Practical, Usable, and Secure Authentication and Authorization on the Web

Alexei Czeskis

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Program Authorized to Offer Degree:
UW Computer Science and Engineering

University of Washington

Abstract

Practical, Usable, and Secure Authentication and Authorization on the Web

Alexei Czeskis

Chair of the Supervisory Committee:
Associate Professor Tadayoshi Kohno
Department of Computer Science and Engineering

User authentication and authorization are two of the most critical aspects of computer security and privacy on the web. However, despite their importance, *in practice*, authentication and authorization are achieved through the use of decade-old techniques that are both often inconvenient for users and have been shown to be insecure against practical attackers. Many approaches have been proposed and attempted to improve and strengthen user authentication and authorization. Among them are authentication schemes that use hardware tokens, graphical passwords, one-time-passcode generators, and many more. Similarly, a number of approaches have been proposed to change how user authorization is performed. Unfortunately, none of the new approaches have been able to displace the traditional authentication and authorization strategies on the web. Meanwhile, attacks against user authentication and authorization continue to be rampant and are often (due to the lack of progress in practical defenses) successful.

This dissertation examines the existing challenges to providing secure, private, and usable user authentication and authorization on the web. We begin by analyzing previous approaches with the goal of fundamentally understanding why and how previous solutions have not been adopted. Second, using this insight, we present three systems, each aiming to improve an aspect of user authentication and authorization on the web. *Origin-Bound Certificates* provide a deployable and secure building block for user credential transfer on the web. *PhoneAuth* uses Origin-Bound Certificates in order to allow users to securely authen-

ticate to service providers in the face of strong attackers while maintaining the traditional username/password authentication model. Finally, *Allowed Referrer Lists* allow developers to easily protect applications against authorization vulnerabilities.

We present the design, implementation, and evaluation for each of the three systems, demonstrating the feasibility of our approaches. Together, these works advance the state of the art in practical, usable and secure user authentication and authorization on the web. These systems demonstrate that through deep consideration of fundamental stakeholder values and careful engineering, it is possible to build systems that increase the security of user authentication and authorization without adversely impacting the user and developer experiences, while at the same time being deployable and practical.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to extend sincere gratitude to my advisor, Prof. Tadayoshi Kohno for mentoring, guiding, and supporting me (academically, financially, emotionally) through all aspects of my doctoral career. Thank you to all other members of my dissertation committee (Prof. Arvind Krishnamurty, Prof. Hank Levy, and Prof. Raadhakrishnan Poovendran) for their valuable feedback and support.

I would like to thank all my collaborators and coauthors through the years for their insight and for allowing me the privilege of working with them. Particularly, I would like to thank Dr. Dirk Balfanz, Dr. Michael Dietz, Dr. Alexander Moshchuk, Dr. Helen Wang, and Prof. Dan Wallach for their collaboration on parts of this dissertation.

I would like to extend my gratitude to Dr. Lindsay Michimoto for providing invaluable guidance through the Ph.D. process and to Melody Kadenko for making all administrative aspects of my life easier (and for providing chocolate).

Thank you to my laboratory family: Tamara Denning, Miro Enev, Karl Koscher, Adam Lerner, Temitope Oluwafemi, Franzi Roesner, and Alex Takakuwa. A special thank you to Karl Koscher for his continuous collaboration with me throughout the years. Thank you to Taryn Kohno for allowing me to steal Yoshi at odd times of the evening.

Additionally, thank you to Ivayla Dermendjieva, Hussein Yapit, and Ian Smith for allowing me to mentor you.

Thank you to my amazing family — my sister, dad, and mom — for your motivation and support. Finally, thank you to my girlfriend and friends (especially Alexandr Raevskiy) for keeping me sane during my graduate career.

DEDICATION

To those who have been humble enough to lend me a hand.

Chapter 1
INTRODUCTION

Secure and reliable communication on the web has many moving parts, but two of the most critical aspects are user *authentication* and *authorization.* Authentication is the process by which one entity (*e.g.*, a server) identifies another entity (*e.g.*, a user) remotely over the web (see Figure 1.1). Authorization, on the other hand, is the process by which a web service decides whether to allow or deny a user action (see Figure 1.2). It is important that both authentication and authorization are performed securely, lest attackers be able to impersonate legitimate users and gain access to sensitive user data, cause data loss, or engage in identity theft. Authentication and authorization have been important since the first day of multi-user systems and interconnected networks; now they have become indispensable as virtually all aspects of an individual's life are moving on-line.

While much of the web infrastructure has changed over the past several decades, user authentication and authorization have seen little progress. Users authenticate to web services much like they did in 1990 — by providing a username and password. In return, web applications set cookies in the user's browser which must then be returned (by the browser) with subsequent HTTP requests. Any requests bearing the correct user cookies are considered authenticated. Many websites, also consider cookie bearing requests to be authorized. Since web browsers attach cookies to all requests (with or without user intent), many requests thus have incorrect or *ambient* authority. Unfortunately, this leads to a class of attacks called Cross-Site Request Forgery (CSRF) — see Chapter 5 for more details. To protect against CSRFs, the long-term best-practice is to use an approach called *tokenization*, whereby secret tokens are included in webpages and must be returned in order for subsequent requests to be considered legitimately authorized.

One may be quick to assume that the permanence of these authentication and authorization methods implies that "we got it right" or that passwords and cookies are sufficient and no replacement is actually needed. However, the reality is quite different. Researchers have continuously shown that the status quo is lacking in security and usability [22, 27]. For example, on the authentication front, users have difficulty remembering complicated passwords, leading them to choose short, guessable passwords [51]. Additionally, since there is a cognitive burden associated with remembering multiple passwords, users often reuse passwords across sites [51]. This allows attackers to compromise passwords databases on weakly protected sites and then use those passwords to compromise user accounts on more strongly protected sites. On the authorization front, tokenization is difficult to implement correctly, often leading to CRSF vulnerabilities (Chapter 5 covers this extensively).

The security community has provided developers with tools to address and patch many of the "low-hanging", easily exploitable vulnerabilities in software systems. Perhaps as a result, attackers are increasingly pursuing user authentication and authorization as viable attack vectors. Indeed, attacks on user authentication and authorization are being launched in the wild, sometimes with serious consequences [3, 25, 87, 122]. For example, weaknesses in authentication have allowed attackers to compromise user email accounts, send scam emails to their family members, and then wipe the user's account — deleting all contacts, emails, and stored documents [47]. In another attack, attackers have compromised an email account, impersonated the user on social networks (posting socially offensive remarks), and then used the credentials in the email account to remotely wipe his laptop, phone, and tablet [60].

On the authentication front, researchers have proposed a wide variety of alternative (and complementary) mechanisms. For example, proposed schemes include graphical passwords (*e.g.*, requiring users to recognize faces [93] or to draw patterns [8]), hardware tokens (*e.g.*, RSA SecurID [42]), biometrics (*e.g.*, fingerprint [64]), and many others. Nevertheless, passwords still reign as the *de facto* authentication mechanism on the web. Practical progress in user authorization has also remained largely stagnant — unable to move past tokenization despite several proposals. Some natural questions are: *Why?* and *How do we move forward?* This dissertation offers some insight into answering these questions.

Specifically, this dissertation analyzes the challenges that exist in the modern-day authentication and authorization space, and proposes novel technologies to *provide deployable,*

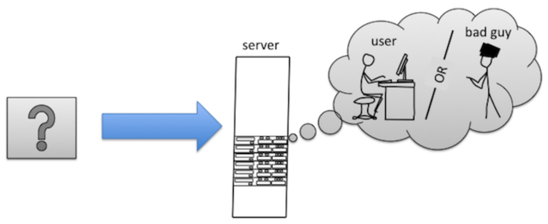


Figure 1.1: *Authentication.* The goal of authentication is for the server to identify if the incoming request is being issued by a legitimate user (if so, then which) or an attacker.

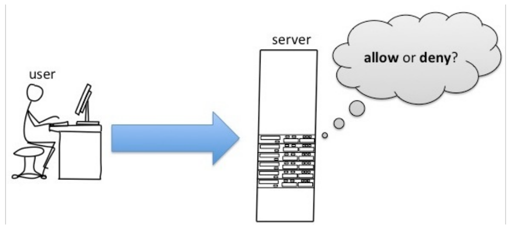


Figure 1.2: *Authorization.* The goal of authorization is for the web service to accurately decide whether a particular user request should be allowed or not.

usable, and secure authentication and authorization on the web. By examining the current technological landscape, we identify three underlying weaknesses, which in turn lead to attacks against authentication and authorization: (1) web clients have no means of establishing strong authenticated channels with web servers; (2) current methods for augmenting password-based user login with strong second factors are not practical because they do not provide a sufficient mix of usability and security or do not consider deployability factors; and (3) current authorization defenses are error-prone and heavyweight. As a result, many web applications are built without adequate security protections and put user accounts and data at risk.

Contributions The work presented in this dissertation seeks to address weaknesses in current authenticaiton/authorization landscape and provide developers with practical tools to protect user accounts against adversaries. Towards this goal, we presents the design, implementation, and evaluation of three systems, each focusing on improving one of the aforementioned underlying weaknesses in user authentication and authorization. *Origin-Bound Certificates* [40] provide a deployable and secure building block for user credential transfer on the web by allowing developers to establish strongly authenticated channels between web clients and servers. *PhoneAuth* [31] uses Origin-Bound Certificates in order to allow users to securely authenticate to service providers in the face of strong attackers while maintaining the traditional username/password authentication model. Finally, *Allowed Referrer Lists* [32] allow developers to easily protect applications against authorization vulnerabilities.

Taken together, these systems demonstrate that it is possible to design and build solutions for strengthening user authentication and authorization on the web without sacrificing usability for both developers and end-users. A key step to making our systems deployable was to understand the constraints and realities under which industry operates — we did this by drawing insight from and collaborated with leaders in the IT industry such as Google and Microsoft. As an additional contribution of this thesis, we share what we've learned about the industry nuances that can often hinder the adoption of new approaches. For example, Origin Bound Certificates are specifically designed to work with the way major cloud providers deploy TLS terminators in data centers.

This thesis is organized as follows: Chapter 2 provides background and motivation for our work by discuss the challenges of offering secure user authentication and authorization. Additionally, Chapter 2 also explores prior efforts in this area, specifically focusing on how other approaches have succeeded and failed. We then use this insight in Chapters 3 – 5, where we present new systems along with their specific motivation, design, and evaluation. Chapter 6 concludes and offers some future directions.

Chapter 2
BACKGROUND: CHALLENGES AND PREVIOUS APPROACHES

This chapter explores the current and past landscape of user authentication and authorization on the web. We begin by considering the needs of key stakeholders involved (such as users and service providers) as well as the attackers' capabilities and the threat model they create. We demonstrate how providing usable and practical authentication and authorization for users and service providers creates non-trivial challenges in the face of trying to build security against attackers. We then analyze the merits and shortcomings of a number of previous approaches for providing usable, practical, and secure user authentication and authorizatioon.

2.1 Challenges to Secure User Authentication and Authorization

Improving usable and secure user authentication and authorization on the web is deceivingly difficult. Specifically, both users' and service providers' goals and constraints must be considered in evaluating any potential authentication approach. Additionally, any authentication or authorization mechanism must remain viable under adversarial pressure. We now further consider the key stakeholders in turn below:

2.1.1 Users

We begin with the following premise: we assume that users want to do as little work as possible in order to get access to their account. Furthermore, we assume that many users don't want anyone else to access or modify their data. To this end, users are willing to spend a limited amount of time an effort authenticating. Exactly, "how much effort" is an open question. We expect that users would be willing to spend more effort on security while authenticating to a bank rather and less when authenticating to a link-sharing site. Users have been trained to use passwords for authentication. Consequently, despite their known usability issues [58], we believe that the usability of passwords is a good litmus test for evaluating new approaches. That is, if it takes any more time or effort to authenticate with some mechanism than with passwords, then users may find it too cumbersome; if it takes less time, then users may consider it to be more usable.

On the authorization front, in most scenarios, users currently have to do little or no work to authorize requests (this all happens "under the hood"). Though asking users to confirm transactions (*e.g.*, for banking) seems to be commonplace, asking users to authorize every click or even e-mail is not. Approaches should be careful about creating additional overhead for users when authorizing user requests.

2.1.2 Service Providers

We continue with these additional premises: on one hand, service providers want users to authenticate as *easily* as possible (in order to please users); on the other hand, service providers want to make user authentication *thorough* to ward off attackers and prevent user account compromise.

Besides these constraints, service providers operate with business interest. That is, we assume service providers will generally be reluctant to adopt authentication and authorization solutions that cause them to perform significant software (or hardware) development or that cause them a large per-user cost. The cost function includes, besides other variables, time and money to set up authentication with a new user (*e.g.*, mailing them a physical token or sending an SMS) as well as the effort required to recover an account in case the users lock themselves out (*e.g.*, by forgetting a password).

2.1.3 Attackers / Threat Model

There are several main forms of attackers which are present both in literature and in practice. We examine each in turn — from the least, to the most powerful types. For the purpose of this thesis, we are interested in studying the case when users browse the web with patched browsers and operating systems, making it difficult for attackers to silently install malware on the user's machine. While unpatched users exist (as do zero-day vulnerabilities), these type of attackers are out the of scope of this thesis.

Web attackers

Entities who operate malicious websites (*e.g.*, `bad.com`[1]) are called *web attackers.* We assume that legitimate users will eventually accidentally visit `bad.com`. The attacker may design `bad.com` to visually mimic the user's bank or e-mail website in an attempt to get (*i.e.*, *phish*) the user's login credentials. As the attackers are quite skilled, and may occupy clever URLs (such as `6ank.com`[2]), we assume that most users will fall for this trick. That is, they will enter credentials (such as their password) into the attacker controlled `bad.com`. Therefore, we assume that attackers will be able to steal user passwords in this manner.

Additionally, web attackers can cause users' browsers to make arbitrary HTTP requests to legitimate sites by including content (such as iframes or links to images) in the malicious sites. This is an important capability as it allows attackers to perform Cross-Site Request Forgery (CSRF) attacks. The attack is illustrated in Figure 2.3 and works as follows: The user logs into the legitimate site (*e.g.*, `bank.com`), which then sets a cookie on the user's browser. Sometime later, the user visits the malicious site (*e.g.*, `bad.com`), which includes content (such as an image tag) that causes the user's browser to make an HTTP request to the legitimate site (*e.g.*, `bank.com/transfer_funds?to=eve`). Furthermore, since the browser has cookies stored for the legitimate site, the browser will attach those cookies with the request. If the legitimate site is not careful, it may assume that this request was authorized by the user and perform the malicious transaction.

Network attackers

Entities who are able to observe and modify the network traffic between the user and the legitimate site are called *network attackers.* A network attacker may be as localized as an adversary sniffing wireless traffic in a coffee shop. This attack is not difficult and tools for doing so are readily available on the Internet [5, 25]. A larger scale example of a

[1]This is just an example, the real `bad.com` may not be malicious

[2]This is just an example, the real `6ank.com` may not be malicious

network attacker is a nation-state that interposes themselves on all traffic entering, leaving, or transitioning the country. Countries such as China, Iran, Armenia, and many others are known to do so [61, 98]. These type of attacks are called *man-in-the-middle* attacks.

Network attackers can clearly extract sensitive authentication data (such as passwords and cookies) out of unencrypted network traffic (*e.g.*, HTTP). The Transfer Security Layer (TLS) [38] is specifically designed to defeat such attacks by providing message confidentiality and integrity between the user and the legitimate server. Unfortunately, attackers have been able to defeat the security properties offered by TLS by forging rogue server TLS certificates [3] or exploiting vulnerabilities in the TLS cryptographic layer [99, 6]. The net result is the same — attackers are able to extract sensitive data (such as cookies and passwords) out of network traffic.

Related-site attackers

Some attackers will compromise sites having weak security practices in order to steal the site's user credentials. As users often reuse credentials across sites [51], related-site attackers will try to reuse the stolen credentials on more secure sites in hopes of gaining access to the user accounts [50].

2.2 Techniques for Authenticating Users

A number of alternative techniques have been proposed for authenticating users. We now examine these techniques and comment on their technical and usability contributions, their security properties, and their potential for practical deployment. Enumerating all proposed authentication approaches is not possible, therefore we attempt to give a good background of the field as a whole through several canonical examples. While we take a somewhat informal approach to analyzing these mechanisms, other works have attempted to quantify the differences between these schemes (*e.g.*, Bonneau *et al.* [22]). While the Bonneau *et al.* work was very informative in writing this chapter, we attempt to offer a more broad coverage of the authentication space (*e.g.*, Bonneau *et al.* doesn't cover TLS client certificates, but we do) while at the same time providing more (somewhat subjective) insight into why a scheme failed or succeeded to be adopted.

2.2.1 Password Managers

As we saw in Section 2.1.1, users struggle with producing and remembering distinct strong passwords. *Password Managers* help users overcome this usability barrier by (as their name implies) storing and managing users' passwords.

Most major browsers (such as Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer) offer to remember and auto-complete user generated passwords [52, 80, 83]. Some browsers (*e.g.*, Chrome and Firefox) support syncing password data across different computers, allowing users to freely move between different devices without fear of losing access to their passwords. The syncing is done through the service provider's servers and requires users to provide a "master" password. Additionally, browser-based password managers allow users to view their stored passwords — a really useful feature as users can forget their passwords since users no longer have to type them.

Besides in-browser password managers, several third-party password managing applications exist — of which LastPass [72] is currently the most prominent. LastPass supports installation on the top five browsers, across the three most popular operating systems, and on all of the popular mobile phone platforms. Like the in-browser password managers, LastPass records, auto-completes, and syncs users' passwords across devices. LastPass also offers advanced options such as using two-factor authentication mechanisms (*e.g.*, hardware tokens, see Section 2.2.7) and the ability to auto-generate passwords during account creation on websites.

Similar to LastPass, another class of password managers attempts to more proactively prevent users from sharing passwords across sites. Two notable examples are PwdHash [100] and Password Multiplier [56]. Both extensions work by using a cryptographic hash to generate a site-specific password. In order to invoke the password manager, users must perform a particular action (such as typing "@@" or double-clicking the password field).

Analysis Password managers help users with keeping track of all their passwords. With the recent advances in cross-platform password syncing and storing passwords in the cloud, users are no longer "locked out" of their accounts as they move from device to device.

Since password managers don't change the low level way in which users authenticate to web services (with passwords), they require no server-side changes — making password managers very deployable.

However, user studies have shown that users are uncomfortable with relinquishing control over their passwords [28]. Furthermore, since most password managers allow users to view their passwords, users are still vulnerable to phishing as attackers can ask users to copy-paste the password into a web form. Password managers that sync passwords through the cloud are secured using a master password. If this password leaks, so do all of the users' other passwords. Finally, password managers do nothing to stop man-in-the-middle attacks.

2.2.2 Single Sign-On

Single Sign-On (SSO) are technologies for allowing users to authenticate once to an Identity Provider and then have access to many systems (potentially from different organizations) without having to authenticate again. *Microsoft Passport* [79] (closely related to and interchangeable with *Windows Live ID*), *OpenID* [96], *Google ID* [53], *Facebook Login* [45], and *Kerberos* [81] are examples of SSO. The typical user experience is for users to navigate to `example.com`, indicate which account (Google, Facebook, etc...) they would like to authenticate with, be redirected to the respective identity provider, and either authenticate (typically by providing a username and password) or just approving the login (if already authenticated). Kerberos is the most well-known SSO solution, but is not used on the Web in its raw form. Microsoft Passport, though meant to be used by non-Microsoft entities is mostly used by Microsoft or business doing business with Microsoft (to our knowledge). Facebook Login appears to be the most popular SSO product deployed as of this writing.

Analysis Similar to password managers, SSO allows users to enter passwords less often — thereby reducing the risk of phishing and increasing usability. Additionally, SSO allows smaller sites, which may not have a sufficient developer force, to not have to store or deal with user passwords — a security win.

At the same time, however, just as with the master password for password managers, losing or exposing the user's password becomes much more catastrophic as it now gives the attacker access to many more accounts. By opting into an SSO solution, websites hand authentication over to another party. While this may be good for smaller sites, larger sites view loss of control as a downside — partly for security reasons (sites don't want to rely on the other site's security measures), for availability reasons (sites don't want to lock users out if the other site goes down), and for business reasons (user data is financially valuable). This means that many large sites create authentication/SSO solutions of their own. In turn, this creates usability issues for users, whereby users are presented with many different authentication options. This is colloquially known as the NASCAR problem[3]. Finally, by using SSO, users share their browsing patterns with their identity provider.

2.2.3 Authentication Proxies

Closely related to Single Sign-On and password managers, some researchers have proposed proxy-based authentication. In this scheme, users first register all of their accounts and passwords at the proxy. When the user wants to authenticate to an actual website, the user first authenticates to the proxy (we'll describe how below), which then authenticates to the server on the user's behalf using the stored username and password credential.

Two examples of proxy-based authentication are URRSA [49] and Impostor [92]. With URRSA, each of the user's passwords is stored n times, encrypted with n different keys. At registration time, the user prints out a sheet of one-time-codes. The codes are the passwords encrypted under the n different keys. The codes are organized into groups — each group represents a username / password combination that the user originally registered. In order to authenticate to the proxy and to indicate to which site the user wants to authenticate, the user chooses the next unused code off their sheet in the appropriate group. The proxy decrypts the code and then forwards the password to the appropriate site.

At registration time, Imposter stores the user's usernames / passwords and requires the user to remember a passphrase. In order to use a web service, the user must first authenticate to Imposter, this involves first configuring the browser proxy settings to use Imposter as a proxy. Next, Imposter asks the user for random letters of the secret phase.

[3]in the sense that NASCAR cars have many company logos — just like SSO login pages have many identity provider logos

This allows the user to authenticate from untrusted clients as the client will not learn secrets that it can replay in the future to get access.

Analysis Both of the proxy-based approaches do a good job of protecting users against authenticating from untrusted devices. Similar to password managers and SSO schemes, proxy-based approaches allow users to type their passwords less often. However, with URRSA, users end up typing one-time codes instead. Imposter's pass-phrase has the same weakness as the master password for password managers and SSO approaches — if it is lost or stolen, the consequences could be dire. Furthermore, if the proxy is compromised, then security for multiple sites could be compromised. Since passwords are still used under the hood by the proxy to authenticate to web services, these approaches are still vulnerable to a man-in-the-middle attacker between the proxy and the web service. These approaches also create a single point of failure — if the proxy becomes unavailable, the user is prevented from accessing any of his accounts. Furthermore, both approaches change the user experience during the authentication flow. To our knowledge, neither proxy-based approach is deployed on the general internet.

