Protecting Web-based Single Sign-on Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-directional Authenticated Secure Channel

Yinzhi Cao[†], Yan Shoshitaishvili[‡], Kevin Borgolte[‡],
Christopher Kruegel[‡], Giovanni Vigna[‡], and Yan Chen[†]

[†]Northwestern University,
{yinzhi.cao@eecs, ychen@cs}.northwestern.edu
[‡]University of California, Santa Barbara
{yans, kevinbo, chris, vigna}@cs.ucsb.edu

Abstract. Web-based single sign-on describes a class of protocols where a user signs into a web site with the authentication provided as a service by a third party. In exchange for the increased complexity of the authentication procedure, SSO makes it convenient for users to authenticate themselves to many different web sites (*relying parties*), using just a single account at an *identity provider* such as Facebook or Google.

Single sign-on (SSO) protocols, however, are not immune to vulnerabilities. Recent research introduced several attacks against existing SSO protocols, and further work showed that these problems are prevalent: 6.5% of the investigated relying parties were vulnerable to impersonation attacks, which can lead to account compromises and privacy breaches. Prior work used formal verification methods to identify vulnerabilities in SSO protocols or leveraged invariances of SSO interaction traces to identify logic flaws. No prior work, however, systematically studied the actual root cause of impersonation attacks against the relying party.

In this paper, we systematically examine existing SSO protocols and determine the root cause of the aforementioned vulnerabilities: the design of the communication channel between the relying party and the identity provider, which, depending on the protocol and implementation, suffers from being a one-way communication protocol, or from a lack of authentication. We (a) systematically study the weakness responsible for the vulnerabilities in existing protocols that allow impersonation attacks against the relying party, (b) introduce a dedicated, authenticated, bi-directional, secure channel that does not suffer from those shortcomings, (c) formally verify the authentication property of this channel using a well-known cryptographic protocol verifier (ProVerif), and (d) evaluate the practicality of a prototype implementation of our protocol.

Ultimately, to support a smooth and painless transition from existing SSO protocols, we introduce a proxy setup in which our channel can be used to secure existing SSO protocols from impersonation attacks. Furthermore, to demonstrate the flexibility of our approach, we design two different SSO protocols: an OAuth-like and an OpenID-like protocol.

1 Introduction

The proliferation of web applications on the Internet has led to a sharp increase in the number of accounts (and corresponding credentials) that a web user has to create and remember. Inconveniences stemming from having to keep track of the accounts, and the tendency of web sites to suffer from security breaches, in which user credentials are exposed, has resulted in a recent push to adopt *single sign-on* (SSO) systems more widely[1]. As the name implies, these systems help reduce the large amount of account credentials a web user has to keep track of, by replacing these credentials with a single identity with which she can authenticate herself at many different web sites.

SSO systems allow users to sign into a web site (the *Relying Party*, or RP), such as StackOverflow, with authentication provided by a third party (the *Identity Provider*, or IdP), like Facebook or Google. While this greatly increases the convenience for both the users and the web site operators, it also brings new opportunities for attackers. A recent analysis of web-based SSO protocols by Wang et al. [37] identified five vulnerabilities in which a malicious attacker can impersonate a benign RP or intercept the communication between a benign RP and the IdP, leading to a compromise of the user's account's security. In fact, further research by Sun et al. [32] shows that 6.5% of RPs[2] are vulnerable to such impersonation attacks.

To understand SSO vulnerabilities better, formal security analysis has been carried out for existing SSO protocols. For example, AuthScan by Bai et al. [17] extracts the protocol specification from a SSO implementation and verifies the retrieved specification using formal analysis. On the other hand, ExplicatingSDK by Wang et al. [38] leverages formal analysis combined with a semantic model to identify hidden assumptions in the designs of the software development kits that are the foundation of many SSO implementations. Similarly, InteGuard by Xing et al. [39] correlates program invariants, to identify and mitigate logic flaws in SSO specifications, through a proxy situated between the user and the IdP. However, none of these prior approaches identified the actual root cause for vulnerabilities in SSO protocols that allow impersonation attacks, i.e., why these flaws exist in the current protocols in the first place. In this paper, we examined the design of existing SSO protocols and determined that the root cause of the aforementioned vulnerabilities lies in the broken design of the communication channel between the RP and the IdP. Depending on the protocol implementation, this channel either suffers from being a one-way communication protocol, or from a lack of authentication. This untrustworthy channel, in turn, makes existing protocols prone to impersonation attacks. Therefore, we propose to use a secure, authenticated, bi-directional channel between the RP and the IdP to prevent these attacks, by eliminating the root cause.

It is important to realize that the attacks discussed in this paper are not just theoretical. In fact, the vulnerabilities present in these protocols are currently being used by attackers to subvert authentication via SSO. Real-world, high-profile attacks have been carried out in the past on extremely popular web sites. One such example is last year's attack on the SSO communication between the web site of the New

[1] Clearly, using the same credentials on multiple web sites is not viable solution if data might leaked or a privacy breach might occur.

