Strengthening User Authentication through Opportunistic Cryptographic Identity Assertions

Alexei Czeskis
University of Washington
Seattle, WA
alexei@czeskis.com

Michael Dietz
Rice University
Houston, Texas
mdietz@rice.edu

Tadayoshi Kohno
University of Washington
Seattle, WA
yoshi@cs.washington.edu

Dan Wallach
Rice University
Houston, Texas
dwallach@cs.rice.edu

Dirk Balfanz
Google
Mountain View, CA
balfanz@google.com

ABSTRACT

User authentication systems are at an impasse. The most ubiquitous method – the password – has numerous problems, including susceptibility to unintentional exposure via phishing and cross-site password reuse. Second-factor authentication schemes have the potential to increase security but face usability and deployability challenges. For example, conventional second-factor schemes change the user authentication experience. Furthermore, while more secure than passwords, second-factor schemes still fail to provide sufficient protection against (single-use) phishing attacks.

We present PhoneAuth, a system intended to provide security assurances comparable to or greater than that of conventional two-factor authentication systems while offering the same authentication experience as traditional passwords alone. Our work leverages the following key insights. First, a user's personal device (*e.g.,* a phone) can communicate directly with the user's computer (and hence the remote web server) without *any* interaction with the user. Second, it is possible to provide a layered approach to security, whereby a web server can enact different policies depending on whether or not the user's personal device is present. We describe and evaluate our server-side, Chromium web browser, and Android phone implementations of PhoneAuth.

Categories and Subject Descriptors

D.m [**Software**]: Miscellaneous

General Terms

Design, Human Factors, Security

Keywords

Authentication, Login, Second Factor, Web

1. INTRODUCTION

The most common mechanism for users to log into web sites is with usernames and passwords. They're 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 [4, 14], at which point the compromise of one web site leads to compromise of others [6, 26]. 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 [9, 25], causing anywhere from minor annoyances, to financial harm, to very real threats to life and well-being [15, 21].

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.* [5] 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 [7] or TLS client certificates [11]). 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 paper 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.

In Section 2 we evaluate previous efforts at strengthening user authentication and establish the threat model and goals for our system in Section 3. Section 4 outlines the system at a high level while Section 5 delves into practical implementation details.

The Bonneau *et al.* study [5] 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 6, and discuss potential future directions in Section 7.

2. RELATED WORK

We examine some related work and how it attempts to address the security issues with passwords, and then use the lessons learned to motivate and inform our design goals in Section 3.

TLS CLIENT CERTIFICATES. One example of a password-less authentication system is TLS Client Authentication [12], where a TLS client certificate is used to authenticate a user. Using client certificates means that the client does not send the authentication secret (a private key) to the server, and that users cannot get phished (since there is no password to be entered). There are, however, several problems with TLS Client Authentication, which have impeded its widespread adoption across the Web:

- *Poor User Experience.* Since the certificate is needed during the TLS handshake, users must approve or reject its use before they can interact with the website. This leads to a user interface in which the browser asks the user to select an identity (usually in terms of having to select a "distinguished name" or an "X.509 certificate") without presenting any context about where and how that identity will be used.
- *Privacy.* Once a certificate has been installed on a user's machine, any site on the web can request TLS client authentication with that certificate. The user now has two options, do not log in at all or choose to log into more than one site with the same certificate. Logging into more than one website with the same certificate is possible, but creates the potential for colluding sites to track a user's browsing habits by observing the certificate used to authenticate.

  Another approach is for the user to create a different certificate for each site he authenticates to, but this leads to even worse user experience: Now the user is presented with an ever growing list of certificates every time he/she attempts to authenticate to a site requiring TLS client authentication.
- *Portability.* Certificates ideally are related to a private key that cannot be extracted from the underlying platform. Therefore, they cannot be easily moved from one device to another. Hence, any solution that involves TLS Client Authentication also must address and solve the problem of user credential portability. Potential solutions include re-obtaining certificates for different devices (which can be a difficult process by itself), extracting private keys (against best security practices) and copying them from one device to another, or cross-certifying certificates from different devices.

We belive that the hassle of obtaining TLS client certificates and the unfamiliar user interface leads users towards using less secure (but more usable) password mechanisms when given the choice.

CARDSPACE. Microsoft developed the *CardSpace* [7] authentication system that had several features relevant to our work. Most notably, it 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.

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. We strive to learn from CardSpace's failure and have carefully designed our system to minimally alter the user experience (and burden on developers) from what users (and developers) are already used to.

FEDERATED LOGIN. The general approach behind federated login allows users to have only one account – at an *identity provider* – to which they directly authenticate (or perhaps have a limited number of such accounts). All other websites (usually called *relying parties*) do not ask the user to authenticate directly – instead they consume *identity assertions* from the identity provider.

OpenID [22], Facebook Connect [3], OpenID Connect [24], and Security Assertion Markup Language (SAML) are examples of this approach: after the user logs into the identity provider and approves the issuance of the identity assertion, the identity provider sends the identity assertion to the user's browser, which then sends it to the relying party. BrowserID [2] works in a slightly different manner: here the browser, and not the identity provider, issues the identity assertion (although the identity assertion includes a certificate from the identity provider, which the user has to obtain ahead of time).