2.2.4 Graphical Passwords

A set of authentication approaches relies on the human ability to recognize and remember visual patterns; these are called *Graphical Passwords*. Biddle *et al.* [16] give a good summary overview of the last 12 years of graphical passwords. The canonical example of a graphical password is a type of draw-a-secret scheme [65] or PassGo [114] where the user must recall and draw a secret pattern. The most in-use example of this scheme is perhaps the connect-the-dots pattern available on android mobile phones that users can enable in order to unlock their screen [8].

Another example of graphical passwords utilizes the ability of people to recognize human faces. One prominent example is *PassFaces* [93]. To use PassFaces, users must select n (where n is typically less than 10) faces to remember. During the authentication phase, the user is presented with a grid of faces, from which the user must select the correct faces. A variant of this scheme is used by Facebook when risky/suspicious activity is detected on a account (such as logging in from a strange IP address) [46]. The user is shown several images of his/her "friends" and asked to identify the name of the person in each picture.

Yet another form of graphical passwords is click-based, where users must click (or touch) the correct sections of the screen. Many other graphical password schemes exist. For example, some have users recognize icons [15], while others have users recognize art [35].

Analysis Graphical passwords have somewhat recently become deployed on major services (Android and Facebook being the prominent examples). It's not yet clear whether these approaches will grow in popularity or wane away into non-existence.

Studies have shown that, just as users predictably choose passwords, they also predictively choose faces to remember (attractive female faces being the most popular) [44]. These studies suggest that systems choose faces, instead of letting users do this. Additionally, it's not clear that remembering many distinct graphical passwords (one for each site) is cognitively simpler than remembering multiple passwords [44]. Implementing graphical password authentication on the server requires redesigning the server architecture. From a security point of view, graphical passwords are still vulnerable to phishing and man-in-the-middle attacks.

2.2.5 Cognitive Authentication

A new and fresh approach is based on modifying and analyzing a user's subconscious. Bojinov *et al.* [20] recently proposed an authentication scheme where they carefully designed a computer game to implant a secret in the user's brain without the user knowing. To authenticate, the user would be asked to play a game and would (unknowingly) prove knowledge of the secret through the way he/she played the game. During the learning phase, the user was trained to do a specific task called Serial Interception Sequence Learning (SISL) — a specific set of keystrokes. To authenticate, the user was presented with multiple SISL tasks (where one of the tasks includes some of the previously trained elements), and the user ends up performing better on these trained tasks. In the paper, training took approximately 30 minutes and authentication took approximately five minutes.

Denning *et al.* [34] proposed a similar approach using pairs of images: a complete image and their degraded counterparts. Users would be shown a random subset of complete images during the *priming* phase. During the authentication phase, users would be asked to identify complete images from their degraded versions. Some of the degraded images shown to the user included images that were shown to the user during the priming phase. Denning *et al.* found that users were able to better identify images on which they had previous been primed. As a result, they conclude that this is a viable potential authentication method.

Analysis One interesting property of the Bojinov approach is that the user does not know the secret. In their paper, Bojinov *et al.* observer that this property allows the authentication scheme to be resistant against "rubber-hose cryptanalysis" whereby the user is physically coerced into revealing the password. While this claim seems to be plausible, at the present the scheme has too much overhead for being practically deployable. Not only would it require significant restructuring of server infrastructure, but also takes too long to authenticate (five minutes is too long to login on the web). Indeed, Bojinov *et al.* observed that their approach is appropriate for "on-site" authentication, not remote authentication over the web.

The approach proposed by Denning *et al.* seems viable, but has a couple of shortcomings. The most significant weakness is the low entropy throughput; that is, it is necessary to show many images to users before being able to successfully authenticate them. The authors note this weakness and suggest that the scheme is most applicable to password recovery, rather than being a replacement for password authentication.

2.2.6 Biometrics

Biometric authentication strategies are based on unique physical properties of humans — also known as the "what you are" method of authentication. Many biometric schemes are possible; common examples of biometrics are fingerprints matching, iris scans, vein matching, voice recognition, face recognition, and keystroke dynamics [64].

Analysis Biometrics has some well known and well studied trade-offs. For example, if a biometric leaks, it becomes very hard to revoke or change (*e.g.*, we are born with only one set of fingerprints). Additionally, for high value targets, attackers have been known to cut off fingers in order to get access to protected resources [68]. With the exception of voice recognition and keystroke timing, biometrics require the existence of dedicated sensors at the client's computer. These additional sensors are not available on most computers. However, some modern laptops and mobile phones are experimenting with biometric authentication. For example, Android phones have an experimental feature allowing screen unlock based on facial recognition and several notebooks come with fingerprint readers that can be enabled to unlock the screen. However, to our knowledge, biometrics have not seen any deployment for user authentication over the web — perhaps because it requires re-architecting login systems. Additionally, service providers may be wary of handling such personal data for authentication.

2.2.7 One-Time Passcodes

One-Time Passcodes (OTP) enable non-replayable login credentials. One popular example of a hardware OTP generator is the RSA SecurID [42] family of tokens. The SecurID token is usually in the form-factor of a small token which has a small output window — capable of showing six or so digits. Each device is programmed with an initial secret seed. This seed is then used as input to a function that outputs a pseudo-random six digit code approximately every 60 seconds. The secret seed also stored by the server; this allows the server to generate and verify the correct code at any point in time. Each user is issued a token with a different seed. The seed is very difficult to physically extract from the SecurID token. In order to log into a web service, users first type their password and then copy the current 6 digit code into the web browser.

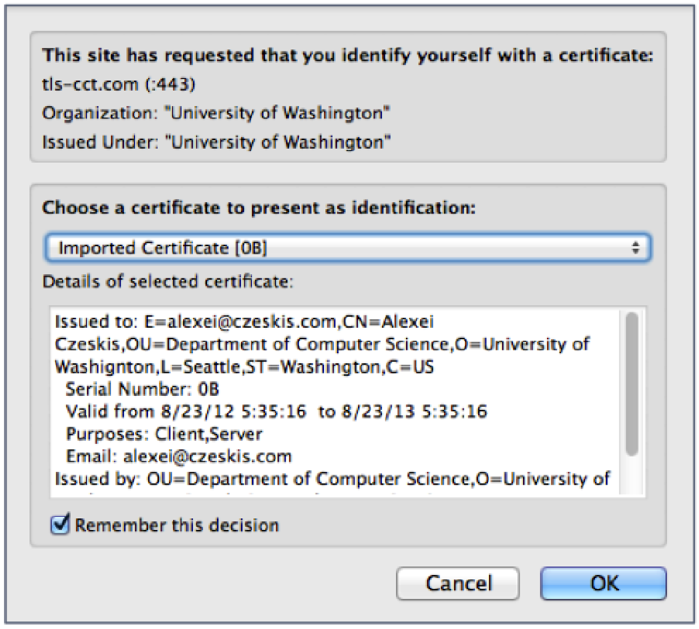
A closely related hardware token is made by Yubico and is called the YubiKey [120]. Similar to the RSA SecurID, the YubiKey stores a secret seed and generates pseudo-random six-digit one time codes. However, the YubiKey takes the form factor of a USB key with a single button. To use it, users plug the YubiKey into the USB drive and press the button on the token. The YubiKey imitates a keyboard and the enters/types the next valid code into the currently selected window — the advantage being that users don't have to manually type the code. Similar to RSA SecurID, the YubiKey is meant to be used as a second factor in conjunction with a regular password.

OTP generators can also be based in software or paper. For example, Google Authenticator [110] is a software based OTP generator that runs on many mobile phones (Android, IOS, Blackberry). Some services (such as Google's 2-step verification [54]) allow users to print out a sheet of one-time passcodes to use. Several financial institutions in the U.S. such as PayPal and Bank of America allow users to optionally enroll in additional verification that forces the user to submit, in addition to their username and password, a code sent to the users mobile phone via SMS; the practice is much more common in some other parts of the work like Europe.

Analysis The use of one-time passcodes as second factor to authenticating with a password has positive security implications. Specifically, it's no longer sufficient for the attacker to simply steal a user's password. The attacker must also get access to a valid code in order to authenticate as the user. Unfortunately, attackers can do this through phishing or a man-in-the-middle attack. Additionally, the use of one-time passcode generators adds overhead to the user authentication experience. For hardware-based OTP generators, users must carry around an additional piece of hardware and/or must manually type in the code from the device into their web browser. Hardware OTP generators do not scale well, as users must have a separate OTP generator for each web service for which they would like to use two-factor authentication. Software OTP generators and OTP over SMS have better usability implications, but still require users to take phones out of their pocket, unlock them, and manually copy the codes into the browser. At the end of the day, these approaches are also vulnerable to phishing and man-in-the-middle attacks.

2.2.8 TLS Client Certificates

User authentication at the TLS layer (via client certificates), if implemented and deployed correctly, solves the problem of phishing and man-in-the-middle attacks. In this scheme, users are issued certificates that bear the user's identity. These certificates can be stored in the user's browser or on external hardware devices (such as a smart card). In order to



This site has requested that you identify yourself with a certificate:
tls-cct.com (:443)
Organization: "University of Washington"
Issued Under: "University of Washington"

Choose a certificate to present as identification:

Imported Certificate [0B]

Details of selected certificate:

```
Issued to: E=alexei@czeskis.com,CN=Alexei
Czeskis,OU=Department of Computer Science,O=University of
Washignton,L=Seattle,ST=Washington,C=US
  Serial Number: 0B
  Valid from 8/23/12 5:35:16  to 8/23/13 5:35:16
  Purposes: Client,Server
  Email: alexei@czeskis.com
Issued by: OU=Department of Computer Science,O=University of
```

Figure 2.1: *TLS Client Certificate Prompt.* The prompt shown to a user during TLS Client authentication. The user must select the correct certificate to use. The prompt contains many fields that may be confusing to users.

authenticate, users navigate to a web service that requires TLS client authentication. This will cause the browser to prompt the user for their client certificate. Once the user selects the correct certificate to use, it will be sent to the requesting website. This will authenticate the user without the user having to enter a username or password. TLS client authentication is part of the standard TLS protocol and is resistant to phishing and man-in-the-middle attacks [38]. The particular security details of this protocol is outside the scope of this chapter.

Analysis TLS client certificate authentication provides significant security benefits over passwords. Unfortunately, it has significant downsides:

- *Poor 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 (see Figure 2.1. 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 user 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 (or allowing users to procure certs for their own 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 [11], sometimes 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, or harvest seen cookies, 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.

2.2.9 Mobile Phones (or PDAs)

Mobile phones (and formerly PDAs) are both powerful and fairly ubiquitous, making them attractive tools for securing user authentication over the web. For example, Phoolproof [91] effectively outsources TLS client authentication from the browser to the mobile phone. Users must first pair their phone with a PC and with several websites. For each paired website, the user's phone displays a bookmark. Clicking on this bookmark will cause the user's browser to navigate to the specified page.

Another system, MP-Auth [74] employs the user's mobile phone as a secure input terminal for passwords. That is, in order to log into a website, users type their passwords into their phone, which then participates in a challenge-response scheme with the website (through the user's browser) in order to authenticate the user.

Analysis While mobile phones are indeed powerful, systems to-date fall short of providing the necessary combination of usability and security. Both Phoolproof and MP-Auth require users to interact with their phones during the login process. We believe this change in the authentication procedure may be confusing and too onerous for users. Finding the phone (in their purse or backpack), unlocking the screen, and entering their password is a task that users shouldn't have to do. Both Phoolproof and MP-Auth are research systems, and unfortunately, leave several important questions unanswered. It's not clear how easy it would be for users to enroll in the system or what users would do if their phone became unavailable either temporarily (*e.g.*, dead batteries) or permanently (*e.g.*, lost or broken). To our knowledge, neither of these schemes has been deployed in practice.

2.2.10 Identity in the Browser

Several approaches have tried to manage user identity in the browser. One notable example is Microsoft CardSpace [77]. CardSpace replaced passwords with a public-key based protocol. Users would manage their digital identities through virtual identity "cards." When visiting a website that supported CardSpace, users would be presented with a UI that allowed them to choose which card, and thus which identity, to use with the site. Under the hood, CardSpace authenticated users by creating cryptographic attestations of the user's identity that could be communicated to the verifying website. This approach had the advantage of not revealing the authentication secret (typically a private key) to the verifying site. Furthermore, because users logged in by selecting a "card" rather than typing a password they could not be phished.

Mozilla has recently developed a prototype of an authentication mechanism called Persona (formerly known as BrowserID [2]), which abstracts identity to the level of email addresses. With Persona, instead of using a password, users authenticate by providing a cryptographic proof of email address ownership. Similarly to CardSpace, the browser maintains a cache of email addresses (identities) and generates the respective proofs (tokens) for the user. Unlike CardSpace, Persona 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).

Analysis Unfortunately, CardSpace was not widely adopted and was eventually discontinued altogether. We believe that CardSpace's attempt to provide many new features increased its overall complexity and contributed to its demise by unnecessarily complicating the user interface, interaction, and development models. Unfortunately, it's too early to tell whether Persona will see mass adoption.

2.3 Techniques for Authorizing Requests

We now describe existing defenses for authorizing user requests. We explain how existing approaches fall short of a comprehensive solution. We dive deep into how current solutions are designed, deployed, and used across the web, looking at both currently deployed defenses and those that have been proposed but not yet adopted. We begin by providing a bit of background regarding authorization on the web and how incorrectly performing authorization leads to Cross-Site Request Forgery (CSRF) attacks. We then discuss CSRF defenses, which come in three flavors: server-side, client-side, and server/client hybrids; we will consider each in turn.

2.3.1 Background

E-commerce web sites, webmail services, and many other web applications require the browser and server to maintain state about user sessions. Today, the *de facto* method of doing so is through HTTP Cookies, which are simply key/value pairs that a server

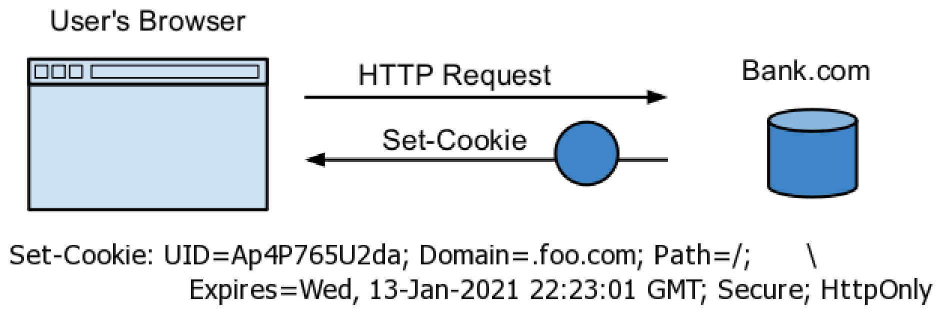

User's Browser
HTTP Request
Set-Cookie
Bank.com
Set-Cookie: UID=Ap4P765U2da; Domain=.foo.com; Path=/; \
Expires=Wed, 13-Jan-2021 22:23:01 GMT; Secure; HttpOnly

Figure 2.2: **Example of a server *bank.com* setting a cookie.** The cookie name is "UID" with a value of 11Ap4P765U2da. The cookie will be sent by the browser with every request to **.foo.com/** ("domain" and "path"), but only over HTTPS ("secure"). The cookie cannot be read or set from JavaScript ("HttpOnly"), and will expire on 13-Jan-2021.

can pass to and retrieve from the user's browser. A server "sets" a cookie by adding a `Set-Cookie` HTTP header to an HTTP response. By default, the browser stores the cookie for the current browsing session and uses the `Cookie` header to attach it to any subsequent HTTP requests it makes to the same web domain. The server may add attributes in the `Set-Cookie` header to change how the browser should handle the cookie. For example, the server can set a cookie's expiration date with the "Expires" attribute (making the cookie persistent), restrict the cookie to be sent only over HTTPS with the "Secure" attribute, and disallow JavaScript access to the cookie with the "HttpOnly" attribute. Additionally, the server may limit the cookie's scope to particular sub-domains and/or URL paths via the "Domain" and "Path" attributes. Figure 2.2 shows an example of how a server sets a cookie.

Web servers use cookies to store a variety of client-side state. For example, to tie HTTP requests to users, many servers store the user ID or session ID in a cookie. Some web applications also reduce load on their backend servers by using cookies to store frequently queried values, such as language preferences or UI settings.

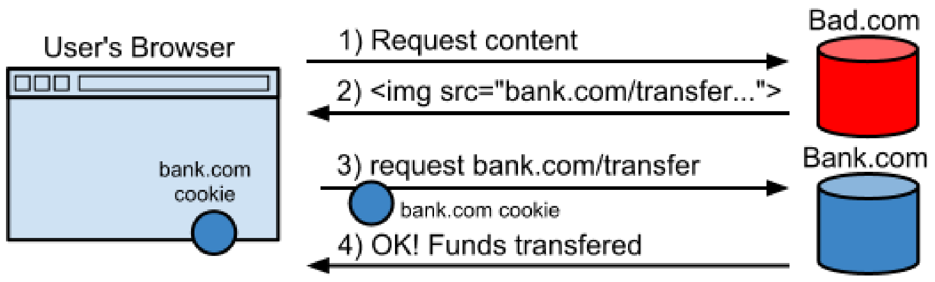


Figure 2.3: **An example CSRF attack.** When the user visits the adversary's page, the HTTP reply (step 2) includes code that causes the user's browser to make a request to *bank.com* (and attach *bank.com's* cookie). *Bank.com* erroneously treats this request as legitimate since it has the user's cookie.

Cross-Site Request Forgery

As mentioned above, cookies are often used for authentication — as bearers of user identity. Many web applications, however, mistakenly use the same cookie not only for *authentication*, but also for *authorization*. Specifically, many sites assume that an HTTP request bearing the user's cookie must have been initiated by the user. Unfortunately, this is not necessarily true in today's web. In fact, if the user visits an attacker's web page, the attacker can cause the user's browser to make HTTP requests to *any* web origin. Figure 2.3 explains how this mechanism can be used by adversaries in order to attack users. This is known as Cross-Site Request Forgery (CSRF)

Cause of CSRFs: Ambient Authority The root cause of CSRFs is the prevalence of *ambient authority* on today's web. Ambient authority means that web sites rely on data *automatically* sent by browsers as an indication of authority and thereby legitimate user intent. While cookie misuse is the most widespread cause of ambient authority and CSRF attacks, there are a number of lesser-known means by which CSRF can happen:

- **HTTP Authentication**: Some web sites use HTTP Authentication [14] to authenticate users. Browsers prompt the user for a username/password and send the user-supplied credentials in an "Authorization" HTTP header. The browser then caches these credentials and resends them whenever the server requests them. Note that *authentication* data is being sent in an *authorization* header — both confusing and misleading. Attackers may create CSRFs by causing the user's browser to send requests to an origin with cached HTTP Authentication credentials. A separate "Proxy-Authorization" header similarly authenticates users to proxies, with similar implications for CSRF. More advanced techniques such as NTLM [78] exist for verifying authenticated clients, but they eventually cause similar tokens to be sent in an HTTP header.

- **Source IP Address**: A corporation may grant access to intranet sites based on a client's source IP. For example, employees may request vacation days, add dependents, or divert parts of their paycheck towards a charitable organization through the intranet. When visiting an attacker's site, a user's browser may be instructed to connect to an intranet IP address with potentially malicious consequences.

- **Client-Side TLS Certificates**: The TLS protocol has support for both TLS server and (less popular) TLS client certificates. Client certificates can encode the user identity and, just like cookies, be used to identify a user. Unlike cookies or HTTP Authentication, TLS client certificates are not sent with every request. Instead, they are used to initially establish an authenticated TLS session. A web application can then consider any data sent through the authenticated TLS session as belonging to the respective user. However, if a site uses TLS client certificates for authorization (rather than purely authentication), the site may be vulnerable to CSRFs, since attackers can cause browsers to send requests over authenticated TLS sessions.

In each of the above scenarios, a web application relies on a single "token" (IP address, cookie, HTTP header, or client certificate) as an indication of authorization. We call these tokens *bearer tokens*. Because most of today's web sites implement authorization based on cookies, the majority of known CSRF attacks are cookie-based. Nevertheless, other types of attacks have been observed in the wild. For example, "router pharming" attacks [111] use JavaScript in the user's browser to change DNS settings in home routers, many of which use Basic Authentication or source IP for user authorization.

2.3.2 Server-side Defenses

Server-side solutions rely solely on server logic for CSRF protection. They are currently the most popular type of CSRF defense.

Tokenization

The current best practice for CSRF protection involves the use of a secret token. This approach works as follows:

1. When the user loads a page from the web application, the web server generates a secret token (a string) and includes it in the body of the web page.

2. As the user interacts with the web page and causes state-changing requests to be issued back to the web server, those requests include the secret token.

3. The web server then verifies the existence and correctness of the token received in the request before continuing execution.

Note that the secret token is not sent automatically by the browser (as is the case with cookies). Instead, the secret token is stored in the web page's DOM, and the page attaches it to requests programmatically via scripts or HTML forms. The security of this approach stems from the token being tied to the user's current session and being random enough to not be guessable by an attacker.

Implementing anti-CSRF tokenization involves three steps: (1) limit all "unsafe" operations to POST requests (as per RFC 2616 [48]), (2) include tokens in all HTML forms and AJAX requests that issue POSTs back to the server, and (3) verify the existence of the correct CSRF token when processing POST requests at the server.

Traditionally, developers implemented tokenization in a manual manner. A developer would write code to generate and validate tokens and then find and secure each part of the application that generates or handles POST requests. To simplify this daunting process, several CSRF protection frameworks have been developed (*e.g.*, CSRF Guard [88, 95]). Most frameworks automate POST request tokenization by rewriting HTML forms and adding token information to AJAX queries. Although these frameworks exist, we believe that many web applications still implement CSRF protection manually; this appears to be especially true for applications written using older web platforms, such as PHP or Perl.