[2] The original text states "we also found that 13% of RPs use a proxy service from Gigya, and half of them are vulnerable to an impersonation attack."

York Times (the 38th most popular web site in the US; henceforth NYTimes), acting as the RP, and Facebook, acting as the IdP [37]. In this case, attackers exploited weaknesses in the design of current web-based SSO protocols that allowed them to access a victim's account with the same privileges that the victim had given to the NYTimes, including access to private user data and, in some cases, the ability to post messages on the user's behalf. Similar attacks have also been carried out against the communication between Facebook and Zoho.com, the JanRain wrapping layer over Google's SSO implementation, and other web sites that rely on SSO protocols [37].

This makes the solutions presented in this paper not just *theoretically interesting*, but actually *practically relevant* in solving a pressing problem in the area of user authentication, privacy, and web security.

Additionally, to support a smooth transition from existing systems, i.e., to give time for our design to be adopted, we introduce a *proxy* that can be deployed by web sites' operators (RPs in the current model) to secure existing, insecure SSO protocols. We are confident that our proxy design will ease the adoption of our secure SSO protocol and mitigate current vulnerabilities.

In this paper, we make the following technical contributions:

- *Dedicated, Bi-directional, Authenticated, Secure Channel.* Utilizing an existing in-browser communication channel, we establish a dedicated, bi-directional, authenticated, secure channel between the RP and the IdP. Leveraging public-key cryptography, an RP and an IdP authenticate each other and share a common secret, i.e., the session key. All further communication between the RP and the IdP is then encrypted with the session key and is kept secret from eavesdroppers.
- *Flexible SSO Protocol over the Secure Channel.* The aforementioned secure channel provides flexibility for SSO protocol designers. Specifically, our channel design provides a secure platform for customization by protocol architects. We discuss possible implementations of the most popular SSO protocol designs: an OAuth-like and an OpenID-like protocol, both build upon our channel design. In contrast, existing, verified protocols, such as the Secure Electronic Transaction (SET) [12] protocol, do not allow any further customization and require customers and users to strictly follow the protocol specification at hand.
- *Formal Verification of the Channel.* We formally verify the correctness and the security properties of our channel design with the cryptographic protocol verifier ProVerif by Blanchet [6] to ensure that the security guarantees hold in our respective threat model.
- *Performance Evaluation.* We evaluate a prototype implementation of our secure channel design. In our evaluation (Section 6), we show that the overhead our approach introduces is only about 650ms of initial latency (given a network delay of 50ms to reflect the latency observed at standard, residential Internet connections), i.e., a one-time authentication overhead of 650ms incurred at the beginning of each authentication session, which we believe is acceptable for the security guarantees it provides. In addition, we provide a detailed breakdown of the latency of each step of our SSO protocol implementation.
- *Gradual Deployment.* Apart from our clean-slate design, which must be deployed by the IdP, we introduce a proxy that acts as a "fourth-party" IdP. This proxy accommodates and protects existing SSO protocols and, thus, allows for a smooth and painless transition from existing SSO protocols to our more secure

one. Aiming for broad adoption and a general solution, we designed our proxy so that it can be deployed by a legacy RP or a legacy IdP. From the viewpoint of a legacy IdP, our proxy acts as an RP retrieving users' information in a controlled, secure environment; from the viewpoint of the legacy RP, it acts an IdP relaying the users' information it retrieved from the real IdP. To guarantee the various security properties that must hold to ensure protection against impersonation attacks with respect to the threat model, we formally verify the design of our proxy with ProVerif.

2 Threat Model

In this section, we introduce several key concepts relating to web-based SSO protocols and present several known attacks against current protocols and their implementations. For completeness, and to provide a better understanding of the threat model, we discuss attacks that are in the scope of this paper, as well as attacks that are out of scope.

2.1 Concepts

Generally, three entities are involved in an instance of a SSO protocol. The three participating parties are:

- *Identity Provider (IdP).* The identity provider serves as a centralized identification service for its users. Some examples of such identity parties are OpenID [5], Facebook Connect [23], and Google's single sign-on implementation. Additionally, there also exist aggregation services, such as JanRain Engage [2], which handle SSO services for a set of multiple IdPs. Special to the latter case, both JanRain Engage and the original IdP act as IdPs.
- *Relying Party (RP).* The relying party is a web site that uses the services an IdP provides to authenticate its users. The way users are authenticated differs between protocols: for an OpenID-like service, the RP acquires the user's identity with the IdP's signature; alternatively, for an OAuth-like service, the RP acquires a token or key from the user (who interacts with the IdP), which can then be used to fetch additional information from the IdP (e.g., to verify personal information or the identity of the user).
- *User.* The user is a client of both the IdP and the RP. The user maintains a SSO identity on the IdP's web site (e.g., an account with the IdP) and uses this identity to authenticate to different RPs. It is important to note that in our threat model, users are benign; in other words, we do not consider attacks initiated from the user's browser, and we require that the same-origin policy is enforced in the user's web browser.

```
GET https://www.idp.com/login?app_id=****
    &redirection_url=https://www.idp.com/granter?
    next_url=https://www.rp.com/login
Host: www.idp.com
Referer: https://www.rp.com/login
Cookie: ****
```

Listing 1.1. Example of a HTTP request from the RP to the IdP. Here, a malicious RP can initiate the communication by using a benign RP's app_id or intercept the communication by changing the redirection_url or next_url parameter.

