Bound Certificates. http://tools.ietf.org/html/draft-balfanz-tls-obc-01, Nov 2011.

[3] J. Barr. AWS Elastic Load Balancing: Support for SSL Termination. http://aws.typepad.com/aws/2010/10/elastic-load-balancer-support-for-ssl-termination.html, Oct 2010.

[4] S. Blake-Wilson, T. Dierks, and C. Hawk. ECC Cipher Suites for TLS. http://tools.ietf.org/html/draft-ietf-tls-ecc-01, March 2001.

[5] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (tls) extensions. http://tools.ietf.org/html/rfc4366, Apr 2006.

[6] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for web applications. In *Web 2.0 Security & Privacy*, 2011.

[7] E. Butler. Firesheep. http://codebutler.com/firesheep, 2010.

[8] A. Czeskis and D. Balfanz. Protected Login. In *Proceedings of the Workshop on Usable Security (at the Financial Cryptography and Data Security Conference)*, March 2012.

[9] T. Dierks and C. Allen. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force, Jan. 1999. RFC-2246, ftp://ftp.isi.edu/in-notes/rfc2246.txt.

[10] T. Dierks and E. Rescorla. The Trandsport Layer Security (TLS) Protocol Version 1.2 – Client Certificates, 2008. http://tools.ietf.org/html/rfc5246#section-7.4.6.

[11] I. Hickson. HTML5 Web Messaging. http://dev.w3.org/html5/postmsg/, Jan 2012.

[12] J. Hurwich. Chrome benchmarking extension. http://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/chrome-benchmarking-extension, Sept 2010.

[13] Microsoft. Introducing windows cardspace, 2006. http://msdn.microsoft.com/en-us/library/aa480189.aspx.

[14] Mozilla. BrowserID, 2012. https://developer.mozilla.org/en/BrowserID.

[15] S. Murdoch. Hardened stateless session cookies. *Security Protocols XVI*, pages 93–101, 2011.

[16] A. Mushaq. Man in the Browser: Inside the Zeus Trojan, 2010. http://threatpost.com/en_us/blogs/man-browser-inside-zeus-trojan-021910.

[17] R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication–or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2338–2246, 2006.

[18] R. Opplinger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication revisited. *Computers & Security*, 27(3-4):64–70, 2008.

[19] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[20] D. Recordon and B. Fitzpatrick. OpenID authentication 1.1. http://openid.net/specs/openid-authentication-1_1.html, May 2008.

[21] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). http://tools.ietf.org/html/rfc5705, March 2010.

[22] J. Rizzo and T. Duong. Beast. http://vnhacker.blogspot.com/2011/09/beast.html, Sept 2011.

[23] N. Sakimura, D. Bradley, B. de Mederiso, M. Jones, and E. Jay. OpenID connect standard 1.0 - draft 07. http://openid.net/specs/openid-connect-standard-1

[24] C. M. Shields and M. M. Toussain. Subterfuge: The MITM Framework. http://subterfuge.googlecode.com/files/Subterfuge-WhitePaper.pdf, 2012.

[25] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, Massachusetts, May 1986.

[26] The Chromium Project. SPDY, 2012. http://www.chromium.org/spdy.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994.