More recent web development platforms (*e.g.*, Ruby on Rails, ASP.NET, and Django) include token-based CSRF protection as part of the standard development platform package. In some cases, CSRF protection is enabled for all pages; in others, developers must mark specific classes, controllers, views, or other platform components as requiring CSRF protection. In these cases, the web platform issues CSRF tokens when creating HTML output and validates the tokens when processing POST data submissions.

While CSRF frameworks and integrated tokenization in web platforms have simplified tokenization's deployment, we argue that tokenization is an incomplete defense having many drawbacks.

Incompatible with GET requests Tokens must not be sent over GET requests since GET requests may be logged by proxies or other services, or may be posted on the web by users, thus leaking the token. One may think that this problem does not arise in practice, as RFC 2616 specifically designates GET as a safe and idempotent method, which would make tokens unnecessary for GETs. However, real web applications don't follow this paradigm. For example, we investigated several popular web sites and found that (as of late 2012) Google, Amazon, live.com, and PayPal all use GET URLs to log users out. This is clearly not an idempotent action and, because none of the four web applications use CSRF protection for the logout action, an attacker can terminate a user's session with any of these applications without user consent. As another example, we found that Flickr.com (as of early 2012) uses GET requests for actions like changing the display language. Flickr does protect these requests with a CSRF token (sent as a URL parameter), but unfortunately uses the same token for POST requests as well. Because a Flickr user's token is the same from session to session, token leakage over a GET request could lead to more serious CSRF attacks that target POST APIs. Tokens may leak because URLs for GET requests may be stored in the browser history, server and proxy log files, bookmarks, etc. Attackers may then use techniques such as browser history sniffing to discover CSRF tokens [63].

Potentially extractable In some situations, attackers may be able to extract CSRF tokens directly. For example, attackers could convince users to drag content that includes tokens from one web frame (the victim) to another (the attackers), or to copy-and-paste the token into the attacker's frame [69]. Attackers have used these techniques to trick Facebook users into exposing their CSRF tokens [124]. Researchers have also shown that many web sites are vulnerable to CSRF token extraction through a variety of HTML and script injection attacks [121, 26]. Recent work shows how CSRF tokens may be extracted using only cleverly formed CSS [57].

Error-prone manual implementation Tokenization has many "moving parts", and custom implementations may thus be quite prone to errors. For example, a developer can easily overlook an important location where tokenization is needed and leave the application open to CSRFs. On the other hand, if the developer is overzealous with tokenization, tokens can leak; this is particularly bad if tokens are not ephemeral (like with Flickr) or made easily reversible to a session ID as suggested by some tutorials [43].

Frameworks confuse developers On the other end of the spectrum, developers using a CSRF protection framework may misuse it, or they may falsely believe that it protects them from all types of CSRF attacks. For example, some frameworks do not tokenize AJAX requests [41, 19]. Other frameworks may only rewrite forms generated using web platform calls, leaving forms written using raw HTML unprotected. As another example, without understanding how CSRF frameworks work, developers can accidentally cause cross-domain POSTs to send a CSRF token to an untrusted third party. Finally, it's possible for developers to unwittingly accept GET requests while thinking the data came from POSTs — popular libraries such as Perl's CGI.PM module allow a developer to fetch a parameter without caring if it came in via a GET or POST request. Thus, attackers using GET requests would still succeed [7].

Poor third-party subcomponent support Many modern web development platforms (such as Drupal, Django, and Ruby on Rails) allow developers to use third-party components or plug-ins for added functionality. By integrating a poorly written component, developers might introduce a CSRF into their application. In such cases, it may be difficult to check whether a component correctly protects all actions, especially if it has a large code base [108].

Language dependence For large, complicated web applications (such as large e-commerce sites) each part of the page may be generated using a different programming language. CSRF token generation and verification may need to be implemented separately by every one of those components.

Origin Checking

Besides tokenization, web application developers may opt to use the proposed `Origin` HTTP header [12, 119], a privacy-preserving replacement for the `Referer` header. Like its predecessor, the Origin header is used by browsers to convey the originating domain of a request to the server. Web application developers can use that information to decide whether a request originated from a web origin the application trusts and hence is a legitimate request. For example, *bank.com* may trust *broker.com* and treat any request having an Origin header value of *broker.com* or *bank.com* as a valid state-modifying request, and treat all other requests as untrusted. As of late 2012, the Origin header is supported in Chrome and Safari.

In practice, the Origin header has restrictions that complicate and impede the ability of developers to use it as an anti-CSRF mechanism. Next, we discuss two such challenges.

No path information To preserve privacy, the Origin header does not contain the path part of the request's origin. For example, suppose *bank.com* wants to assert that only requests from *broker.com/accounts/* can have side-effects on *bank.com*, but requests from any other location from *broker.com*, such as *broker.com/forum/*, may not. By design, the Origin header lacks the path information necessary for a web application to make this decision; making it impossible for *bank.com* to distinguish a request for the (legitimate) *accounts* page from a CSRF attack inside a malicious post on the *forum* page.

One workaround could be for *broker.com* to separate the */forum/* and */accounts/* web applications into multiple subdomains (*e.g.*, *accounts.broker.com* and *forum.broker.com*), but subdomain configuration may be problematic. Many open-source web applications (such as forums or blogs in a box) do not support subdomain creation via scripts, instead forcing the web developer to manually perform potentially confusing server configuration. Moreover, subdomain creation may be disallowed or may incur additional cost from hosting providers [109]. Finally, if using TLS, web developers would have to procure additional costly TLS certificates for subdomains or pay more for a wildcard certificate. Because of these complications, developers often separate web application modules by path rather than by subdomain.

Origin sent as null If a request originated from an anchor tag or a window navigation command such as *window.open*, the Origin header is sent as null. The rationale is that "hyperlinks are common ways to jump from one site to another without trust. They should not be used to initiate state-changing procedures" and "many sites allow users to post links, so we don't want to send Origin with links" [84]. The suggested workaround is to convert all anchor tags and window navigation calls to a form GET. Such overhauls may be tough for maintainers of legacy sites, making it difficult for them to rely on the Origin header for CSRF protection.

Additionally, for business and security reasons, many sites (such as banking sites) do not allow users to post links. For these sites, using anchor tags as trusted sources of state-changing requests may be a valid decision. However, since the Origin header is *null* for all requests originating from anchor tags, these sites would be forced into using forms instead if they wish to leverage the Origin header.

2.3.3 Client-side Defenses

Some defenses check for CSRF on the client side (browser) rather than the server side. Client-side solutions first identify "suspicious" cross-origin requests and then either block the request outright [122] or strip the request of all cookies and HTTP authentication data [102, 33, 66, 76, 75]. The biggest advantage of client-only solutions is that they do not require any web site modifications, relying instead on either heuristics or central policy sources. Unfortunately, this makes them prone to false positives which break legitimate sites and/or false negatives which fail to detect CSRF.

CsFire

CsFire [102, 33] is a browser plug-in that strips cookies and authentication headers from outgoing HTTP requests that are deemed unsafe. By default, all cross-origin requests are considered unsafe, except for requests that pass a fairly strict set of rules to identify trust relationships between sites (*e.g.*, *a.com* may make a request to *b.com* if *b.com* redirected to *a.com* earlier in the same browsing session). This policy breaks many legitimate sites, so CsFire maintains a whitelist of exceptions on a central server (maintained by the CsFire authors) and also allows users to add exceptions manually.

Unfortunately, in our experience, CsFire still results in false positives and breaks legitimate functionality during normal browsing, such as logging into Flickr or Yahoo via OpenID. This shows that such an architecture would need to rely on users and/or CsFire developers to constantly update the list of whitelisted sites that are allowed to send authentication data. Moreover, existing sites may change at any moment and break existing CsFire policies. We believe maintaining such an exception list for the whole web is close to impractical, as is relying on users to manually add exceptions.

In addition, CsFire's policies make a binary decision to either send or strip *all* cookies and HTTP authentication headers. This may cause overly restrictive policies, as cross-origin requests could harmlessly include cookies that are not used for authentication (such as user settings). Worse, this may also lead to insecure policies: if a site needs a non-sensitive cookie in order to function, a user may be tempted to add a CsFire exception for such a site, which would allow requests to the site to attach *all* cookies, including sensitive authentication cookies, which could lead to CSRF.

RequestRodeo

RequestRodeo [66] is a client-side proxy, positioned between the browser and web sites, that stops CSRFs by stripping authentication data from suspicious requests. Requests may contain authentication headers only if they are initiated "because of interaction with a web page (*i.e.*, clicking on a link, submitting a form or through JavaScript), and if the URLs of the originating page and the requested page satisfy the same-origin policy."

To trace request sources, the proxy rewrites "HTML forms, links, and other means of initiating HTTP requests" with a random URL token. The token and response URL is then stored by the proxy for future reference. When the browser makes a request, the proxy looks up the URL token and compares the destination URL with the request's referring URL. If they do not match, the request is considered suspicious. The proxy also detects intranet cross-domain requests and validates them with explicit user confirmation.

This approach has several downsides. First, it mandates that all user traffic must go through a TLS man-in-the-middle proxy. Second, many applications use JavaScript and other active content to dynamically create forms and issue HTTP requests directly from the client, making rewriting of such requests impossible in a proxy. Third, rewriting all HTML content may have unpleasant latency implications. Finally, some cross-domain requests can be legitimate (*e.g.*, for federated login/single-sign-on or for showing users personalized content when they visit a link); RequestRodeo does not support these cases.

BEAP

Browser-Enforced Authenticity Protection (BEAP) [75] attempts to infer whether a user intended the browser to issue a particular request. User intent is determined via a browser extension that monitors user actions. For example, if the user enters a URL into the address bar, the request is considered intended. However, if the user clicks on a link in an e-mail message or from the browser history list, the request is considered unintended. The browser extension strips unintended requests of "sensitive" authentication tokens, which include (1) all *session* (non-persistent) cookies sent over POST requests and (2) HTTP Authorization headers. Persistent cookies are treated as non-sensitive, as are session cookies sent over

GET requests. BEAP authors note that these rules were generated by analyzing real-world applications. However, this analysis does not hold today: for some web sites we analyzed, such as Woot or Google, some sensitive cookies were persistent.

Cross-site Request Forgeries: Exploitation and Prevention

In their 2008 paper [122], Zeller and Felten identified a variety of high-profile CSRF vulnerabilities and gave general guidelines for how to prevent them, recommending tokenization and using POSTs for state modifying requests. The authors also provided two tools for CSRF prevention: a plug-in for developers to perform automatic tokenization, and a plug-in for browsers to block cross-domain POST requests (unless the site has an Adobe cross-domain policy that specifies exceptions).

We believe the auto-tokenization plug-in is useful, but suffers from the same drawbacks as other tokenization frameworks (Section 2.3.2). We believe that all client-side solutions will be both too coarse in how they handle POST requests and too permissive when they freely pass any GET requests through, including potentially dangerous GETs (see examples in Section 2.3.2).

Adobe Cross-Domain Policy

Adobe cross-domain files specify how Adobe clients (such as Flash Player and Reader) should make requests across domains. These policies are hosted on remote domains and grant “read access to data, permit a client to include custom headers in cross-domain requests, and are also used with sockets to grant permissions for socket-based connections.” [4]

We believe these policies do not provide enough control over ambient authority to prevent CSRFs without restricting functionality. For example, developers cannot specify names of cookies to which the policies apply or control valid embedding.

2.4 Summary

We examined a number of proposed alternatives for password-based authentication and found that previously proposed alternatives have non-trivial security, usability, and deployability drawbacks. While some approaches increased security (such certificate-based authentication in Section 2.2.8), they simultaneously reduced usability. While other approaches such as Single Sign-On might be considered more usable than passwords, they have deployability drawbacks. Since no approaches could "best" passwords, it seems logical that passwords still reign as the main user authentication mechanism.

Similarly, for authorization we found that all current approaches have significant drawbacks. For example, server-side approaches required nontrivial development effort, failed to protect web applications against state-modifying `GET` requests, and were vulnerable to some types of attacks (*e.g.*, token extraction). Client-side approaches used heuristics to identify "suspicious" requests and "important" authentication tokens. However, because client-side solutions cannot know developer intentions, they were prone to either false positives which broke sites or to false negatives which missed attacks due to being too lenient with allowed requests.

In the end, we found that currently existing approaches for performing both user authentication and authorization are insufficient. We believe new approaches are needed in both areas that carefully consider all of the challenges, stakeholders, and tensions. For authentication, we believe that approaches utilizing ubiquitous second factor devices (such as smartphones) have a promising future; we present such a system in Chapter 4. For authorization, we anticipate the development of new policy-like solutions that allow developers to have more control of how cookies (and other credentials) are sent by the browser; we present such a system in Chapter 5. However, first, in Chapter 3 we propose Origin Bound Certificates with the goal of strengthening the transport layer in order to support secure authentication and authorization.

Chapter 3

TLS ORIGIN-BOUND CERTIFICATES: STRENGTHENING THE TRANSPORT LAYER

In this chapter we examine the fundamental transfer layer upon which user authentication and authorization are built. We consider the ways in which this layer is vulnerable and how previous approaches have failed to fix it. We then present a novel approach called *TLS Origin-Bound Certificates* for strengthening this layer and providing a strong foundation for addressing user authentication and authorization. TLS Origin-Bound Certificates were originally described in a 2012 publication [40]. We begin by providing a high-level overview and motivation for TLS Origin-Bound Certificates.

3.1 Motivation and Overview

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

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

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 [86]), cross site scripting attacks, or adversaries that manage to sniff the network or insert themselves into the network between the client and server [3, 25]).

The academic community, has known of authentication mechanisms that avoid the weaknesses of bearer tokens since before the proliferation of the Internet. 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 [37] or fail outright [77].

[2]Bearer tokens, originally called "sparse capabilities" [113], were widely used in distributed systems, well before the web.

In this chapter, 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 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 the efforts in gaining community support for an IETF draft [10], 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: in the proposed scheme, 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. In subsequent chapters, we will also show how TLS-OBC's enable other types of systems.

3.2 Design

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 3.2.4). We begin by carefully examining our threat model.

3.2.1 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 [99], mounting a man in the middle (MITM) attack through stolen server TLS certificates [3], or utilizing man-in-the-browser malware [86]. 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 [107].

In this chapter 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. Protecting the application-layer user logins, is handled in the next chapter and is mostly outside the scope of this chapter. 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 himself 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 chapter, 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 3.2.4. Attacks that target user credentials during the initial TLS connection, rather than authentication tokens during subsequent TLS connections, are dealt with in the following chapter.

3.2.2 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 2.2.8), 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.

[3]We use the same notion as TAOS [117] does, of a cryptographically strong link between two nodes.

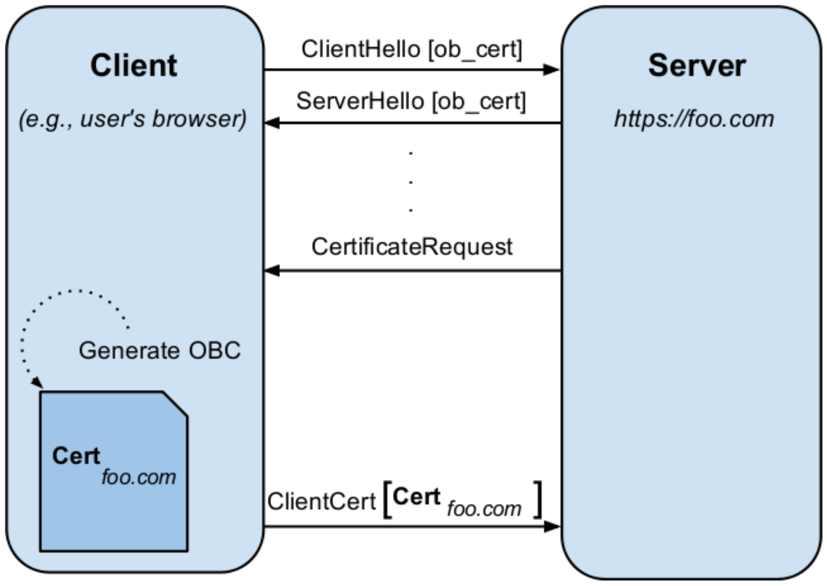


Figure 3.1: TLS-OBC extension handshake flow.

- 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.

3.2.3 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 3.1 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, 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.

3.2.4 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 3.2.4 we explain how HTTP cookies can be bound to TLS channels using TLS-OBC. In Section 3.2.4 we show how federation protocols (such as OpenID or OpenID Connect) can be hardened against attackers, and in Section 3.2.4 we turn briefly to application-level user authentication protocols.

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).

Note that this different from prior work on hardening cookies such as the method presented by Murdoch cookies are toughened by encoding values not only based on on a secret server key, but also on a hash of the user's password [85]. 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.

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 3.2. 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 the client) to the backend service. The

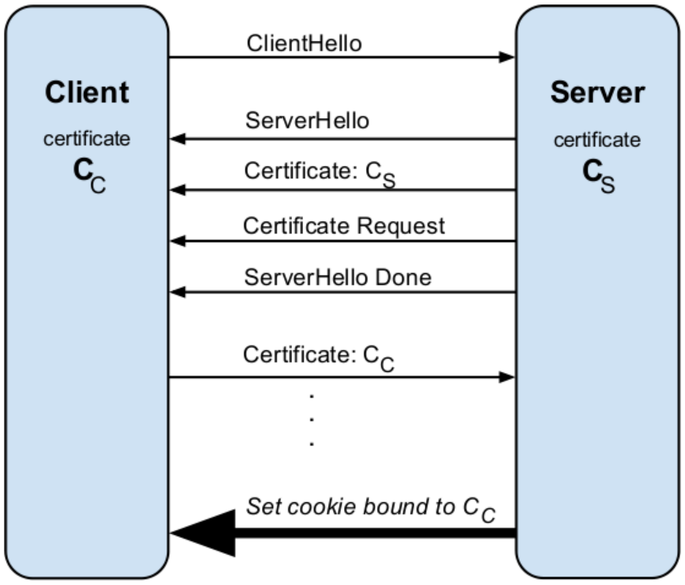


Figure 3.2: Process of setting an OBC bound cookie

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.

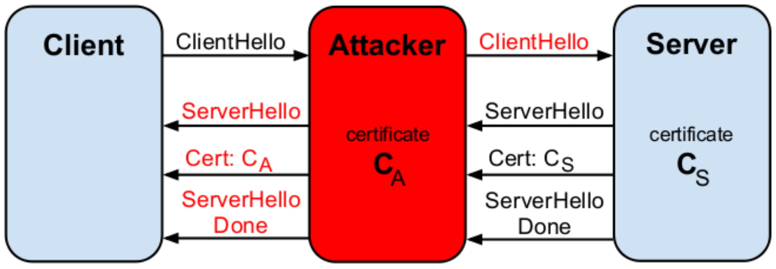


Figure 3.3: A MITM attack during a TLS handshake

Channel-Bound Cookies Protect Against MITM

As mentioned earlier, TLS MITM attacks happen and some can go undetected (see Figure 3.3 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 3.2. 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 3.4. 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 log the user in as another identity. We cover this more in Section 3.2.4 and in an upcoming report.

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 [96],

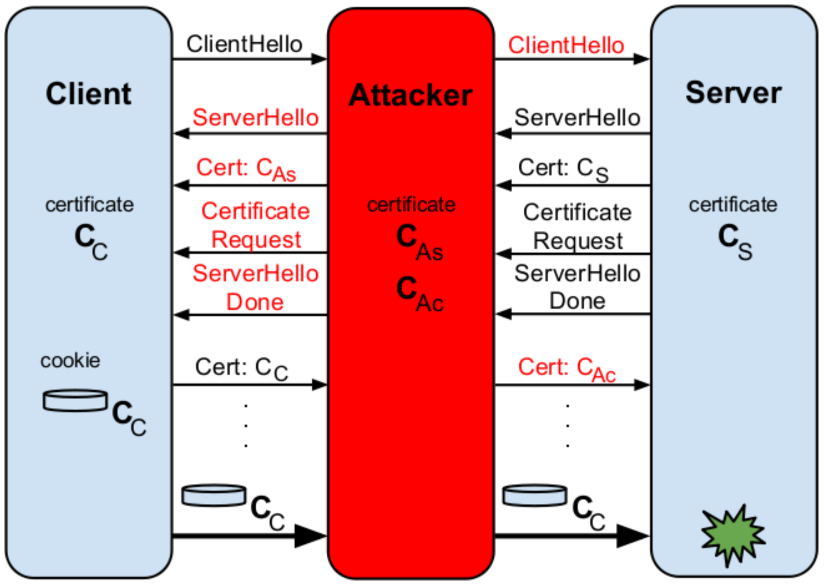


Figure 3.4: Using OBCs and bound cookies to protect against a MiTM attack. The server recognizes a mismatch between the OBC to which the cookie is bound and the cert of the client (attacker) with who it is communicating.

OpenID Connect [103], and BrowserID [2] 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* [59] communication mechanism that allows distinct windows within the browser to send messages to each other using inter-process communication 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 3.5 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

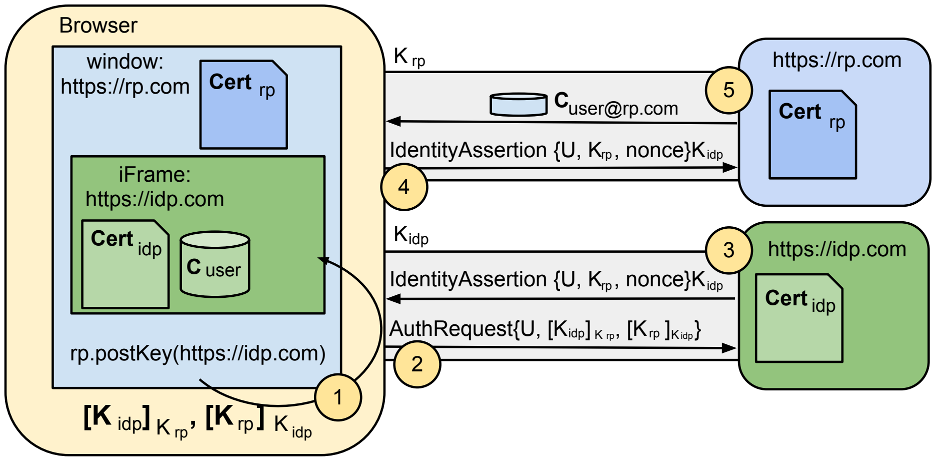

Browser
window: https://rp.com
Cert rp
iFrame: https://idp.com
Cert idp
C user
rp.postKey(https://idp.com)
[K idp] K rp, [K rp] K idp
1
K rp
C user@rp.com
5
IdentityAssertion {U, Krp, nonce}Kidp
4
Kidp
3
IdentityAssertion {U, Krp, nonce}Kidp
AuthRequest{U, [Kidp] K rp, [Krp]Kidp}
2
https://rp.com
Cert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

Figure 3.5: Simplified federation protocol authorization flow using PostKey and OBCs.