Federated login carries the promise of fewer passwords that users need to manage. Of course, this promise can only be met when most sites on the web are relying parties to at most a handful of identity providers. The value proposition for a website to become a relying party to an identity provider however, is not always a given: what if the identity provider is insecure, or goes out of business, or does not effectively block accounts that have been taken over by attackers? Similarly, users may not be comfortable with the identity provider knowing which sites the users frequent or relying parties learning so much about users' identities from the identity provider – they might want to choose to have a new identity at a relying party site instead of "reusing" their identity from the identity provider.

Finally, federated login really just reduces one problem (that of securely authenticating to relying parties) to a previously unsolved one (that of securely authenticating to an identity provider). It does not, by itself, address the issue of users getting phished for their password at the identity provider, or sharing that password with such sites that decide not to work with a user's identity provider.

TRADITIONAL TWO-FACTOR AUTHENTICATION. Some websites use a variety of two-factor authentication schemes. In some cases, the user has to enter, in addition to the password ("something that they know"), some other code that they obtain from a device they carry ("something that they have"). In other cases (*e.g.,* smart cards/tokens), users must actually plug the device into the PC on which they're authenticating.

Apart from the immediately obvious usability issues (the user has to learn about the second factor and not forget to carry the device with them), there are some more subtle ones: If a user opts into, say, Google's 2-Step Verification system, then some of their legacy apps or devices might stop working, since they use protocols (such as IMAP, SMTP or XMPP) that assume a single-factor authentication mechanism. This in turns leads to further complications that the user has to deal with. In Google's case there are machine-generated passwords (called "Application-Specific Passwords") that the user has to learn about, backup options that need to be configured in case the second-factor device goes missing, *etc*. As a result, users that sign up for two-factor authentication are more likely to be locked out of their account than those users that use only passwords.

Apart from raising the cognitive load on the user, two-factor authentication does not completely solve the security issues of passwords: while it does address the issue of re-using passwords across websites, a clever attacker could theoretically phish a victim both for their password and second factor.

ADVANCED PASSWORD MANAGERS. PwdHash [23], Password Multiplier [16], and PassPet [29] are examples of advanced kinds of password managers that are built into browsers: they prevent phishing attacks and reduce password sharing by enforcing that different sites will receive different passwords (in particular, the legitimate site and the impostor site will receive different passwords). These advanced password managers, however, come with their own set of usability issues [8], ranging from the fact that users no longer know the passwords for certain sites to problems with the interaction model (users sometimes have to press certain key combinations to invoke the password manager, or be careful to only type the password into the browser chrome).

OTHER RELATED WORK. The TLS-SA [19] work by Oppliger *et al.* shares many of our insights. Unfortunately, it does not provide long-lived TLS sessions and hence will not allow credential binding in the same manner as our system. Additionally, the TLS-SA papers do not take a firm stance on what the user experience (UX) should be, instead enumerating a number of possible UXs, none of which score well in the Usability section of the Bonneau *et al.* [5] matrix.

The previously mentioned study by Bonneau *et al.* lists more "non-standard" authentication mechanisms and critically analyzes them. We agree with most of that analysis and refer the interested reader to that work.

One take-away of the Bonneau *et al.* work is that authentication schemes where the user experiences strays from the traditional "username + password" model have difficulty overcoming the barrier for adoption. What's more, much of the previous work in this area dramatically shifts the user experience of login (*e.g.,* by requiring a second factor or redirecting the user to an identity provider) while falling prey to some of the same attacks as passwords do – *e.g.,* phishing.

3. GOALS AND ASSUMPTIONS

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

- 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 redesign 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).

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 [20] 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 [17].

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. ARCHITECTURE

4.1 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 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 as 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. Recently, Czeskis *et al.* [10] introduced the concept of *Protected Login* whereby they group 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.

The work by Czeskis *et al.* shows 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) recently introduced in [13]. 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.

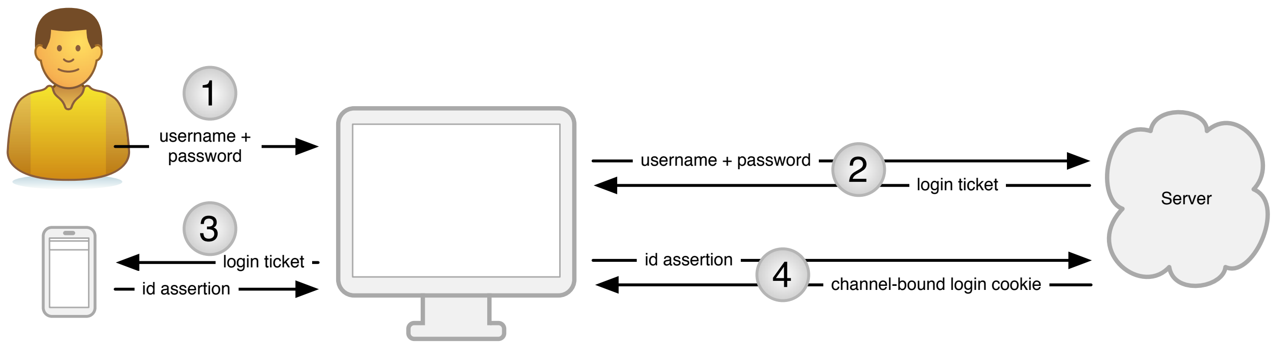

Figure 1: PhoneAuth Overview

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.