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.

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 chapter. 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 our previous workshop paper [30], where we frame authentication in terms of protected and unprotected login. We 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). We 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. We argue that websites should move all possible authentications to protected login, minimize unprotected login, and then *alert* users when unprotected logins occur. The chapter 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 3.4, 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. The system presented in Chapter 4 addresses this issue.

3.3 Implementation

In order to demonstrate the feasibility of TLS origin-bound certificates for channel-binding HTTP cookies, we implemented the extensions discussed in Section 3.2. 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.

3.3.1 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 [18]. 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.

3.3.2 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.

3.4 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.

3.4.1 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 3.4.1 were gathered with the Chromium benchmarking extension [62] and the HTML5 Navigation Timing [90] JavaScript interface.

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.

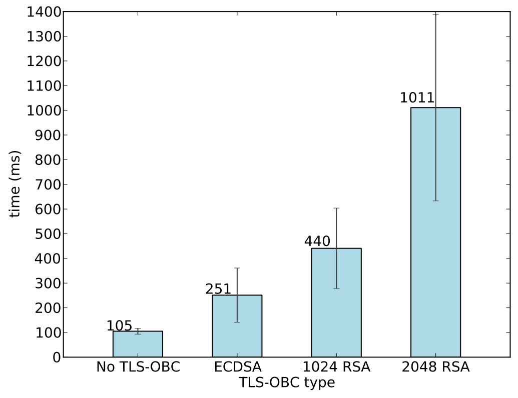


Figure 3.6: Observed Chromium network latency (ms) with TLS-OBC certificate generation

The first test shown in Figure 3.6 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.

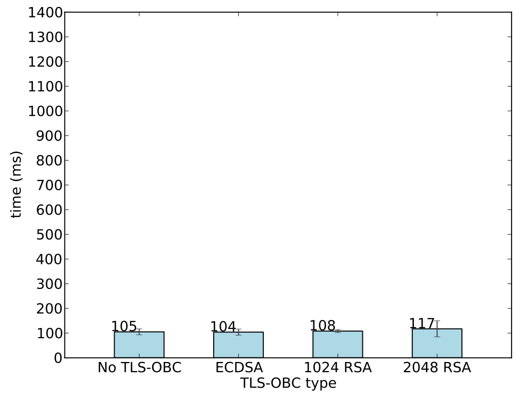


Figure 3.7: Observed Chromium network latency (ms), TLS-OBC certificate pre-generated

The results shown in Figure 3.7 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.

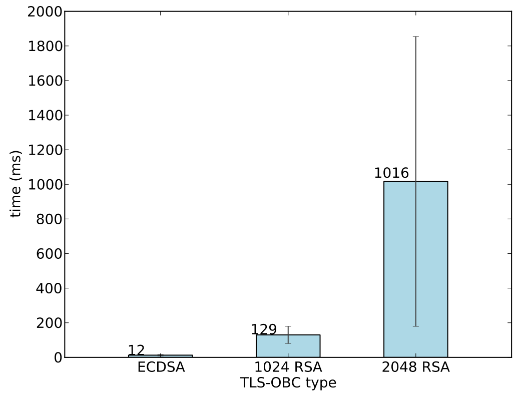


Figure 3.8: NSS certificate generate times (ms)

The differences between the latencies observed in Figures 3.6 and 3.7 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 3.8, 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.

3.4.2 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 3.9 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%.

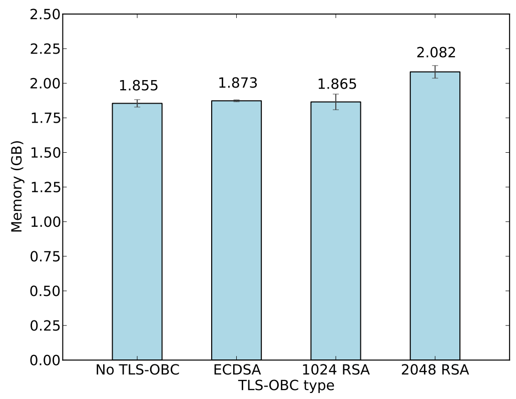


Figure 3.9: Server-side memory footprint of various client-side key sizes.

Figure 3.10 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 3.11 through Figure 3.14 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.

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 [17], however the ECDSA server costs might be worth the client-side benefits.

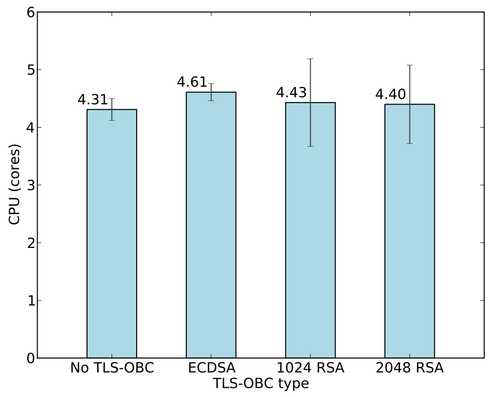


Figure 3.10: Server-side CPU utilization for various client-side key sizes.

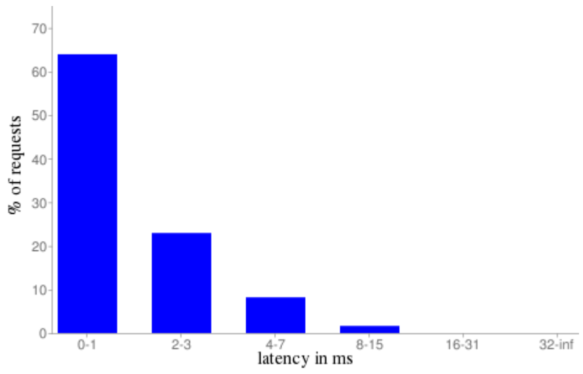


Figure 3.11: Latency without client certificates

3.5 Discussion

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.

3.5.1 Domain Cookies and TLS-OBC

In Section 3.2.4 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

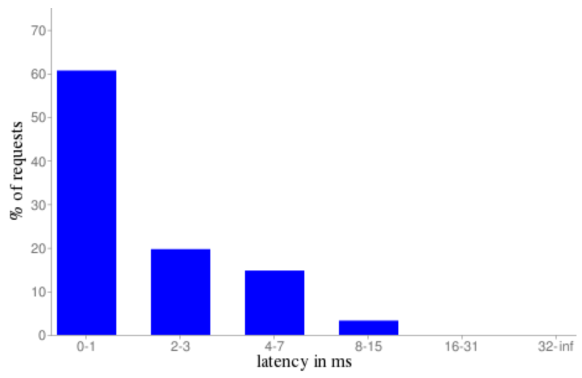


Figure 3.12: Latency with 1024-bit RSA certificate

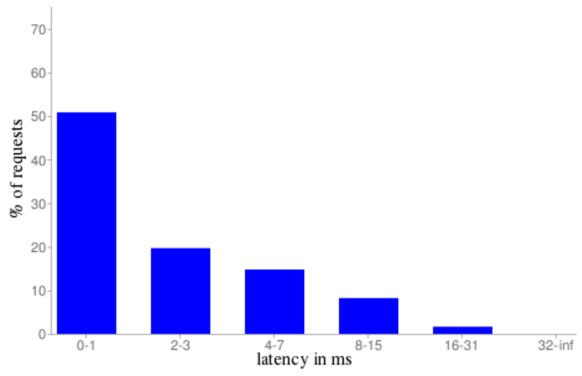


Figure 3.13: Latency with 2048-bit RSA certificate

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.* [23] 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

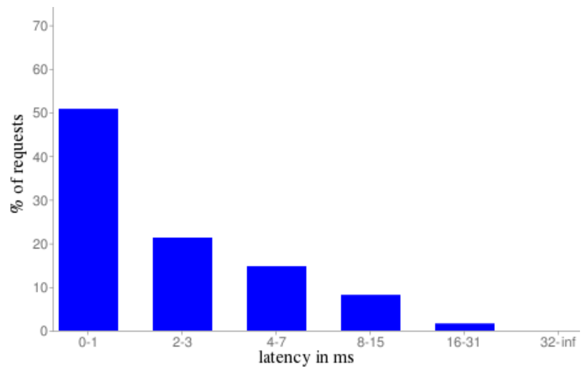


Figure 3.14: Latency with 163-bit ECDSA certificate

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 chapter 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.

3.5.2 Privacy

The TLS specification [39] 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.

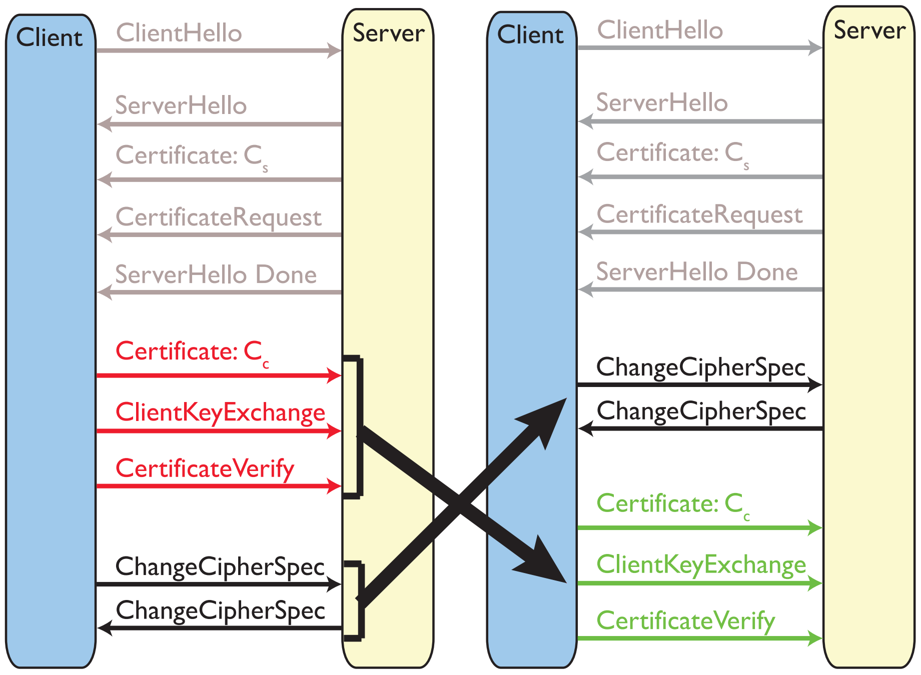


Figure 3.15: TLS encrypted client certificates

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 3.15 shows the effect this extension has on TLS message ordering.

3.5.3 SPDY and TLS-OBC

The SPDY [115] protocol multiplexes several HTTP requests over the same TLS connection, thus achieving higher throughput and lower latency. SPDY is enabled in the Chrome, Firefox, and Opera browsers. 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 [97] with the private key corresponding to the public-key certificate.

3.5.4 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.

Consider, for example, the problem of replay attacks: A man-in-the-middle might have observed an HTTP request that deletes a user's email inbox. We would like it to be the case that the man-in-the-middle can't re-use the same request to later delete the user's inbox at will. Introducing a timestamp to solve this problem is risky: what if the client (who issues the request to delete a user's inbox) and the server (who is supposed to execute the request) don't share sufficiently synchronized clocks? To solve this issue, we need to include a server-issued challenge in the request signature, requiring extra round-trips.

Another example is session renegotiation. Presumably, in order to save processing cost, the client wouldn't actually perform a public-key signature operation every time it issued an HTTP request. Instead, it would somehow negotiate a symmetric signing key with the server and then use that symmetric signing key to create (cheaper) symmetric-key signatures on the HTTP request. What if the server wants the client to prove that it is still in possession of the original private key (we would have to do this periodically to detect attacks in which the negotiated symmetric key has been stolen by the attacker)? Would we have to pause all pending HTTP requests to the server, run a session-renegotiation protocol, and then restart the HTTP requests? Would this be possible without impacting user-visible 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.

3.6 Summary

In this chapter we presented TLS origin-bound certificates — a new approach to public key-based client authentication. 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. In the following chapter, we build on top of origin-bound certificates and show how it is possible to design a better system for authenticating initial user logins.

Chapter 4

PHONEAUTH: STRENGTHENING USER AUTHENTICATION

In today's on-line systems user authentication is at an impasse: by and large, despite our best efforts, we have not been able to move past username and password authentication. In this Chapter, we present *PhoneAuth*, a system for strengthening user authentication through opportunistic cryptographic identity assertions which leverage the TLS Origin-Bound Certificates we presented in Chapter 3. PhoneAuth was initially published in a 2012 publication [31]. We begin with the motivation and overview of the PhoneAuth system.

4.1 Motivation and Overview

The most common mechanism for users to log into web sites is with usernames and passwords. They are simple to implement on a server and they allow web sites to easily interact with users in a variety of ways.

There are a variety of problems with these simple approaches, not least of which is that many users will reuse passwords across different web sites [51, 21], at which point the compromise of one web site leads to compromise of others [105, 24]. For users who might want to remember distinct passwords, the cognitive burden makes it impossible at scale. Furthermore, users faced with impostor web sites or forms of phishing attacks often give up their credentials. It should then come as no surprise that large numbers of users see their online accounts accessed by illegitimate parties every day [104, 29], causing anywhere from minor annoyances, to financial harm, to very real threats to life and well-being [55, 94].

As practitioners of computer science we know that passwords offer poor security, yet here we are, four decades after the invention of public-key cryptography and two decades into the history of the web, and we still use passwords. A recent study by Bonneau *et al.* [22] sheds some light onto why that is the case: none of the 35 studied password-replacement mechanisms are sufficiently usable or deployable in practice to be considered a serious alternative or augmentation to passwords, which is unfortunate since many of the proposals are arguably more "secure" than passwords. This includes mechanisms that employ public-key cryptography (such as CardSpace [77] or TLS client certificates [36])[1]. Public-key cryptography would otherwise be an elegant solution to the security problems with passwords outlined above: it would allow us to keep the authentication secret (a private key) *secret*, and to not send it to, and store it at, the parties to which users authenticate (or their impostors).

We have set out to take a fresh look at the use of public-key cryptography for user authentication on the web. We are cognizant of the shortcomings of previous attempts, and of the presence of public-key-based mechanisms in the list of failed authentication proposals in the Bonneau *et al.* study. Yet we argue that public-key-based authentication mechanisms can be usable if they are carefully designed. Our main contribution in this chapter is one such design we call PhoneAuth, which has the following properties:

- It keeps the user experience of authentication invariant: users enter a username and password directly into a web page, and do not do anything else.

- It provides a cryptographic second factor in addition to the password, thus securing the login against strong attackers.

- This second factor is provided opportunistically, *i.e.,* only if and when circumstances allow (compatible browser, presence of second factor device, and so on). We provide fallback mechanisms for when the second factor is unavailable.

Though PhoneAuth does have several operational requirements, we believe that they are reasonable based on current technical trends and do not hinder the deployability of PhoneAuth. See Section 4.5 for more details.

In Section 4.2 we establish the threat model and goals for our system. Section 4.2.3 outlines the system at a high level while Section 4.3 delves into practical implementation details.

[1]See Chapter 2 for an in-depth review of these and other authentication technologies.

The Bonneau *et al.* study [22] presents a framework of 25 different usability, deployability, and security "benefits" that authentication mechanisms should provide. We rate our system against this framework and provide other evaluations in Section 4.4, discuss potential future directions in Section 4.5, and summarize in Section 4.6.

4.2 Design

4.2.1 Goals

Given the lessons from previous chapters, we take a fresh look at strong user authentication on the web. The goals we have set for our work are outlined below:

- The authentication process should not solely rely on token-like credentials (*e.g.*, cookies) as these can be stolen. Instead secret information should be used to prove the user's identity wihtout revealing the secret information. For exapmle, some form of public-key cryptography needs to be involved in the authentication process. Not only does this allow for the authentication secret (the private key) to remain protected on the client device, it also means that this secret is unknown to the user and therefore cannot be stolen through phishing.

- The identity of the user must be established and proven *above* the transport layer. Otherwise, the inability of users to see the context in which they are authenticating leads to poor user experience and privacy problems as we observed in TLS client authentication.

- The action of logging into a website should remain invariant: users type a username and password into a web page (not the browser chrome or other trusted device), and then are logged in. Apart from helping with learnability for the user, this also helps with deployability: websites do not have to re-design their login flows and can gradually "onboard" those users that possess the necessary client software into the new authentication mechanism.

- The design should work well both in a world with very few identity providers, or in a world where every website runs its own authentication services.

- Users need a fallback mechanism that allows them to log in just with something "that they know" in case the public-key mechanism does not work (*e.g.*, they are on a device that does not support the new mechanism, or the device responsible for doing the public-key operation is not available), or in case they *do* have a legitimate need to hand over their credential to a third party (for example, someone asking their more tech-savvy friend/child/parent to debug a problem with their account).

4.2.2 Threat Model

Another goal of our work is to protect users in the face of a strong adversary. In particular, we assume the following threat model: We allow adversaries to obtain the user's password — either through phishing or by compromising weaker sites (for which the user has reused a password).

We assume that the attacker can perform a man-in-the-middle attack on the connection between the user and the server to which user is authenticating. For TLS based connections, this attack assumes that the attacker has a valid TLS certificate for the site to which the user is authenticating, thus allowing him to perform TLS man-in-the-middle attacks. We even allow an attacker to obtain *the correct* certificate for the victim site (presumably by stealing the site's private key). This capability is extremely powerful and would even cause browser certificate pinning [89] to fall prey to a TLS man-in-the-middle attack. Though we have not seen reports of such attacks in the wild, security practitioners do believe such attacks are possible [71].

Finally, we allow the attacker to deploy certain types of malware on the user's machine — for example those that perform keylogging. However, we assume the attacker is not able to simultaneously perform an attack on both the network connection and the physical radio environment near the user. For example, these constraints make malware that is able to control (and potentially man-in-the-middle or denial of service) both the LAN NIC and the Bluetooth chip out-of-scope, but leave in-scope malware that rides in the browser session.

Finally, we assume the attacker is not able to simultaneously compromise the same user's PC and user's personal device.

4.2.3 Architecture

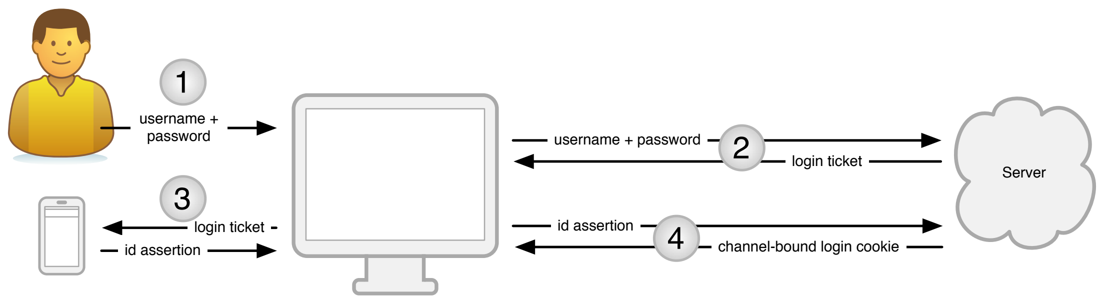


Figure 4.1: PhoneAuth overview

Architectural Overview

Our PhoneAuth authentication framework meets the goals above by *opportunistically providing cryptographic identity assertions* from a user's mobile phone while the user authentications on another device. Figure 4.1 explains this process:

- In step 1, the user enters their username and password into a regular login page, which is then sent (in step 2) to the server as part of an HTML form.

- Instead of logging in the user, the server responds with a *login ticket*, which is a request for an additional identity assertion (more details below).

- In step 3, the browser forwards the login ticket to the user's phone, together with some additional information about key material the browser uses to talk to the server.

- The phone performs a number of checks, and if they succeed, signs the login ticket with a private key that is known to the server as belonging to the user. The signed login ticket constitutes the *identity assertion.* It's *cryptographic* because we use public-key signatures to sign the browser's public key with the user's private key.

- In step 4, the browser forwards the identity assertion to the server. The server checks that the login ticket is signed with a key belonging to the user identified in step 2, and if so, logs in the user by setting a cookie that is *channel-bound* to the browser's key pair (see below). As a result, the phone certified the browser's key pair as speaking for the user, and the server records this fact by setting the respective cookie.

We now provide additional notes about the overall architecture:

Opportunistic Identity Assertions We do not assume that every user will have a suitable mobile phone with them, or attempt logins from a browser that supports this protocol. That is why in step 4 the browser can also return an error to the server. If this is the case, the user has performed a traditional login (using username + password), and in the usual manner (by typing it into a login form), which means that the protocol essentially reduces to a traditional password-based login. The cryptographic identity assertion is *opportunistic*, *i.e.*, provided when circumstances allow, and omitted if they do not.

The server may decide to treat login sessions that carried a cryptographic identity assertion differently from login sessions that did not (and were only authenticated with a password). For example, the server could decide to notify the user through back channels (SMS, email, *etc.*), similar to Facebook's Login Notifications mechanism. The server could also restrict access to critical account functions (*e.g.*, changing security settings) to sessions that did carry the identity assertion. We call this mode of PhoneAuth *opportunistic* mode.

An alternative mode of PhoneAuth is *strict* mode, in which the server rejects login attempts that did not carry a cryptographic identity assertion. This is more secure, but comes at the cost of disabling legacy devices that can't produce identity assertions. The decision whether to run in strict or opportunistic mode can either be made by the server, or it can be made on a per-user basis: Security-conscious users could opt into strict mode, while all other users run in opportunistic mode. A user who has opted into strict mode would not be able to log in when his phone was unavailable, while a user has not opted in (*i.e.*, runs in opportunistic mode) would simply see a login notification or a restricted-access session when logging in without his phone.

User Experience The user does not need to approve the login from the phone. The server will only issue a login ticket if the user has indicated his intent to log in by typing a username and password. When the phone sees a login ticket, it therefore knows that user consent was given, and can sign the login ticket without further user approval.

This means that there is no user interaction necessary during a PhoneAuth login, other than typing the username and password. If the phone and browser can communicate over a sufficiently long-range wireless channel, the user can leave the phone in their pocket or purse, and will not even need to touch it.

Protected Logins In an earlier workshop publication [30] we introduced the concept of *Protected Login* whereby we grouped logins into two categories — protected and unprotected. Protected logins are those that are a result of strong, unphishable credentials (*e.g.*, a cookie or an *identity assertion* in our case). Unprotected logins are logins that result from weaker authentication schemes (*e.g.*, just a password or a password and secret questions). Following this nomenclature, opportunistic PhoneAuth attempts to perform a protected login, but reverts to an unprotected login if the identity assertion is not available.

Our work showed that only *first* logins from a new device need special protection via a second factor device — subsequent logins can be protected by channel-bound cookies (see below) that were set during the first login. This observation further shows the usability of our scheme: we obtain strong protection with a login mechanism that quite literally asks the user to do nothing but type their username and password, and (assuming a wireless connection between browser and phone) bring their phone into the proximity of the browser *only during the first login from that browser*.

TLS Channel IDs The security of PhoneAuth relies on the concept of TLS origin-bound certificates (OBC) introduced in Chapter 3. TLS-OBC is currently an experimental feature in Google's Chrome browser and is under consideration by the IETF as a TLS extension.

OBCs are TLS client certificates that are created by the browser on-the-fly without any user interaction and used during the TLS handshake to authenticate the client. OBCs don't carry any user-identifying information and are not used directly for authentication. Instead, they simply create a TLS "channel" that survives TLS session resets (the client re-authenticates itself with the same OBC to the server, recreating the same channel). We can *bind* an HTTP cookie to this TLS channel by including a *TLS channel ID* (a hash of the client's OBC) as part of the data associated with the cookie. If the cookie is ever sent over a TLS channel with a different channel ID (*i.e.*, from a client using a different OBC), then the cookie is considered invalid.

At the heart of PhoneAuth is the idea that the server and browser will *each communicate their view of the TLS channel between them to the user's phone.* The server uses the login ticket as the vehicle to communicate its view of the TLS channel ID to the phone. The browser communicates the TLS channel ID directly to the phone. If there is a man-in-the-middle between browser and server (which doesn't have access to the browser's private OBC key), these two TLS channel IDs will differ (the server will report the ID of the channel it has established between the man-in-the-middle and itself, while the browser will report *its* channel ID to the phone). Similarly, if the user accidentally types his credentials into a phishing site (which then turns around and tries to submit them to the server), the two TLS channel IDs will differ.

The user's phone compares the two TLS channel IDs and will not issue an identity assertion if they differ, causing a login failure in strict mode, and an unprotected login in opportunistic mode. The phone can then potentially alert the user that an attack may be in progress or send a message to the server if a cellular connection is available.

Protocol Details

In describing this protocol, we also describe inline how our design addresses risks such as credential reuse, protocol rollback attacks, TLS man-in-the-middle attacks, and phishing.

Recall that in step 2, the user's entered username and password are sent to the server. The server then verifies the credentials and generates a *login ticket.* The *login ticket* structure

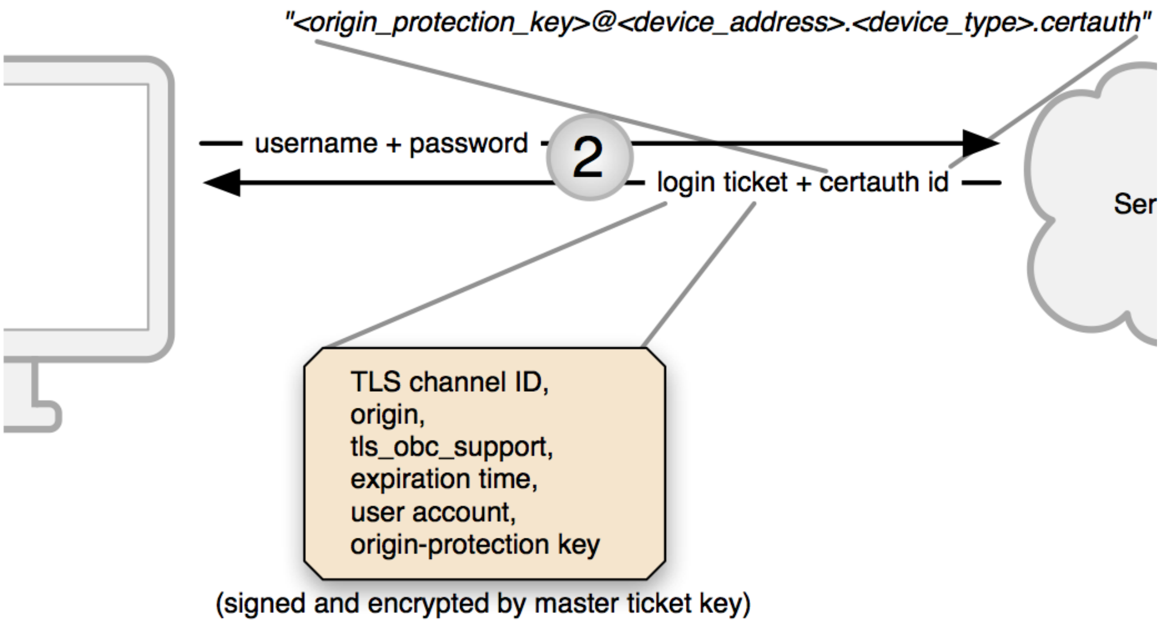

"<origin_protection_key>@<device_address>.<device_type>.certauth"
username + password
2
login ticket + certauth id
Ser
TLS channel ID,
origin,
tls_obc_support,
expiration time,
user account,
origin-protection key
(signed and encrypted by master ticket key)

Figure 4.2: Login ticket structure

is shown in Figure 4.2. The ticket contains a TLS channel ID (for binding the ticket to the TLS channel), the web origin of the webapp, the expiration time (to prevent reuse), whether the login request used TLS-OBC (to prevent rollback attacks), an account (to bind the ticket to a user), and an origin protection key (to allow the phone to decrypt assertion requests sent over insecure mediums). The *login ticket* is encrypted and signed using keys derived from a per-account master secret known only to the server and the user's phone; we describe later how the server and phone derive this master secret key. Observe that the *login ticket* is opaque to the browser — it can neither "peek" inside nor modify the login ticket.

The server sends the *login ticket* along with a *certauth id* that tells the browser how to contact the user's phone. The *certauth id* is in the form of:

<origin_protection_key>@<device_address>.<device_type>.certauth

After receiving the login ticket, the browser generates an *assertion request* (shown in Figure 4.3) which includes the *login ticket* along with some metadata about the TLS session. The metadata also includes the TLS channel ID as seen by the browser and helps to prevent

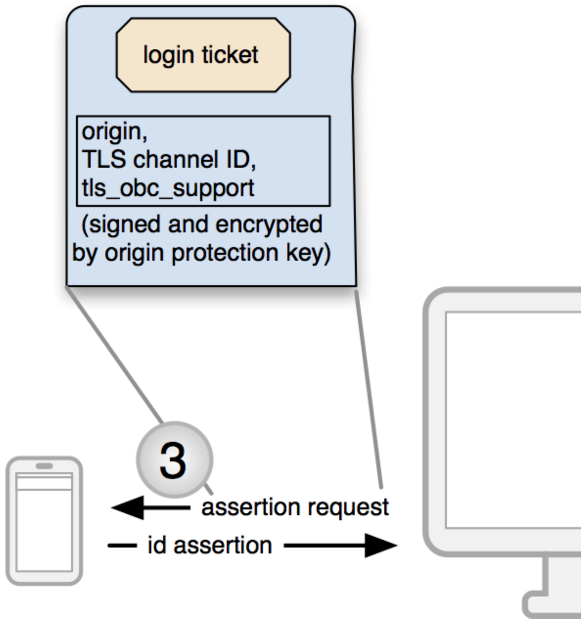


Figure 4.3: Assertion request structure

a TLS man-in-the-middle attack. This data is encrypted and authenticated under the origin protection key obtained from the *certauth id* (obtained in step 2).

The browser sends the *assertion request* to the user's phone in step 3. The phone then unpacks and validates the *assertion request*, making sure that the TLS channel IDs match, that the device can vouch for the requested user, and that the *assertion request* was indeed for this device. Next, the device generates an *identity assertion*. The *identity assertion* simply contains the login ticket signed by the private key of the user's personal device. The phone sends the *identity assertion* to the browser, which forwards it to the webapp (shown in Figure 4.4). The webapp unpacks and validates the assertion (again checking for TLS Channel ID mismatches) and incorrect signatures.

Finally, the webapp gives the browser a channel-bound cookie, thus completing the protected login.

Additional Security Discussion Though we discuss several attacks and their mitigations above, we now highlight several additional aspects of our security design. Observe

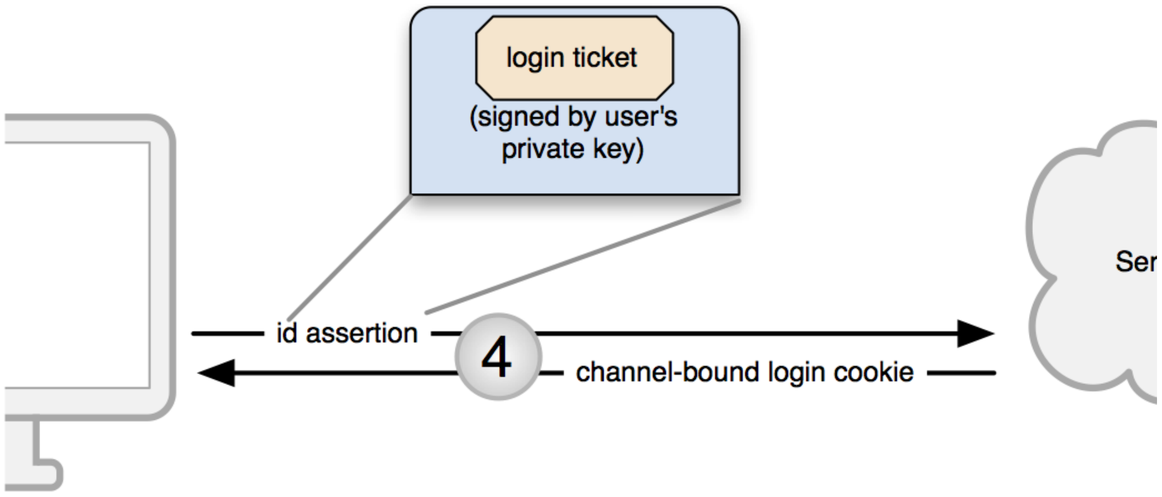


Figure 4.4: Identity assertion structure

that by including the TLS channel ID in the *login ticket*, the server binds that ticket to the server-browser TLS channel. Because the *login ticket* is end-to-end encrypted to the user's personal device, a rogue middle party is unable to undetectably modify it. By using the TLS channel ID that the browser has placed in the *assertion request* in conjunction with the TLS channel ID from the *login ticket*, the user's phone is able to determine if a man-in-the-middle is present. Observe that the metadata provided by the browser is encrypted and authenticated by the *origin-protection-key* which the user's device extracts after decrypting the *login ticket*. When the *identity assertion* returns to the server, it can be sure that: 1) the identity assertion came over the same TLS channel as the user password (no phishing occurred), 2) there was no TLS man-in-the-middle between the browser on which the password was entered and the server, and 3) the user's phone was near the PC during authentication[2].

Enrollment

As we mentioned briefly earlier, the user's phone must be enrolled with the server prior to use during authentication. Specifically, the user's phone registers itself with the server by

[2]We discuss the reasoning for this after providing some implementation details.

telling the server its public key and identifying which user(s) it will vouch for. The user's personal device and the server also agree on a master encryption key during the enrollment process. The architecture of the enrollment protocol is fairly simple (occurring as a single HTTP POST request) and is discussed in detail in Section 4.3.

Enrollment need only be done once per website and phone. Once a user has enrolled his phone device with a server, he will not have to do this again.

Clearly, prior to enrolling into this system, users do not have the benefits of the system and are vulnerable to some the attacks against which this system protects. Namely, we assume there to be no TLS man-in-the-middle between the user's PC and the server during enrollment.

Practical Maintenance Operations

During the normal use of PhoneAuth, several maintenance operations will occur. We address each in turn.

Adding More Phones Users may want to have more than one phone. However, to make it easier for users to maintain consistency (the lack of which which may introduce user confusion) we suggest only allowing users to have one enrolled phone as their authentication device. We enforce this by overriding enrollment information every time the user registers a (new) phone.

Recovery / Replacing a Phone Users will want to replace their phone for a variety of reasons — upgrades, loss (or breakage), or just because. In our system this is easily accomplished if the user has at least one PC which has an active login session. The user will simply elect to "replace their authentication device" in the web UI. This will present a QR code which the user can scan with their new phone. The QR code includes session information from the PC's active login session which the phone can use to prove to the server that the user did have a valid session. As an alternative, users can elect to have a special SMS sent to their new phone (presumably the phone number has stayed constant), which will help them get through the replacement process.

In the case where the user does not have an active protected login session and has a new phone number, users will need to go through a thorough account recovery procedure. For example, a service provider may send an e-mail to a backup email address or ask questions about the content of the account (such as recent e-mails). There best recovery technique for a service provider largely depend on the type of service being offered. We therefore do not give concrete guidance on what the account recovery procedure should be.

Revocation Users may want to revoke their phone (in case it is stolen or they decide to withdraw from the protected login system). Similar to replacing a phone, this can be accomplished through the web interface at the server if the user has an active session. Otherwise, device revocation can potentially be a very dangerous action. In case of no active session, we recommend that service providers verify the user identity via a thorough account recovery procedure (see above).

4.3 Implementation

Having presented the overall architecture, a key question arises: why did we choose to implement identity assertion generation on a smart phone? A standard option might have been a Near Field Communication (NFC) smartcard or dedicated token, as used by other constructions. While offering good security properties, the use of dedicated tokens has usability problems — *e.g.*, requiring changes in user behavior (either to keep the token with the user or to place the token near the computer when authenticating); these user behavior changes violate our goal to keep the action of logging in invariant. An additional advantage of using a phone is that users *already* possess such a device, whereas otherwise they would have to obtain a special-purpose authentication device from somewhere.

Phones and Bluetooth Our system requires that the PC and phone communicate wirelessly, since a wired connection would have undesirable usability consequences. While they could clearly communicate through a trusted third party known to both (*i.e.*, a server in the cloud [1]), this approach introduces unacceptable latency and the need for cellular connectivity. Instead, we have elected to have the PC and phone communicate directly through

Bluetooth. Though other alternative ad-hoc wireless protocols exist (*e.g.*, wifi direct, NFC), they are not sufficiently ubiquitous or have other inherent limitations. Unlike NFC (which is for extremely close range communication, *i.e.*, "touching"), Bluetooth allows the user to keep the phone in their pocket during the authentication process — a huge usability benefit. Though the range of Bluetooth has been shown to be artificially extendible by attackers [123], this is not a security issue for our design unless the attackers are also able to mount a TLS man-in-the-middle attack on the PC-Server connection — in which case, such attackers are outside the scope of our threat model (see Section 4.2).

4.3.1 Key Challenges and Methods

While implementing this system, we encountered a number of interesting technical and design challenges.

Phone and PC Communication The central challenge with using Bluetooth in our environment is that we want to simultaneously support (1) Bluetooth communication between the user's phone and the user's browser without any user interaction with the phone and (2) have this work even when the user has never had contact with the computer / browser before. This is a challenge because, without prior interaction, the phone and the computer / browser will not be paired. To overcome these challenges, we modify both the browser and leverage a seldom used feature of the Bluetooth protocol.

In order for the PC and phone to contact one another over Bluetooth they need to learn one-another's Bluetooth MAC address. In most scenarios, this is usually done by putting one or both devices in discoverable mode, scanning for devices, then using a UI to pick the corresponding device from the menu. Since this process is highly interactive and time consuming (especially the scanning portion), we investigated ways of short circuiting the process. We leverage the fact that if one of the devices knows the MAC address of the other device, then the discovery phase can be bypassed and communication can immediately commence. Note that the phone and the PC are assumed to not have any prior association and therefore do not know each other's address.

We considered two bootstrapping mechanisms: 1) the phone would "be told" the PC's address and would initiate a connection with the PC or 2) the PC would “be told” the phone’s address and would initiate a connection with the phone. For the first mechanism, the server could send a message to the phone through the cloud. However requires a cellular connection (which may not be available) and introduces high latency thereby changing the user experience and violating our goals from Section 4.2. For the second mechanism, the PC can obtain the phone’s Bluetooth MAC address from the already existing (and lower latency) server connection[3].

Though the PC and Phone can make radio contact, there are still a number of challenges to overcome. Traditionally, before any Bluetooth communication takes place, the user must first “pair” the two devices. This usually involves showing the user some interface where he is asked to compare several numbers and usually press a button on one or both devices. This is both labor and time intensive from the user’s point of view. Instead, we utilize the ability of Bluetooth devices to communicate over unauthenticated RFCOMM connections. This technique allows us to create a “zero-touch” user experience by not forcing the user to interact with the mobile phone at all while authenticating on the PC. Recall from Section 4.2.3 that although the Bluetooth connection is unauthenticated at the RFCOMM level, the data is end-to-end authenticated and encrypted on the application level using the *origin-protection-key*.

Browser Support for Phone Communication Our architecture proposes that webpages should be able to request identity assertions from the user’s mobile phone. One way of achieving this goal is to create an API that would allow webpages to send arbitrary data to the user’s phone. At the extreme, this would amount to a Bluetooth API in JavaScript. This approach is unattractive for a variety of both security and usability reasons. For example, it might allow malicious sites to freely scan for and send arbitrary data to nearby Bluetooth devices. This may expose those devices to DOS attacks, make them even more vulnerable to known Bluetooth exploits, and allow attackers to potentially track users via their Bluetooth address. Instead, we chose an approach that exposes a much higher level

[3]This must be done carefully, lest the designer creates a Bluetooth address oracle. See Section 4.5 for more discussion of this pitfall.

API — thereby severely constraining the attackers' abilities. We describe this in detail below.

4.3.2 Implementation Details

Browser We extended the Chromium web browser to provide websites a new JavaScript API for fetching identity assertions. We modeled our approach after the BrowserID [2] proposal by using the `navigator.id` namespace. The full API consists of the function:

```
navigator.id.GetIdentityAssertion()
```

This API accepts three parameters: 1) a certauth id, 2) a login ticket, and 3) a JavaScript callback function that will be called when the identity assertion is ready.

If an identity assertion is not able to be fetched (either because the phone is not in range or the ticket is incorrect), the callback function may not be called — this is to help prevent malicious actions such as brute-forcing correct login tickets and tracking users by Bluetooth address.

Since regular Chromium extensions don't have the ability to interact with peripheral devices (*i.e.*, Bluetooth), we also wrote an additional NPAPI plugin that is embedded by the extension. The extension currently supports the Chromium browser on both Linux and Windows platforms. In total, the modification consisted of 3300 lines of C and 700 lines of JavaScript.

Pending work is ongoing to implement this functionality into the core Chromium browser code. Additionallyu, we are currently investigating together with the Firefox team whether our `GetIdentityAssertion` API and the `BrowserID` API can be combined into a single API.

Mobile Phone We modified the Android version of the open source Google Authenticator application [110] to provide identity assertions over unsecured RFCOMM. The application is able to provide identity assertion while the screen is off, and the application is in the background. The total changes required were 4000 lines of Java code.

Server We chose a service-oriented design for the server-side implementation. The central service exposes three RPCs: RegisterDevice, GenerateTickets, and VerifyTicket. The RegisterDevice RPC is exposed as a REST endpoint directly to users' phones. The other two RPCs are intended for login services. The idea is that a (separate) login service will call the GenerateTickets RPC after it performed a preliminary authentication of the user (using username and password), and will forward the login tickets returned by this RPC to the user's browser. Once the user's browser has obtained an identity assertion from the user's phone and has forwarded it to the login service, the login service will use the VerifyTicket RPC to check that the identity assertion matches the previously issued login ticket.

The basic signatures of the three RPCs is:

RegisterDevice Input parameters include an OAuth token identifying the user account for which the device is registered, a public key generated by the device, and the Bluetooth address of the device. This RPC returns the ticket master key.

GenerateTickets The following input parameters are included in the login tickets:

- The user id of the user for which the login service needs Login Tickets.
- The URL of the login service.
- The TLS channel ID (see Section 4.2.3) of the client that has contacted the login service. This is an optional parameter and only included if the client (browser) supports TLS-OBC.
- A boolean designating whether the user has explicitly indicated an intent to log in (such as typing a username and password), or not (such as during a "password-less" login that is triggered purely by the proximity of the phone to the browser). This boolean is embedded in the login ticket and allows the phone to present a consent screen on the phone if no previous user consent has been obtained by the login service for this login.
- A boolean indicating whether the login service supports TLS-OBC. This allows us to detect an attack in which a man-in-the-middle pretends to a TLS-OBC-capable browser (respectively login service) that the login service (respectively

browser) doesn't support TLS-OBC. This boolean will be compared by the phone to a similar boolean that the browser reports directly to the phone.

This RPC returns a login ticket for the indicated user's registered device. As noted earlier, a login ticket includes many of the input parameters, together with an expiration time and an origin protection key, and is encrypted and signed with keys derived from the ticket master key established at device enrollment time. Every login ticket is accompanied by an identifier that includes the Bluetooth address of the device possessing the ticket master key.

VerifyTicket This RPC's input parameter is an "identity assertion", which is simply a counter-signed login ticket. The service simply checks that the ticket is signed by a key that corresponds to the user for which the ticket was issued, and returns an appropriate status message to the caller (the login service).

The complete implementation of this service (not including a backend database for storing device registration information, unit tests, and the actual login service) consisted of 5500 lines of Java.