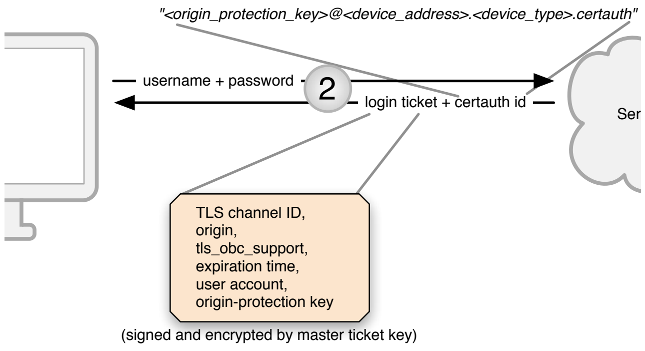
"<origin_protection_key>@<device_address>.<device_type>.certauth"
username + password
2
login ticket + certauth id
Ser
TLS channel ID,
origin,
tls_obc_support,
expiration time,
user account,
origin-protection key
(signed and encrypted by master ticket key)

Figure 2: Login ticket structure

4.2 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 is shown in Figure 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

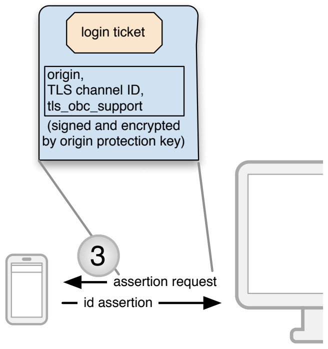

Figure 3: Assertion request structure

After receiving the login ticket, the browser generates an *assertion request* (shown in Figure 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 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). 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 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

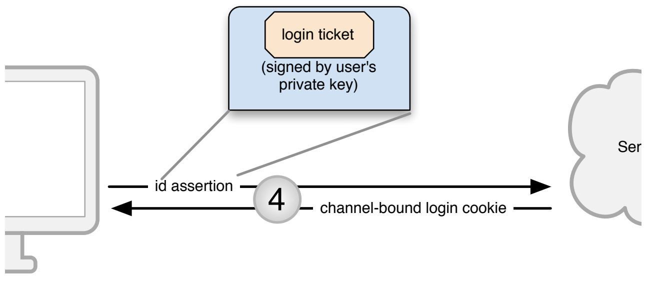

Figure 4: Identity assertion structure

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[1].

4.3 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 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 5.

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.

4.4 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).

5. 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 [30], 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 3).

5.1 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 /

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

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 3. For the second mechanism, the PC can obtain the phone's Bluetooth MAC address from the already existing (and lower latency) server connection[2].

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

5.2 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. 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 [27] 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 4.1) 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

[2]This must be done carefully, lest the designer creates a Bluetooth address oracle. See Section 7 for more discussion of this pitfall.

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.

6. EVALUATION

6.1 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 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 1.

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.

6.2 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 [13]. That work shows 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

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

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.

7. 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 [28]. 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 [18]. 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.

8. CONCLUSION

In this paper 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 this concept "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.

9. ACKNOWLEDGEMENTS

We would like to thank the team at Google, including Mayank Upadhyay, Diana Smetters, and Alex Klyubin for helping us develop the ideas presented in this paper, and for helping with the implementation. We would also like to thank the anonymous reviewers of our manuscript for helping us make this a better paper.

References

[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] Core Concepts - Authentication, 2012. `https://developers.facebook.com/docs/authentication/`.

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

[5] J. Bonneau, C. Herley, P. C. van Oorschot, and F. 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.

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

[7] D. Chappell. Introducing windows cardspace. http://msdn.microsoft.com/en-us/library/aa480189.aspx, April 2006.

[8] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[9] G. 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/`.

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

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

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

[13] M. Dietz, A. Czeskis, D. Wallach, and D. Balfanz. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *Proc. 21st USENIX Security Symposium*, 2012. Preprint obtained from authors.

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

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

[16] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 471–479, New York, NY, USA, 2005. ACM.

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

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

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

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

[21] J. 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.

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

[23] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[24] 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

[25] F. Security. National Cybersecurity Awareness Month Updates, 2011. https://www.facebook.com/notes/facebook-security/national-cybersecurity-awareness-month-updates/10150335022240766.

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

[27] O. Source. Google Authenticator, 2012. https://code.google.com/p/google-authenticator/.

[28] R. Vogelei. Bluetooth-Enabled Device Shipments Expected to Exceed 2 Billion in 2013, 2011. http://www.instat.com/press.asp?ID=3238&sku=IN1104968MI.

[29] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *Proceedings of the second symposium on Usable privacy and security*, SOUPS '06, pages 32–43, New York, NY, USA, 2006. ACM.

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