4.4 Evaluation

4.4.1 Comparative

We now evaluate our system using Bonneau *et al.*'s framework of 25 different "benefits" that authentication mechanisms should provide. We evaluate the two modes of using PhoneAuth — strict and opportunistic. Recall from Section 4.2.3 that in strict mode, the user can only successfully authenticate if an identity assertion is fetched from his phone. In opportunistic mode, however, identity assertions are fetched opportunistically and users achieve either "protected" or "unprotected" login, with the latter possibly resulting in user notifications or restricted account access. We also include the incumbent passwords and a popular 2-factor scheme as a baseline; we reproduce scores for passwords exactly as in Bonneau *et al.*'s original publication, but disagree slightly with the scores reported for Google 2-Step Verification (2SV). The results of our evaluation are shown in Table 4.1.

	Usability									Deployability						Security										
Scheme	Memorywise-Effortless	Scalable-for-Users	Nothing-to-Carry	Quasi-Nothing-to-Carry	Physically-Effortless	Easy-to-Learn	Easy-to-Use	Infrequent-Errors	Easy-Recovery-from-Loss	Accessible	Negligible-Cost-Per-User	Server-Compatible	Browser-Compatible	Mature	Non-Proprietary	Resilient-to-Physical-Observation	Resilient-to-Targeted-Impersonation	Resilient-to-Throttled-Guessing	Resilient-to-Unthrottled-Guessing	Resilient-to-Internal-Observation	Resilient-to-Leaks-from-Other-Verifiers	Resilient-to-Phishing	Resilient-to-Theft	No-Trusted-Third-Party	Requiring-Explicit-Consent	Unlinkable
Passwords			y	y		y	y	s	y	y	y	y	y	y	y		s						y	y	y	y
Google 2-Step Verification (2SV)				y		y	s	s	s	s			y	y			s	y			y	y	y	y	y	y
PhoneAuth – strict		s		y		y	y	s	y	y	s	s	s	s	y	y	y	y	y	s	y	y	y	y	y	s
PhoneAuth – opportunistic		s	s	y		y	y	s	y	y	y	s	y	y	y	s	s	s	s	s	s	s	y	y	y	s

Table 4.1: Comparison of PhoneAuth against passwords and Google 2-Step Verification using Bonneau *et al.*'s evaluation framework. A 'y' indicates that the benefit is provided, while 's' means the benefit is somewhat provided. For some scores, we disagree with the Bonneau scoring.

Usability In the usability arena, the strict and opportunistic modes are similar to passwords and 2SV in that they provide the *easy-to-learn* and *easy-to-use* benefits since neither mode requires the user to do anything beyond entering a password. We rated both strict and opportunistic modes as somewhat providing the *infrequent-errors* benefit since they will cause errors if the user forgets his password or if the PC-phone wireless connection does not work. The strict mode does not provide the *nothing-to-carry* benefit since users won't be able to authenticate without their personal device. On the other hand, opportunistic mode somewhat provides that benefit since users may get a lower privileged session without their personal device. Both PhoneAuth modes provide the Quasi-Nothing-to-Carry benefit, since the device that the user is required to carry is a device they carry with them already anyway. We indicated that both strict and opportunistic modes at least somewhat provided the *scalable-for-users* benefit since they reduce the risk of password reuse across sites.

Deployability Assessing the deployability benefits comes down to evaluating how much change would be required in current systems in order to get our proposed system adopted. We note that the opportunistic mode is fairly deployable since it can always fall back to simple password authentication. Strict mode provides less deployability benefits, but is not far behind. Since the system is not proprietary, the changes that would need to be done both on the browser and server are minimal. Similarly, the cost-per-user of these systems is minimal as well.

Security The security benefit arena is where our approach really shines over passwords and 2SV. While the Bonneau *et al.* study indicated that 2SV was resistant to phishing, unthrottled guessing, and somewhat resistant to physical observation, we do not believe this to be the case. Attackers can phish users for their 2SV codes and, in conjunction with a phished password, can compromise user accounts. The same is true under physical observation and unthrottled guessing.

In comparison, PhoneAuth in strict mode is able to provide all of the security benefits except for *unlinkable*, which we say it provides somewhat because even though the user will be exposing his or her Bluetooth MAC address to multiple verifiers, privacy conscious users can change their Bluetooth MAC address to not be globally unique. The opportunistic mode provides all of the security benefits of passwords, but is also able to somewhat provide the other security benefits by restricting users (or attackers) who don't provide an identity assertion to less privileged operations and notifying users of the less secure login.

Discussion Given this evaluation, we believe that PhoneAuth fares very well against the Bonneau *et al.*'s metric and compares favorably with the 35 authentication mechanisms investigated in the Bonneau *et al.* study.

4.4.2 Performance

Measuring the performance impact of our login scheme is a complex task. Issues range from the impact of the Bluetooth service on the phone battery life, to overhead introduced by the additional cryptographic functions (both for TLS-OBC and for the login ticket issuance, signature and verification), and finally the additional overhead introduced during login by communicating to the phone, additional round trips between browser and server, and so on. Below we discuss a number of these issues.

Overhead of Cryptography A key concern is the use of TLS-OBC — since every connection to the login service will incur the respective penalty. If the latency and overhead introduced by TLS-OBC is too great, then this will manifest itself in slow load times of the login page, for example. We refer the reader to a detailed discussion of the TLS-OBC performance in Chapter 3. In that chapter, we show that the overhead is negligible once the browser has generated a client certificate (and very small for certain key types even when a new client certificate needs to be generated).

The overhead of cryptography during the login process (generating the login ticket, checking and signing it, and checking the ticket signature) is dwarfed by the "human-scale" operations performed during login (typing a username and password), and by the additional round trips between browser and server. For example, a typical login ticket generation and verification took about 1 millisecond in our setup of 1000 test runs. We examined the timing of other cryptographic operations, but do not report on them as they incur a delay of approximately the same order, but have no end-user-impact (which is dominated by other latency; see below).

Overhead of Additional Round Trips During login, the browser makes an additional request to the server — to obtain the login ticket from the login service. The latency introduced by such a request is highly variable — from a few milliseconds for clients on a good network connection close to a datacenter where the login service is running, to a few seconds for mobile clients in rural areas far away from any datacenter. The *relative* overhead of a single additional round trip, however, is relatively low. Bringing up the login pages for Gmail, Facebook, and Hotmail, for example, involves 14, 11, and 14 HTTP requests as of the time of this writing (and this does not include submitting the password, getting redirected to the logged-in state, and so on — simply loading and displaying the login page).

Overhead of Involving the Phone During Login This is perhaps the most interesting type of overhead incurred: The browser has to establish a Bluetooth connection to the phone and obtain an identity assertion. As a baseline comparison, we measured how long it took a member of our team to log into a simple password-based login service (type username, password, and submit) — an average of 8.8 seconds. Repeating the same login while also obtaining an identity assertion increased the average time to 10.3 seconds. The additional 1.5 seconds are mostly spent establishing the Bluetooth connection, with processing time on the phone and penalty for the additional round trip being much less of an issue in comparison.

We noticed that the "long tail" of Bluetooth connection setup time, however, was considerably slower — sometimes taking up to 7 seconds. As a result, our test login service tries for as long as 7 seconds to connect to the phone before giving up and proceeding with a password-only "unprotected" login. Not surprisingly, when we tested login with the phone turned off (simulating a situation in which the phone wasn't available to protect the login), the average login time increased to 16.7 seconds — almost all of the additional time was spent waiting (in vain) for the Bluetooth connection to the phone to be established.

We envision techniques that may shorten the login time even more. For example, "lazy verification" of the second factor credentials (for opportunistic rather than strict logins) may work as follows. The user is allowed to login like normal if a second factor device is not found within 1 second, but behind the scenes the server continues to search for the second factor device for another 20 seconds. If the second factor device is found, the user session is upgraded and no notifications will be sent out.

This is still faster than a typical two-factor login, however. We measured an average login time of 24.5 seconds for a 2-factor login service that included typing a username and password, and copying a one-time code from a smart phone app to the login page.

Note that for a user that uses 2-factor authentication, and whose login service may perhaps accept both traditional one-time codes and the (considerably more secure) cryptographic assertions from the phone as a second factor, login actually *speeds up* dramatically with our system (from 24.5 seconds to 10.3 seconds), while at the same time reverting the login experience to a simple "username+password" form submission *and* improving security.

4.5 Discussion

Operational Requirements and Deployability As the careful reader has noticed, PhoneAuth has several operational requirements which must be met in order for the system to be deployed. First, our browser extension's functionality should be ported to be part of the actual browser. We have approached the Chromium browser team and have interacted with the Firefox team to make that happen. Second, it must be simple for developers to deploy this authentication scheme to their websites. Our service-oriented implementation of the server-side PhoneAuth functionality makes this easy, but a roll out of PhoneAuth to a non-trivial deployment is still in its planning phase. Third, the system must be tested and approved by users. We believe the main reason similar systems have failed is that none have been able to support opportunistic strong user authentication without modifications to the user experience — a feature which our system provides. We are planning on running field tests of the system in the near future. Finally, Bluetooth should be a ubiquitous technology on most phones and PCs. We found that the majority of new devices do indeed ship with Bluetooth [116]. Examining several major device manufacturers, we found that all Apple computers, almost all laptops (HP and Dell), and about half of Desktop PCs (HP and Dell) have integrated Bluetooth. Given these statistics, we believe the ubiquity of Bluetooth goal to be realistic.

Other Methods for Testing Phone/PC Colocation Instead of relying on a wireless channel between the phone and PC, an alternative approach for testing for proximity between phone and PC may be to query both for their location. Most phones can provide their location coordinates (for example through GPS or cell triangulation). Recently, browsers have begun to expose geolocation APIs as well [82]. However, without a GPS fix, phones may provide location data with too coarse of a granularity. More troublesome, however, is that the browser geolocation API (which is based on IP addresses) does not work from behind a VPN or on large managed networks (such as our university). These two issues make the location based approach impractical.

As yet another approach, it may be feasible to transfer identity assertions via NFC (by having users tap their phones on NFC readers) or by having users scan QR codes. Both of these approaches carry a non-negligible user experience impact. Users must take their phone out of their pocket, purse, or backpack, potentially unlock the screen, and potentially launch an app. We believe this usability impact is too severe and therefore do not consider these modes of operations.

Finally, some designs may be possible that leverage the cellular network. We have chosen not to use them because of occasional lack of cellular coverage and potentially high latency.

Avoiding a Bluetooth Address Oracle We briefly considered a very attractive design, but discarded it for privacy reasons because it inadvertently created a Bluetooth address oracle. Specifically, in the design, users could just type their username into the webpage login page and an identity assertion would be fetched from their phone without requiring users to enter a password. This, however, required that web sites expose an API that would return a Bluetooth address based on username. Even though this design presented nice usability benefits, we stayed clear of this approach.

4.6 Summary

In this chapter we introduced PhoneAuth, a new method for user authentication on the web. PhoneAuth enjoys the usability benefits of conventional passwords — users can, for example, approach an Internet kiosk, navigate to a web page of interest, and simply type their user name and password to log in. At the same time, PhoneAuth receives the benefits of conventional second-factor authentication systems and more. Specifically, PhoneAuth stores cryptographic credentials on the user's phone. If present when the user logs into a site, then the phone will attest to the user's identity via Bluetooth communications with the computer's browser; this happens even if the user has never interacted with that particular computer before. Since users may occasionally forget their phones, we further considered a layered approach to security whereby a web server can enact different policies depending on whether or not the user's phone is actually present.

We called the general layered approach "opportunistic identity assertions". Opportunistic identity assertions allow the server to treat logins differently based on how the user was authenticated — allowing the server to provide tiered access or restrict dangerous functionality (*e.g.*, mass e-mail deletion). Thus, while opportunistic identity assertions may not always be available to all users (*e.g.*, lack of Bluetooth support), there are still advantages in providing them. Similarly, an adversary who is able to make it appear that Alice's phone is "not there" simply degrades Alice's login and prevents access to dangerous functionality.

We implemented and evaluated PhoneAuth, and our assessment is that PhoneAuth is a viable solution for improving the security of authentication on the web today. We believe that PhoneAuth has the potential to significantly improve the security of user authentication without impacting the usability of the login experience.

Chapter 5

ALLOWED REFERRER LISTS: STRENGTHENING USER AUTHORIZATION

In the previous chapters, we presented two systems for making user authentication more resilient against strong attackers while striving to provide a good user experience. We now turn towards examining user authorization and present Allowed Referrer Lists (ARLs) as a lightweight mechanism for protecting against Cross-Site Request Forgery attacks. ARLs were originally published in a 2013 paper [32]. We begin by describing the motivation for and the overview of the work.

5.1 Motivation and Overview

Web application developers have relied on web cookies to not only provide simple state management across HTTP requests, but also to be bearers of authentication and authorization state. This programming paradigm, combined with the fact that web browsers send cookies by default with every HTTP request, has led to the proliferation of ambient authority (see Section 2.3.1 for more details), whereby HTTP requests can be automatically authenticated and authorized with the transport of a cookie. Other sources of ambient authority include state in HTTP authentication headers, client-side TLS certificates, and even IP addresses (which are used for authorization in some intranets or home networks). Such ambient authority, in turn, has led to the proliferation of Cross-Site Request Forgery (CSRF) attacks.

CSRF attacks occur when malicious web sites cause a user's web browser to make unsolicited (or forged) requests to a legitimate site on the user's behalf. Browsers act as confused deputies and attach any existing cookies (or other ambient authority state) to the forged request to the victim site. If the web application looks at the cookie or other state attached to an HTTP request as an indication of authorization, the application may be tricked into performing an unwanted action. For example, when a user visits *bad.com*, the displayed page may force the browser to make requests to *bank.com/transfer-funds* (*e.g.*, by including an image). When making the request to *bank.com*, the user's browser will attach any cookies it has stored for *bank.com*. If *bank.com* verifies the request only via the attached cookies, it may erroneously execute the attacker's bidding. Section 2.3.1 provides more background on these attacks.

CSRF attacks are a major concern for the web. In 2008, Zeller *et al.* [122] demonstrated how several prominent web sites were vulnerable to CSRF attacks that allowed an attacker to transfer money from bank accounts, harvest email addresses, violate user privacy, and compromise accounts. From January 1 to November 16 in 2012, 153 CSRF attacks have been reported, making 2012 one of the most CSRF-active years [87]. These vulnerabilities are not merely theoretical; they have a history of being actively exploited in the wild [9, 106].

CSRF defenses exist, and some may believe that defending against CSRF attacks is a solved problem. For example, tokenization is a well-known approach adopted in various forms in numerous web development frameworks. In a tokenization-based defense, a web server associates a secret token with HTTP requests that are allowed to cause side-effects on the backend. Assuming an attacker site cannot capture this token, the attacker cannot launch CSRF attacks. However, our in-depth analysis (Section 2.3.2) reveals that tokenization has a number of practical drawbacks, such as lack of protection for GET requests, possible token extraction by adversaries, and challenges dealing with third-party web development components. We show that other defenses are also limited: either too rigid (thereby blocking legitimate content) or too yielding (thereby allowing certain attacks to occur).

By studying drawbacks in existing approaches, we set out to build a new CSRF defense that is (1) developer-friendly, (2) backward compatible (not blocking legitimate content), and (3) has complete coverage (defending against all CSRF attack vectors). We propose a new mechanism called *Allowed Referrer Lists (ARLs)* that allows browsers to withhold sending ambient authority credentials to web sites wishing to be resilient against CSRF attacks. We let participating sites specify their authorization structure through a whitelist of referrer URLs for which browsers are allowed to attach authorization credentials to HTTP requests.

This approach takes advantage of the fact that browsers know the browsing context, while web developers understand the application-specific authorization semantics. By letting browsers carry out the enforcement and having web developers only specify the policies, we ease the enforcement burden on developers. By only having participating sites specify the policies and receive CSRF protection from browsers, we leave other web sites' behavior unchanged, thus providing backward compatibility.

We have implemented ARLs in the Firefox browser. To evaluate ARLs, we studied four open-source web applications for which source code was available and for which the CSRF attacks were reported via the public vulnerability e-mail list "full-disclosure". We analyzed each application and reproduced the reported CSRF attacks. We then developed ARLs for each application and showed that the attacks were no longer possible. We also compared the amount of effort needed to implement CSRF protection using ARLs versus a traditional tokenization patch, finding ARLs to be the easier solution and one that provides better coverage.

We also studied how ARLs could be deployed on three large, real-world sites: Gmail, Facebook, and Woot. We found that most features offered by these sites could be supported with ARLs with very few modifications. We also considered ARL compatibility for complicated real-world web constructs such as as nested iframes, multiple redirections, and federation identity protocols.

In summary, in this chapter we propose, implement, and evaluate a new browser mechanism for not sending ambient authorization state for participating sites based on their policies. We begin by presenting the ARL design in Section 5.2, then describe our browser implementation in Section 5.3. In Section 5.4, we evaluate ARLs against real CSRF attacks on open-source applications and discuss how ARLs would fare on real-world sites. We discuss privacy and limitations in Section 5.5 and conclude in Section 5.6.

5.2 Design

Learning from our analysis of pitfalls in existing defenses, our anti-CSRF design should be (1) **easy for developers** (*e.g.*, unlike checking of anti-CSRF tokens), (2) **transparent to users** (*e.g.*, unlike CsFire or NoScript which ask users to define or approve policies), (3) **backwards compatible** to not break legitimate sites that do not opt in to our defense (*e.g.*, unlike CsFire breaking OpenID), and most importantly, we should (4) **address the root cause** of CSRFs, namely ambient authority, to provide more comprehensive coverage against CSRF than existing solutions.

Recall from Section 2.3.1 that fundamentally, CSRFs result when browsers exercise ambient authority: (1) browsers automatically attach credentials to HTTP requests and (2) web application developers treat the presence of those credentials in a request as implicit authorization for some action. Keeping these root causes in mind, we make the following key observations.

- While splitting authentication and authorization is a classical best practice in computer security [70, 118], it is underutilized on the web. In our experience, most websites use a single token (such as a session cookie or the HTTP basic authentication header) for both authentication and authorization. Decoupling this concept on the web could have significant security benefits.

- The developers of site X are in the best position to determine which other sites are authorized to cause the user's browser to issue HTTP requests that perform state-modifying actions on X. However, server-side code on site X cannot reliably tell which other site caused the user's browser to issue an HTTP request.

- On the other hand, browsers know the full context in which requests are issued. Specifically, browsers know the full DOM layout and can infer whether a request was triggered by a user action, a nested iframe, or a redirection.

In light of these observations, we introduce a new browser mechanism called *Allowed Referrer Lists (ARLs)*, which restricts a browser's ability to send ambient authority credentials with HTTP requests. Sites wishing to be resilient against CSRF must opt in to use ARLs. With ARLs, participating sites specify which state (*e.g.*, specific cookies) serves authorization purposes. Sites also specify a whitelist of allowed referrer URLs; browsers are allowed to attach authorization state to HTTP requests made from these URLs only. This policy is transmitted to the user's browser in an HTTP header upon first visit to a site, before any authorization state is stored. For all subsequent requests, the user's browser attaches the authorization state only if the policy is satisfied.

This approach capitalizes on the fact that browsers know the browsing context, namely determining which web site principal issued an HTTP request, while web developers understand site-specific authorization semantics, namely whether a request-issuing web site should be authorized. By having developers only specify policies and letting browsers carry out enforcement, we ease the enforcement burden on developers. By having only participating sites specify policies to receive CSRF protection, we leave other sites' behavior unchanged, thus providing backward compatibility.

5.2.1 Identifying and Decoupling Authentication and Authorization

To use ARL, developers should identify and decouple the credentials they use for authenticating and authorizing users' requests.

First, developers must determine which credential is being used to identify users. Recall from Section 2.3.1 that developers use various methods to identify users: HTTP authentication, source IP, TLS client certificates, and cookies. In many cases we studied, applications use a single cookie to authenticate users.

Next, developers should create a credential for *authorizing* user requests. For example, developers may choose to set a new authorization cookie called *authz* on the user's browser. Any HTTP request not bearing the authorization credential must not be allowed to induce state-changing behavior (even if the request has the authentication credential). A request bearing *only* the authorization credential should be treated as an unauthenticated request.

Finally, developers need to define ARL policies (see below) to regulate how browsers send the authorization credentials.

Some sites can benefit from ARLs without needing to separate authentication and authorization credentials, whereas others will require this separation to properly work with ARLs. In Section 5.4, we will further discuss these two cases.

5.2.2 Defining ARL Policies

Next developers define an ARL policy, which will then be sent to and interpreted by users' web browsers. The policy specifies rules which govern when and how HTTP requests can be sent. To define an ARL policy, developers list authorization credentials, which may include specific cookie name(s), HTTP Authentication, or even the whole request (for source IP or TLS client cert authentication). If a credential is mentioned in an ARL policy, we say that the credential has been *arled*. By default, *arling* a credential prevents that credential from ever being attached to any HTTP request. Developers specify additional rules to relax this restriction in a least-privilege way. ARLs have two core directives:

- **allow-referrers**: Developers provide a whitelist of referrers that can issue authorized requests. A referrer is a URL with optional wildcards. Referrers can be as generic as *https://*.bank.com/** or as specific as *https://bank.com/accounts/modify.php*. Wildcards and paths allow web sites to be flexible in expressing their trust relationships. For example *bank.com* may be owned and run by the same entity as *broker.com*, but *bank.com* may only want to receive authorized requests from *https://broker.com/transfer/**.

- **referrer-frame-options**: Malicious sites could cause a protected credential to be sent by embedding content (*e.g.*, in an iframe) from a referrer specified in the ARL policy. We therefore allow developers to restrict framing of referrers. Similarly to the HTTP Header Frame Options RFC [101], we allow three framing options: DENY, SAMEORIGIN, or ALLOW-FROM. DENY states that the referrer must not be framed when issuing an authorized request. SAMEORIGIN allows the referrer to make an authorized request while being framed by "itself" (*i.e.*, by a URL matching the ARL whitelist entry for that referrer) or by the target of the request. The ALLOW-FROM option takes additional values in the form of domain URLs such as *https://broker.com/*. We allow framing depth of at most one embedding to prevent attackers from mounting attacks on the embedder. If this directive is omitted, the default value is DENY.

```
arl {
  apply-to-cookie = authz,
  allow-referrers = https://*.bank.com/*
                    https://broker.com/finance/*,
  referrer-frame-options = SAMEORIGIN
}
```

Figure 5.1: **ARL policy example.** With this policy, bank.com forbids the browser from sending the `authz` cookie, except when the request was generated by *https://*.bank.com/** or *https://broker.com/finance/**, and only if the requesting page was framed by a page of the same origin.

Using these two directives, developers can generate simple yet powerful policies. Figure 5.1 shows a policy that may be used by *bank.com*. Here, the cookie `authz` is arled and *https://*.bank.com/** and *https://broker.com/finance/** are listed as the only referrers. This means that the browser will only attach the `authz` cookie to HTTP requests generated by an element from *https://*.bank.com/** or *https://broker.com/finance/**. Note that the *allow-referrers* directive specifies not only the domain, but also the scheme (HTTPS in this case) from which the requesting element must have been loaded. This differs from the Secure attribute of a cookie, which only specifies how to send the cookie itself. By including the HTTPS scheme, a web application developer specifies that protected credentials are never sent by an element not loaded over TLS. This control is not possible with any of the techniques we studied. This policy also states that the `authz` cookie may be attached only if the referrer was either not framed or framed only by a page from SAMEORIGIN (either target *bank.com* or referrer *broker.com/finance*). Note that unlike origin-based framing rules in the X-Frame-Options HTTP header, our rules will check for the full *broker.com/finance* path, *e.g.*, to avoid potentially malicious framing from *broker.com/forum* that *bank.com* did not intend.

```
arl {
  apply-to-http-auth = true,
  allow-referrers = https://*.bank.com/*
                    https://broker.com/finance/*,
  referrer-frame-options = SAMEORIGIN
}
```

Figure 5.2: **Restricting HTTP Authentication.** With this policy, bank.com forbids the browser from sending the HTTP Authentication header, except when the request was generated by *https://*.bank.com/** or *https://broker.com/finance/**, and only if the requesting page was framed by a page of the same origin.

As another example, Figure 5.2 shows how an ARL policy can be applied to HTTP basic authentication credentials instead of cookies (though we suggest that developers use cookies for authorization). Finally, Figure 5.3 shows how ARLs can be used to disallow any requests to a particular destination unless the request is being made by a particular referrer (specified via the **apply-to-requests-to** directive). This directive can protect sites which rely on source IP or TLS client certificates for authorization.

It is important to reiterate that ARLs should be applied to authorization, *not* authentication credentials. That is, from a security point of view (dismissing privacy concerns in this discussion), it is always acceptable for a web site to know from which user's browser an HTTP request came. However, it is not always acceptable for sites to take actions based on those requests. Web sites that do not have separate authentication and authorization credentials may not be able to fully utilize ARLs.

Application: Defeating Login and Logout CSRFs Barth, Jackson, and Mitchell described a Login CSRF attack whereby an attacker "forges a login request to an honest site using the attacker's user name and password at that site" [13]. A successful attack causes the user to be logged into the site with the attacker's credentials, allowing the attacker to

```
arl {
  apply-to-requests-to = https://accounts.bank.com/modify,
  allow-referrers = https://accounts.bank.com/*,
  referrer-frame-options = DENY
}
```

Figure 5.3: **Disallowing requests.** Here, *bank.com* forbids *any* requests (with or without credentials) to *https://accounts.bank.com/modify*, except when the request was generated by *https://accounts.bank.com/**, and only if the requesting page was not framed.

"snoop" on the user.

A Logout CSRF attack is similar in that it allows attackers to disrupt the user's session on legitimate sites. Many sites implement logout by having users visit a URL (*e.g.*, *site.com/Logout*). For many sites, this URL is vulnerable to a CSRF attack: malicious sites can embed an iframe pointing to a site's logout URL and cause any visitor of the malicious site to be logged out of the legitimate site. Google, for example, is vulnerable to this attack.

ARLs can be used to protect web applications against both login and logout CSRF attacks. To protect against login CSRF, a web site may set an arled dummy authorization credential when displaying the login form. Then, the site should verify that the dummy authorization credential is returned along with the user's other login credentials before setting the real authorization and authentication credentials. Similarly, logout CSRF can be stopped by simply checking that the (real) arled authorization credential is present before signing the user out.

5.2.3 Enforcing ARL Policies

Once the browser obtains an ARL policy for a site, the browser must examine each outgoing HTTP request's context to check whether any of a site's current ARL policies apply, and if so, whether or not to attach arled credentials to the request.

Referrers To enforce the "allow-referrers" directive, we leverage the fact that browsers already determine each request's referrer. Broadly speaking, a referrer is the URL of the item which led to the generation of a request. For example, if a user clicks on a link, the referrer of the generated request is the URL of the page on which the link was shown. The referrer of an image request is the URL of the page in which that image was embedded.

Browsers also handle complex cases of referrer determination. For the redirect chain *a.com* ⇒ *b.com* ⇒ *c.com*, modern browsers (IE, Chrome, Firefox, and Safari) will choose *a.com* as the referrer if the redirection from *b.com* ⇒ *c.com* was made via the HTTP `Location` header and will choose *b.com* as the referrer if the redirection from *b.com* ⇒ *c.com* was made by assigning *window.location* in JavaScript. This is not arbitrary. If the redirect is caused by the HTTP header, then none of the subsequent page content is interpreted by the browser. However, a JavaScript page redirect can occur at any point in time after an arbitrary amount of interaction with the user. Further discussion of this issue is beyond the scope of this chapter, but this issue shows that browsers already take great care to identify the source of every request so that they can enforce the same-origin policy — the foundation of many web security properties. We therefore use the default definition of referrer. If a credential is arled, then the request's referrer must match one of the "allowed-referrers". Otherwise, the credential will not be sent with the request.

Frame-options To enforce the "referrer-frame-options" directive, we are helped by the fact that browsers maintain the page embedding hierarchy (*e.g.*, to enforce the X-FRAME-OPTIONS header). Browsers record complicated cases with nested and dynamically created iframes. We simply consult this internal hierarchy when enforcing "referrer-frame-options".

If a credential (*e.g.*, cookie) is arled, then the embedding hierarchy of the referrer must match the "referrer-frame-options" policy as defined above. Our mechanism extends frame-checking to also consider popup windows to prevent attackers from causing CSRFs by opening victim pages in popups.

Mixed Content Sites Some web sites mix HTTP and HTTPS content when presenting web pages to users. For example, a site may choose to serve images over HTTP to increase performance, while all JavaScript and HTML are served over HTTPS. This practice may introduce certain security vulnerabilities. For example, since network attackers can manipulate content sent over HTTP, they can modify "secure" cookies. That is, although cookies bearing the "secure" attribute will only ever be *sent* over HTTPS, they can be *set* or overwritten via a `Set-cookie` header over HTTP [23].

To avoid these vulnerabilities, we introduce two rules for browsers implementing ARLs:

- If an ARL policy is received over HTTP, it may only overwrite an old policy if the old policy was also received over HTTP.

- If an ARL policy is received over HTTPS, it may overwrite any old policy.

These rules prevent ARL hijacking by network attackers who are able to insert an ARL header into an HTTP response.

Requests With no Referrer In several situations, a request may completely lack a referrer. Examples of such events are when users type URLs into the browser location bar, click on a bookmark, or follow a link from an e-mail. In these cases, any arled credential should not be sent, while all the other state should be sent. Note that this means that web sites will not initiate any state changing behavior as a result of this request, but can still show personalized content to the user (since the authentication credentials were sent).

5.3 Implementation

The implementation and deployment of ARLs involves several steps. First, web applications need to be modified to implement ARL policies. This includes designing a policy, potentially separating authentication and authorization credentials, and then inserting the policy into all relevant HTTP responses. Second, the user's browser needs to be modified to understand, store, and enforce ARL policies for arled credentials.

Modifications to web applications are specific to each application's own logic and framework. We explore how this is done with real-world applications in Section 5.4.1. In this section, we focus on options for delivering policies and modifications that browsers need to support ARLs.

5.3.1 Policy Delivery

The simplest delivery option, and one used in our implementation, is to piggyback ARL policies onto existing cookie definitions. Today, the Set-Cookie header allows cookies to specify constraints through attributes such as HttpOnly, Secure, Domain, and Path. We added a new "arl" attribute (bearing an ARL policy) to cookies, allowing each cookie to specify its own ARL policy. Unfortunately, this delivery method makes it difficult to address HTTP authentication and IP-based ambient authority.

An alternate ARL policy delivery mechanism is to integrate ARLs as a new directive for Content Security Policy (CSP) [112], which is specified through its own HTTP header. CSP already specifies a number of rules regulating content interaction on web applications, primarily for mitigating XSS and data injection [112]. Although we did not yet implement this approach, we recommend it over using cookie attributes due to added flexibility, expressiveness, and usability. For example, if a web application used multiple authorization tokens (*e.g.*, combining HTTP Authentication headers with cookies or using multiple authorization cookies), a developer would have to specify multiple (potentially duplicate) policies with the first approach but not with a centralized, HTTP-header-based policy. As well, HTTP headers can transmit longer ARL policies than cookies, which are typically limited to 4 kilobytes (though we don't expect policies to ever grow that large) Finally, having a single policy location that pertains to a variety of authorization tokens is cleaner and more readable than having multiple policies, each attached to a different authorization token.

5.3.2 Browser Modification

To validate the ARL design, we implemented it in the Mozilla Firefox 3.6 browser. Although the 3.6 version of Firefox is slightly dated, it was the most stable version when this project began in 2011. We believe that our modifications and implications of our implementation generalize to more recent versions of Firefox and other browsers.

Rather than create a browser extension or plug-in (as done by CsFire or NoScript ABE), we decided to directly modify Firefox. First, we wanted to have direct access to internal representations of cookies, referrer information and frame embedding hierarchy. Second, we wanted to validate the feasibility of adding native browser support for ARLs.

For our proof-of-concept, we used attribute-based policy delivery described above. We modified Firefox in three key aspects:

- **Parsing**. The first step towards modifying Firefox to support ARLs is to enable the browser to parse ARL policies out of HTTP replies. Since our policies were being sent as a separate cookie attribute, we modified the cookie parsing code for HTTP responses to accept a new cookie attribute "arl" and parse the contents into an ARL policy.

- **Storage.** Once the contents of the "arl" attribute are parsed and converted into policies, they need to be stored and linked to entities to which they apply. To store and retrieve all of the parsed ARL policies, we modified Firefox's cookie manager, cookie database (sqlite), and all of the related structures and methods to allow setting and retrieving a new "ARLPolicy" data structure for each cookie.

- **Enforcement.** Finally, we enforce ARL policies by adding code that checks each cookie's policy before attaching it to an HTTP request. Firefox already considers many factors before attaching cookies to requests, such as verifying that requests adhere to the same-origin policy or that the request is going over TLS (if the cookie's *Secure* attribute is set). We appended our ARL enforcement logic (described in Section 5.2.3) to this code.

Browsers are complex, multi-layered pieces of software, and unsurprisingly, we encountered various engineering challenges during our implementation. A key difficulty was discovering how all the different layers, services, and protocols interacted with each other. In total, our modifications consisted of approximately 700 lines of C++ code spread across 13 files.

5.4 Evaluation

We conducted experiments to answer several questions about the effectiveness and feasibility of ARLs, including: (1) how well do ARLs guard against CSRF attacks, (2) how difficult is

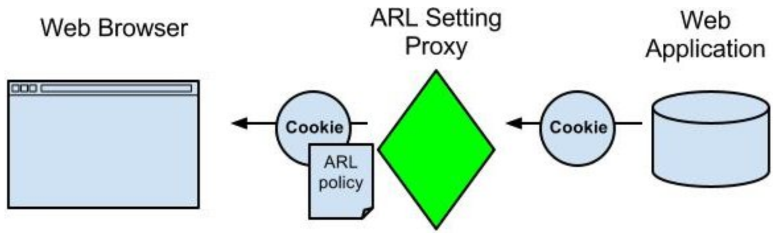


Figure 5.4: **ARL Proxy rewrites HTTP replies to contain ARL policies.**

it to write ARL policies for real sites, (3) how much developer effort is required to deploy ARLs, (4) does deploying ARLs break existing web applications, and (5) do ARLs have a performance impact?

To answer these questions, we first studied four open-source web applications with known CSRF vulnerabilities, implemented ARLs in the applications' code bases, and analyzed how well ARLs protected them against CSRF attacks. Second, we used the Fiddler SDK [73] to develop an ARL Setting Proxy that allowed us to experiment with ARL policies on arbitrary web sites without modifying their code. This proxy, implemented in 120 lines of JavaScript and illustrated in Figure 5.4, allowed us to study how ARLs interact with large, complex, on-line web applications such as Google. We also evaluated ARL performance using browser benchmarks.

We elaborate briefly on our ARL Setting Proxy here. There are many ways in which it may have been possible to test our browser implementation of ARLs. One method may be to modify and test real web applications (we write about doing this below), but it may not always be feasible — for example, if the source code of the website is not available. We, however, wanted to be able to deploy and test ARLs on any website — even if we did not have the website source code. Towards this goal, we developed an ARL Setting Proxy. This proxy intercepted regular HTTP traffic and rewrote HTTP replies from specific origins to contain ARL policies. This allowed us to quickly field test ARLs on legacy web applications or ones we were not able to modify. We performed this implemented with approximately 120 lines of JavaScript using the Fiddler Web Debugging Proxy Engine [73]. Figure 5.4 illustrates how the ARL setting proxy works.

Application	**Version**	**# of source files**	**Lines of Code**	**Type of Application**
Selectapix	1.4.1	39	6k	Image gallery
UseBB	1.011	83	21k	Forum
PHPDug	2.0.0	133	25k	URL/link sharing app (similar to digg.com)
PoMMo	PR16.1	234	32k	Mailing list manager

Table 5.1: Summary of web applications we studied. All applications were written in PHP and used a MySQL backend database.

5.4.1 Studying Applications with Source

We monitored the public security mailing list "full disclosure" during the summer of 2011 for CSRF reports. Using those reports, we chose four projects for which source code was available and for which the CSRF attacks were reproducible. The web applications we studied are summarized in Table 5.1.

For each application, we first performed an in-depth analysis to understand the intended authorization structure and then developed ARL policies to enforce that structure. We confirmed that without modifications, the applications were indeed vulnerable to CSRF attacks. Next, we deployed the applications with ARL policies and tested whether the attacks were now thwarted. Finally, we tested each application to check that ARL deployment did not inhibit any functionality. We report on an in-depth case study of one of the applications below. We found that ARL polices for the other three applications had a similar level of complexity, and that ARLs were effective at protecting these applications against CSRF attacks.

Case Study: UseBB

UseBB is an open-source web application for lightweight forum management. The project's web site advertises that rather than providing as many features as possible, UseBB strives to provide an easy and usable forum with only the most necessary features. UseBB is written in PHP with a MySQL database backend.

Initial vulnerability report and analysis In the summer of 2011, a submission to the security mailing list *full-disclosure* reported "Multiple CSRF (Cross-Site Request Forgery) in UseBB". This report was for version 1.011 of UseBB, which we downloaded and analyzed.

This version of UseBB consisted of 25k lines of code spread over 83 source files. After studying UseBB, we understood that the application's state management worked as follows: when a user logs into the application, UseBB sets a single cookie, called usebb_sid, to authenticate further HTTP requests as coming from that user.

Building a corpus of CSRF attacks The bug report to *full-disclosure* mentioned multiple vulnerabilities, but only identified one page as having a vulnerability. After studying the code, we discovered that UseBB had no CSRF protection on any of its pages. This resulted in any request bearing the *usebb_sid* cookie to be authenticated and authorized to perform arbitrary actions. For example, by exploiting this vulnerability, attackers may have been able to perform a variety of attacks including changing user e-mail addresses, name, or adding illegitimate administrator accounts. We implemented these attacks and formed a corpus of CSRF attacks.

Developing an ARL Policy Next, we developed an ARL policy and analyzed its effectiveness. First, so that we could study an ARL policy in action, we deployed an instance of UseBB on an internal network at the URL: `http://usebb.com`. Next, we verified that all CSRF attacks from our corpus worked as expected. Then, we developed an ARL policy and deployed it via the ARL Setting Proxy. The policy we developed is:

```
arl {
  apply-to-cookie = usebb_sid,
  allow-referrers = http://usebb.com/,
  referrer-frame-options = SAMEORIGIN
}
```

Next, we tested UseBB (now protected by ARLs) against our corpus of CSRF attacks and found the attacks to be no longer functional. Our ARL policy fully protected this deployed application instance against any CSRF attacks that leverage the *usebb_id* cookie.

Deploying an ARL Policy in UseBB The next step was to deploy the ARL policy within the UseBB application itself (rather than via the proxy). We explored several ways in which this may be done. For example, one approach we used to deploy ARLs in UseBB was to add one line to the Apache server configuration file:

```
Header add X-ARL-Policy "arl{ \
apply-to-cookie = usebb_sid, allow-referrers = self, \
referrer-frame-options = SAMEORIGIN }"
```

To use this method, the web application developer must have access to the global Apache configuration file, which may be unavailable on some shared hosting providers. In that case, the developer could also deploy ARLs by modifying the *.htaccess* file (*i.e.*, the local web server configuration file). We verified that this deployment strategy worked as well. The additions to *.htaccess* were:

```
<IfModule mod_headers.c>
 Header set X-ARL-POLICY "arl{ \
 apply-to-cookie = usebb_sid, allow-referrers = self, \
 referrer-frame-options = SAMEORIGIN }"
</IfModule>
```

A similar modification is possible for local configuration files for IIS web servers. We verified that these additions were possible, but did not deploy UseBB via IIS. There may, however, be cases where developers are not allowed to create or modify even local web server configuration files. In such cases, the developer would have to modify the application source directly to implement ARLs. Since UseBB is written in PHP, we accomplished this by adding the following line of code to files which produce HTML code:

```
header('X-ARL-Policy: arl{ \
    apply-to-cookie = usebb_sid, allow-referrers = self, \
    referrer-frame-options = SAMEORIGIN }');
```

We implemented all of the aforementioned approaches and verified that each of them secured UseBB against our corpus of CSRF attacks.

Comparing ARLs to Traditional Patch The official repository of UseBB was later updated to fix the CSRF vulnerabilities in version 1.0.12 of the code. The developers protected UseBB against CSRFs by writing a custom CSRF token framework. By manually inspecting changes introduced in version 1.0.12, we counted approximately 190 line changes that were related to the new CSRF defense. We tested version 1.0.12 against our corpus of CSRF attacks and found that it did prevent them. We believe our solution to be better than the traditional patch in several ways. First, as we saw, implementing ARLs requires many fewer code modifications. Second, while ARLs would protect against any new CSRF attacks that leverage the *usebb_id* cookie, the patch would not — additional code would have to be written for any new pages added to UseBB.

Backwards Compatibility Though ARLs clearly protected UseBB against CSRF attacks, we wanted to investigate whether the deployment of ARLs impeded any UseBB functionality. Since the UseBB source code did not include any unit (or other type of) tests, we performed all functionality testing manually. Even though UseBB did not have separate authentication and authorization credentials, we found that all existing legitimate functionality was maintained.

5.4.2 Studying Large Sites without Source

Beyond studying how ARLs behave in open-source software, we also investigated the feasibility of ARLs in real-world, proprietary web sites. The questions we most wanted to answer were *how complex would ARL policies be for such sites?* and *what kind of modifications would real sites make in order to adopt ARLs?*

Studying real websites is inherently difficult. Modern web applications, such as Google's Gmail, are incredibly complex. The server source code is proprietary, and the code that is shipped to the browser is often obfuscated or has been run through an optimizer that makes the code difficult to read. Furthermore, to optimize user experience, such sites set dozens of cookies on the client, some of which represent preferences, while others are responsible for authenticating the user's requests.

We chose to study three web applications which we believe to be quite complex and which, in our opinion, serve as good representatives of state-of-the-art web applications: Gmail, Facebook, and Woot (a flash deals e-commerce site owned by Amazon.com). For each application, we first identified the "important" application cookies. Next, we observed how those cookies were sent during normal operation. Finally, we created ARL policies for those cookies, deployed them using our ARL Setting Proxy, and examined whether normal functionality was maintained.

Selecting Important Cookies Modern, large web applications use a large number of cookies. For example, Gmail and Facebook set around 40 cookies, while Woot sets about 20[1]. Many of these cookies have to do with preferences, screen resolution, locale, and other application state. Some of these cookies, however, deal with authentication and (potentially) authorization — these are the cookies that need to be protected by ARLs.

The majority of cookies have cryptic names, making it difficult to infer their intended function. We identified authentication cookies by experimentally removing cookies set in the browser until the web site refused to respond to requests or performed a signout. This cookie elimination process was done through manual analysis by individually removing each cookie and testing the result.

Using this strategy, we were able to narrow the set of all cookies down to just a few important cookies for each application. For Gmail, we found the important cookies to be `LSID`, `SID`, and `GX`; for Facebook the important cookies were `xs` and `c_user`; for Woot, the important cookies were `authentication` and `verification`.

[1]The number of cookies varied based on user actions.

Developing ARL Policies Having selected the important cookies for each website, we next needed to determine what the legitimate referrers were for sending these authentication cookies. We accomplished this by performing normal actions (such as sending and checking e-mail on Gmail, viewing photos and friends on Facebook, and browsing items on Woot) and observing the network traffic through the Fiddler web proxy.

Using these traces, we then developed an ARL policy for each site. The policies were less complex than we had assumed they would be. For example, the policy for Gmail was:

```
arl {
  apply-to-cookie = SID LSID GX,
  allow-referrers = https://accounts.google.com
                    https://mail.google.com,
  referrer-frame-options = DENY
}
```

Policies for Facebook and Woot were of similar complexity; that is, they mentioned the important cookies and only a handful of referrers. Furthermore, we found that these policies did not inhibit regular in-app functionality.

Splitting Authentication and Authorization

While the simple ARL policies above can support many interactions with Google, Facebook, and Woot, we discovered some desirable actions that these policies cannot support with the sites' existing cookie structure. For example, these policies are not compatible with the use of Google as a federated login provider or the use of Facebook's Like buttons on other sites[2].

Recall from Section 5.2.1 that web sites should clearly separate authentication credentials from authorization credentials. After doing so, it is only necessary to arl the authorization credential. The authentication credential can be sent about as before. We found that the limitations above are all due to the fact that none of our applications separate authentication

[2]This was an artifact of Facebook's implementation; ARLs supported Google's +1 button.

and authorization in their cookies, and making this modification re-enables the unsupported functionality.

Embedded Widgets Facebook's Like Button and Google's +1 Button are just two examples of a large class of "embedded widgets" that allow users to like, pin, tweet, and otherwise share a web page with their social graph. A web page that wants to enable a specific widget on its page includes HTML or JavaScript, usually provided by the social network, which renders an iframe sourcing the specific social network and displays the social button. The iframe's content is loaded from the social network; this has two key features. First, it prevents the host page from programmatically clicking on the widget. Second, it gives the widget access to the user's cookies from that social network, so that when the user clicks on the widget, the user's state in the social network can be properly affected.

When a user clicks on such a widget, the browser infers the referrer of the consequent request to be the social network and not the host page. This is because the content inside the iframe has already been loaded from the social network. Consequently, one would expect ARLs to work out of the box with embedded widgets. Indeed, this is the case with Google's +1 button. However, ARLs do not currently work with Facebook's Like button because when sending the HTTP request for the iframe's initial contents, the browser determines that the referrer is the host page, preventing any arled cookies from being sent. If Facebook used the authentication/authorization splitting paradigm as above, then only the authorization cookie would not be sent on the initial request, and the iframe could still be loaded with authenticated content (*e.g.*, the number of friends that have liked the host page), allowing the like button to work.

Exploring Federated Login Federated identity management is the mechanism whereby web applications can rely on other web sites, such as Facebook Connect or OpenID, to authenticate users. One of our selected test apps, Woot, supports federated login from Google. That is, a user can click on a button on the Woot login page, which will redirect the user to Google, which will verify the user's authentication credentials and log them into Woot via another redirect. Unfortunately, since Woot did not separate authentication and authorization credentials, the only cookie we can arl is the single authentication cookie, which would then be stripped from redirection requests between the two parties. By splitting authentication and authorization credentials into two cookies (as above) and only arling the authorization credentials, federated login on Woot could be supported.

Limitations of Experiments Sites like the ones we studied interact with many other sites. These relationships are complicated and, sometimes, ephemeral. For example, a user who has logged into Gmail will also be automatically logged into YouTube. We did not explore these types of multi-domain, automatic single-sign-on environments due to the guesswork needed to identify all partnering relationships and correct functionality and, given an absence of access to the code, an inability to know whether the analysis is complete. However, we believe that web site owners should be able to generate appropriate ARL policies given known business relationships.

5.4.3 Browser Performance

To make sure ARLs are feasible for browsers to implement, we checked performance overhead of ARLs. More specifically, we set up a 4K benchmarking page that sets a cookie and, upon loading, sends the cookie back to the server using AJAX. We measured the latency of this HTTP response-request roundtrip with unmodified Firefox browser and with ARL-enabled Firefox and an ARL policy for the cookie. We evaluated two ARL policies: a 1-referrer policy, and a 30-referrer policy where only the last referrer was valid. We averaged our results over 100 runs. Our client ran on a Windows 7 laptop with 2.20GHz CPU and 2GB RAM, and our server ran on a Macbook Pro laptop with a 2.66GHz CPU with 8GB RAM, connected with a 100MBps network.

We found that the latency difference between unmodified and ARL-enabled browsers was negligible for requests with the small (single referrer) ARL policy. For the longer 30-referrer ARL policy, the median latency difference increased to 3ms (2%), which is still very acceptable. We expect most sites to have shorter ARL policies, and we note that our implementation of ARL parsing and matching was not optimized for performance.

5.5 Discussion

Privacy Adversaries may try to learn the referrer of a request by creating many cookies with different ARL policies and then seeing which ones are sent back. However, ARLs introduce no additional privacy leaks compared to the `Referer` header, which is already freely sent by browsers. Some organizations may need to conceal the referrer to prevent revealing secret internal URLs. To do this, they install proxies that remove the `Referer` header [67]. To prevent any additional referrer leaks from ARLs, these proxies (or ARL-enabled browsers) can strip all cookies and HTTP authentication headers from requests generated by following a link from an intranet site to an internet site. This should not negatively impact any functionality, since links from the intranet to outside sites should not be state-modifying.

Limitations ARLs help protect against many types of CSRF attacks, but they are not a panacea. For example, ARLs are unable to protect against "same origin and path" attacks. If a page accepts legitimate requests from itself and somehow an attacker is able to forge attacks from that page, then ARLs may be ineffective.

Another limitation involves deployment. Until ARLs are supported in most browsers, web sites must use older CSRF defenses alongside ARLs. This is also true for any browser-based security feature. For example, many sites still have to use JavaScript-based frame-busting because some older browsers still do not support the X-FRAME-OPTIONS HTTP header.

5.6 Summary

CSRF attacks are still a large threat on the web; 2012 was one of the most CSRF-active years on record [87]. In this chapter, we give a background of CSRF attacks, highlighting their fundamental cause — ambient authority. Next, we study existing CSRF defenses and show how they stop short of a complete solution. We then present ARLs, a browser/server solution that removes ambient authority for participating web sites that want to be resilient to CSRF attacks. We implement our design in Firefox and evaluate it against a variety of real-world CSRF attacks on real web sites. We believe that ARLs are a robust, efficient, and fundamentally correct way to mitigate CSRF attacks.

Chapter 6
CONCLUSION

The modern computing ecosystem presents a number of challenges for secure, private, and deployable user authentication and authorization on the web. While many techniques have been proposed to strengthen user authentication and authorization, in practice the deployed technologies have not kept up with existing attacks. In fact, user authentication and authorization have, in our opinion, become some of the weakest links in the security chain.

This dissertation attempted to provide insight and tools for catalyzing progress in secure user authentication and authorization. The dissertation first considered the challenges that new authentication and authorization technologies face. Next, the dissertation examined previously proposed approaches and offered analysis on why those technologies may have failed to see mass adoption and deployment. Finally, this dissertation presented the design, implementation, and evaluation of three systems each of which addresses an aspect of secure user authentication and authorization on the web. First, we presented *Origin Bound Certificates*, which provided a strong cryptographic foundation for binding credentials and tokens to TLS channels. Second, we described *PhoneAuth*, a system for providing usable and strong second-factor authentication through a user's mobile phone. Finally, we offered *Allowed Referrer Lists* as a lightweight server-side mechanism for strengthening authorization by preventing CSRF attacks.

Throughout the presented work, a major goal was to design systems that are practical — systems that not only increase the security and privacy of user authentication and authorization, but are also deployable. In support of this goal, this dissertation centered around two common themes: First, we focused on the usability implications of a technology — both for the developer and the user. Second, we worked closely with major technology firms to understand their constraints and operating conditions. As a result, our systems appear to be in the initial stages of being adopted by industry. For example, *Origin Bound Certificates* have been deployed in the Chromium browser and are being implemented in the Mozilla Firefox browser. As of this writing, *PhoneAuth* is also in the initial stages of being considered for deployment by industry. We are in the process of contacting the standards organization regarding the adoption of *Allowed Referrer Lists*. Our work and experience demonstrates that through careful consideration of stakeholder values, it is possible to build usable, secure, and deployable systems for stronger user authentication and authorization on the web.

BIBLIOGRAPHY

[1] Android cloud to device messaging framework, 2012. https://developers.google.com/android/c2dm/.

[2] BrowserID – quick setup, 2012. https://developer.mozilla.org/en/BrowserID/Quick_Setup.

[3] 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.

[4] Adobe. Cross-domain policy file specification, 2013. http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

[5] Aircrack. Aircrack-ng homepage, 2013. http://www.aircrack-ng.org/doku.php.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. http://www.isg.rhul.ac.uk/tls/TLStiming.pdf, 2013.

[7] Robert Auger. The cross-site request forgery (CSRF/XSRF) FAQ, 2010. http://www.cgisecurity.com/csrf-faq.html.

[8] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[9] Mark Baldwin. OpenX CSRF vulnerability being actively exploited, 2012. http://www.infosecstuff.com/openx-csrf-vulnerability-being-actively-exploited/.

[10] D. Balfanz. TLS origin-bound certificates. http://tools.ietf.org/html/draft-balfanz-tls-obc-01, Nov 2011.

[11] 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.

[12] Adam Barth. The web origin concept, 2011. `http://tools.ietf.org/html/draft-abarth-origin`.

[13] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[14] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0, 1996. `http://www.ietf.org/rfc/rfc1945.txt`.

[15] K. Bicakci, N.B. Atalay, M. Yuceel, H. Gurbaslar, and B. Erdeniz. Towards usable solutions to graphical password hotspot problem. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 318–323, 2009.

[16] Robert Biddle, Sonia Chiasson, and P.C. Van Oorschot. Graphical Passwords: Learning from the first twelve years. *ACM Comput. Surv.*, 44(4):19:1–19:41, September 2012.

[17] 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.

[18] 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.

[19] blowdart. AntiCSRF, 2008. `http://anticsrf.codeplex.com/`.

[20] Hristo Bojinov, Daniel Sanchez, Paul Reber, Dan Boneh, and Patrick Lincoln. Neuroscience Meets Cryptography: Designing crypto primitives secure against rubber hose attacks. In *Proceedings of the 21st USENIX Security Symposium*, Security'12, pages 33–33, Berkeley, CA, USA, 2012. USENIX Association.

[21] Joseph Bonneau. Measuring password re-use empirically, 2011. `http://www.lightbluetouchpaper.org/2011/02/09/measuring-password-re-use-empirically/`.

[22] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The Quest to Replace Passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 553–567, May 2012. `http://www.cl.cam.ac.uk/~jcb82/doc/BHOS12-IEEESP-quest_to_replace_passwords.pdf`.

[23] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin Cookies: Session integrity for web applications. In *Web 2.0 Security and Privacy (W2SP)*, 2011.

[24] M. Brian. Gawker media is compromised. the responsible parties reach out to tnw [updated], 2010. `http://goo.gl/0SvCj`.

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

[26] Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-Exfiltration: The dangers of browser-enforced information flow control. In *Web 2.0 Security & Privacy (W2SP)*, 2012.

[27] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *Proceedings of the 15th USENIX Security Symposium*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. `http://dl.acm.org/citation.cfm?id=1267336.1267337`.

[28] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. A usability study and critique of two password managers. In *Proceedings of the 15th USENIX Security Symposium*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. `http://dl.acm.org/citation.cfm?id=1267336.1267337`.

[29] Graham Cluley. 600,000+ compromised account logins every day on Facebook, official figures reveal, 2011. `http://nakedsecurity.sophos.com/2011/10/28/compromised-facebook-account-logins/`.

[30] Alexei Czeskis and Dirk Balfanz. Protected login. In *Proceedings of the Workshop on Usable Security (at the Financial Cryptography and Data Security Conference)*, March 2012.

[31] Alexei Czeskis, Michael Dietz, Tadayoshi Kohno, Dan S. Wallach, and Dirk Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[32] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen Wang. Lightweight server support for browser-based csrf protection. In *Proceedings of the 23rd International World-Wide Web Conference*, 2013.

[33] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against CSRF attacks. In *Lecture Notes in Computer Science*. Springer, September 2011. `https://lirias.kuleuven.be/handle/123456789/311551`.

[34] Tamara Denning, Kevin Bowers, Marten van Dijk, and Ari Juels. Exploring implicit memory for painless password recovery. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2615–2618, New York, NY, USA, 2011. ACM.

[35] Rachna Dhamija and Adrian Perrig. Deja vu: A user study using images for authentication. In *Proceedings of the 9th USENIX Security Symposium*, SSYM'00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[36] T. Dierks and C. Allen. The TLS protocol, version 1.0. `http://tools.ietf.org/html/rfc2246`, Jan 1999.

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

[38] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol, version 1.2. `http://tools.ietf.org/html/rfc5246`, Aug 2008.

[39] Tim Dierks and Christopher Allen. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force, January 1999. RFC-2246, `ftp://ftp.isi.edu/in-notes/rfc2246.txt`.

[40] Michael Dietz, Alexei Czeskis, Dan Wallach, and Dirk Balfanz. Origin-Bound Certificates: A fresh approach to strong client authentication for the web. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[41] Django Software Foundation. Cross site request forgery protection, 2012. `https://docs.djangoproject.com/en/dev/ref/contrib/csrf/`.

[42] EMC. RSA SecurID, 2013. `http://www.emc.com/security/rsa-securid.htm`.

[43] Dino Esposito. Take advantage of ASP.NET built-in features to fend off web attacks. Microsoft MSDN, 2005. `http://msdn.microsoft.com/en-us/library/ms972969.aspx`.

[44] Katherine M. Everitt, Tanya Bragin, James Fogarty, and Tadayoshi Kohno. A comprehensive study of frequency, interference, and training of multiple graphical passwords. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 889–898, New York, NY, USA, 2009. ACM.

[45] Facebook. Facebook login, 2013. `https://developers.facebook.com/docs/concepts/login/`.

[46] Facebook Corporation. The Facebook blog, 2011. `https://www.facebook.com/blog/blog.php?post=486790652130`.

[47] James Fallows. Hacked!, 2011. `http://www.theatlantic.com/magazine/archive/2011/11/hacked/308673/`.

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[49] Dinei Florêncio and Cormac Herley. C.: One-time password access to any server without changing the server. In *In: ISC 2008*, 2008.

[50] John Fontana. Stolen passwords re-used to attack Best Buy accounts, 2012. `http://www.zdnet.com/stolen-passwords-re-used-to-attack-best-buy-accounts-7000000741/`.

[51] Shirley Gaw and Edward W. Felten. Password management strategies for online accounts. In *Proc. SOUPS 2006, ACM Press*, pages 44–55. ACM Press, 2006.

[52] Google Chrome. Manager your website passwords, 2013. `http://support.google.com/chrome/bin/answer.py?hl=en&answer=95606`.

[53] Google Inc. Federated login for Google account users, 2013. `https://developers.google.com/accounts/docs/OpenID`.

[54] Google Inc. Google 2-step verification, 2013. `http://support.google.com/accounts/bin/answer.py?hl=en&answer=180744`.

[55] E. Grosse. Gmail account security in Iran, 2011. `http://googleonlinesecurity.blogspot.com/2011/09/gmail-account-security-in-iran.html`.

[56] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, pages 471–479, 2005.

[57] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks - stealing the pie without touching the sill. In *CCS*, 2012.

[58] Cormac Herley and Paul van Oorschot. A Research Agenda Acknowledging the Persistence of Passwords. *IEEE Security & Privacy Magazine*, 2011.

[59] I. Hickson. HTML5 web messaging. `http://dev.w3.org/html5/postmsg/`, Jan 2012.

[60] Matt Honan. How Apple and Amazon security flaws led to my epic hacking, 2012. `http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking/all/`.

[61] Human Rights Watch. How Censorship Works in China: A brief overview, 2006. `http://www.hrw.org/reports/2006/china0806/3.htm`.

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

[63] Inferno. Hacking CSRF tokens using CSS history hack, 2009. http://securethoughts.com/2009/07/hacking-csrf-tokens-using-css-history-hack/.

[64] Anil K. Jain, Patrick Flynn, and Arun A. Ross. *Handbook of Biometrics*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[65] Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K. Reiter, and Aviel D. Rubin. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium*, SSYM'99, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association. http://dl.acm.org/citation.cfm?id=1251421.1251422.

[66] Martin Johns and Justus Winter. RequestRodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, May 2006. http://databasement.net/docs/2006_owasp_RequestRodeo.pdf.

[67] Aaron Johnson. The referer header, intranets and privacy, 2007. http://cephas.net/blog/2007/02/06/the-referer-header-intranets-and-privacy/.

[68] Jonathan Kent. Malaysia car thieves steal finger, 2005. http://news.bbc.co.uk/2/hi/asia-pacific/4396831.stm.

[69] Krzysztof Kotowicz. Cross domain content extraction with fake captcha, 2011. http://blog.kotowicz.net/2011/07/cross-domain-content-extraction-with.html.

[70] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992.

[71] A. Langley. Protecting data for the long term with forward secrecy, 2011. http://goo.gl/YMpXy.

[72] Last Pass. LastPass password manager homepage, 2013. https://lastpass.com.

[73] Eric Lawrence. Fiddler web debugging proxy, 2012. http://www.fiddler2.com/fiddler2/.

[74] Mohammad Mannan and Paul C. van Oorschot. Leveraging personal devices for stronger password authentication from untrusted computers. *Journal of Computer Security*, 19(4):703–750, 2011.

[75] Ziqing Mao, Ninghui Li, and Ian Molloy. *Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection.* Financial Cryptography and Data Security. Springer-Verlag, Berlin, Heidelberg, 2009. `http://dl.acm.org/citation.cfm?id=1601990.1602012`.

[76] Giorgio Maone. NoScript, 2012. `http://noscript.net/`.

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

[78] Microsoft. Microsoft NTML, 2012. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa378749(v=vs.85).aspx`.

[79] Microsoft Corporation. Provide your users with secure authentication capabilities using Microsoft .NET passport, 2002. `http://msdn.microsoft.com/en-us/magazine/cc188941.aspx`.

[80] Microsoft Corporation. Fill in website forms and passwords automatically, 2013. `http://windows.microsoft.com/en-us/windows7/fill-in-website-forms-and-passwords-automatically`.

[81] MIT Kerberos Group. Kerberos: The network authentication protocol, 2013. `http://web.mit.edu/kerberos/`.

[82] Mozilla. Location-aware browsing, 2012. `http://www.mozilla.org/en-US/firefox/geolocation/`.

[83] Mozilla Support. Password manager - remember, delete and change saved passwords in Firefox, 2013. `http://support.mozilla.org/en-US/kb/password-manager-remember-delete-change-passwords`.

[84] Mozilla Wiki. Origin header proposal for CSRF and clickjacking mitigation, 2011. `https://wiki.mozilla.org/Security/Origin`.

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

[86] Atif Mushaq. Man in the Browser: Inside the zeus trojan, 2010. `http://threatpost.com/en_us/blogs/man-browser-inside-zeus-trojan-021910`.

[87] National Institute of Standards and Technology (NIST). National vulnerability database, 2012. `http://web.nvd.nist.gov/`.

[88] OWASP: The Open Web Application Security Project. OWASP CSRFGuard project, 2012. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

[89] Chris Palmer and Chris Evans. Certificate pinning via HSTS, 2011. http://www.ietf.org/mail-archive/web/websec/current/msg00505.html.

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

[91] Bryan Parno, Cynthia Kuo, and Adrian Perrig. Phoolproof phishing prevention. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography and Data Security, 10th International Conference, FC 2006, Anguilla, British West Indies, February 27-March 2, 2006, Revised Selected Papers*, volume 4107 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006.

[92] Andreas Pashalidis and Chris J. Mitchell. Impostor: a single sign-on system for use from untrusted devices. In *GLOBECOM*, pages 2191–2195, 2004.

[93] Passfaces Corporation. Passfaces, 2013. http://www.passfaces.com/.

[94] J.R. Prins. Interim report diginotar certificate authority breach "operation black tulip". Technical report, 2011. http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2011/09/05/diginotar-public-report-version-1/rapport-fox-it-operation-black-tulip-v1-0.pdf.

[95] HTML Purifier. CSRF magic, 2012. http://csrf.htmlpurifier.org/.

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

[97] E. Rescorla. Keying material exporters for transport layer security (TLS). http://tools.ietf.org/html/rfc5705, March 2010.

[98] Ændrew Rininsland. Internet Censorship listed: How does each country compare? http://www.theguardian.com/technology/datablog/2012/apr/16/internet-censorship-country-list.

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

[100] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[101] David Ross and Tobias Gondrom. HTTP header frame options – draft-gondrom-frame-options-01, 2012. http://tools.ietf.org/html/draft-ietf-websec-frame-options-00.

[102] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Proceedings of the Second international conference on Engineering Secure Software and Systems (ESSoS)*, 2010.

[103] 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%5F0.html, Feb 2012.

[104] Facebook Security. National cybersecurity awareness month updates, 2011. https://www.facebook.com/notes/facebook-security/national-cybersecurity-awareness-month-updates/10150335022240766.

[105] P. Seybold. Sony's response to the u.s. house of representatives, 2011. http://goo.gl/YkXSv.

[106] Ofer Shezaf. WHID 2008-05: Drive-by pharming in the wild, 2008. http://www.xiom.com/whid-2008-05.

[107] Christopher M. Shields and Matthew M. Toussain. Subterfuge: The MITM framework. http://subterfuge.googlecode.com/files/Subterfuge-WhitePaper.pdf, 2012.

[108] Anton Sidashin. CSRF: Avoid security holes in your drupal forms, 2011. http://pixeljets.com/blog/csrf-avoid-security-holes-your-drupal-forms.

[109] Softflare Limited. Hosting/e-mail account prices, 2011. http://www.softflare.com/index.php?id=11.

[110] Open Source. Google authenticator, 2012. https://code.google.com/p/google-authenticator/.

[111] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-by pharming, 2006. https://www.symantec.com/avcenter/reference/Driveby_Pharming.pdf.

[112] Brandon Sterne. Content Security Policy – unofficial draft 12, 2011. https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html.

[113] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert 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.

[114] Hai Tao and Carlisle Adams. Pass-Go: A proposal to improve the usability of graphical passwords. *International Journal of Network Security*, 7:273–292, 2008.

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

[116] R. Vogelei. Bluetooth-enabled device shipments expected to exceed 2 billion in 2013, 2011. `http://www.instat.com/press.asp?ID=3238&sku=IN1104968MI`.

[117] 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.

[118] Thomas Y.C. Woo, Thomas Y. C, Woo Simon, and Simon S. Lam. Designing a distributed authorization service. In *INFOCOM*, 1998.

[119] World Wide Web Consortium. Cross-origin resource sharing, 2012. `http://www.w3.org/TR/cors/`.

[120] Yubico. YubiKey, 2013. `http://www.yubico.com/products/yubikey-hardware/yubikey/`.

[121] Michal Zalewski. Postcards from the post-XSS world, 2012. `http://lcamtuf.coredump.cx/postxss/`.

[122] William Zeller and Edward W. Felten. Cross-Site Request Forgeries: Exploitation and prevention, 2008. `www.cs.utexas.edu/users/shmat/courses/library/zeller.pdf`.

[123] K. Zetter. Security cavities ail bluetooth, 2004. `http://www.wired.com/politics/security/news/2004/08/64463`.

[124] Zeljka Zorz. Facebook spammers trick users into sharing anti-CSRF tokens, 2011. `http://www.net-security.org/secworld.php?id=11857`.