2.2 In-scope Attacks

Recent research by Wang et al. [37] identified many vulnerabilities allowing *identity impersonation attacks* in SSO protocols, i.e., attacks where an adversary manages to spoof the identity of parties involved in the execution of the SSO protocol. In our threat model, we only consider the case where an attacker is able to impersonate the RP in an attempt to obtain a user's private information. In fact, these attacks account for five of the eight confirmed attacks on SSO protocols identified by Wang et al. [37] and are the most critical. The remaining three cases are out-of-scope, and we discuss the reason for this decision in more detail later on (Section 2.3). The five attacks that are in-scope can be classified into two categories:

- *A malicious RP initiates the attack.* In this case, as in the vulnerability between the New York Times (NYTimes) and Facebook, a malicious web site pretends to be the NYTimes and initiates the SSO process, i.e., an attacker simply sends a request using the application ID of the NYTimes (`app_id` in Listing 1.1) to spoof the identity of the NYTimes.
- *A benign RP initiates the request, but a malicious RP receives the response.* Here, as in a vulnerability found in the interaction between Zoho.com and Facebook, Zoho.com initiates the communication to Facebook, but a malicious party receives the response from Facebook, i.e., an attacker can change the `redirection_url` or the `next_url` in Listing 1.1 to receive the response that contains the token to access the user's information.

2.3 Out-of-scope Attacks

Since SSO protocols are a very broad topic, some categories of attacks are out-of-scope for this paper. Here, we list the attacks that we deem out-of-scope, and we argue why we decided to not take them into account. Generally, many of the attacks listed can already be mitigated leveraging prior work and we list them simply for completeness.

- *Social engineering, e.g. phishing.* We consider social-engineering attacks, such as the phishing of a user's credentials, to be out-of-scope because, in our opinion, the prevention of social-engineering attacks is more of an educational or user-interface issue than it is a protocol issue.
- *Compromised or vulnerable RP.* Bhargavan et al. [19] consider that an RP might be compromised and propose defensive JavaScript techniques to prevent untrusted browser origins. On the contrary, in our threat model, we assume that a benign RP remains integral and uncompromised. The communication channel between the benign RP and the IdP on the other hand is vulnerable as to that a malicious RP can control the channel fully. We exclude vulnerabilities related to such a compromised or vulnerable RP from our threat model because the user's information can already be obtained at the RP, and because we argue it is more of a secure deployment rather than protocol issue.
- *Malicious browser.* Some of the discovered attacks occur inside of the attacker's browser. For example, in an attack involving Google ID [37], the adversary is able to log into a user's account from his own machine because the RP does not check whether the email field is signed by the IdP. Those attacks cannot be prevented on a protocol level, and, therefore, they are out of scope.

– *Implementation issues.* Some of the discovered attacks exist because the RP and the IdP interpret the protocol or its specification differently. For example, in a vulnerability involving PayPal [37], the RP (PayPal) and IdP (Google) treat a data type differently.
– *Privacy leaks.* Uruena et al. [36] identify possible privacy leaks to third party providers (such as advertisers and corresponding industries) in the OpenID protocol.

Generally, we consider these attacks as out-of-scope because they are due to a user error, an implementation error (specific to improperly-implemented RPs or IdPs; potentially the result of poor documentation), or do not involve any communication between the RP and the IdP. In this paper, we solely focus on protocol level attacks.

3 Revisiting Existing SSO Designs and Attacks

To identify the root cause of the identity impersonation attacks, as defined by Wang et al. [37], it is critical to understand some main aspects of existing SSO protocol designs, especially the communication protocol between the RP and the IdP. Wang et al. [37] abstracted this communication by introducing the concept of a *browser-relayed message* (BRM), which describes a request from an IdP or an RP, to the RP or the IdP respectively, together with the resulting response. While this abstraction is already helpful for identifying vulnerabilities, it does not capture the root cause of many vulnerabilities in existing SSO protocol designs and leaves the communication unnecessarily complex. Instead, we abstract the protocol differently: the RP simply communicates with the IdP through an established channel. In this context, only two questions remain: (*i*) what is the identity of the parties involved (authenticity), and (*ii*) how do these parties communicate to achieve confidentiality and integrity?

3.1 Identity

As said, three parties are involved in a SSO instance. Understanding how each party is identified by the other parties is a prerequisite to analyze the root cause of impersonation attacks.

– *IdP.* The IdP is generally identified by its web origin, i.e., <scheme, host, port>.
– *User.* The user is identified by a unique identifier, such as their username or email address. While the identification of users can cause some confusion in an SSO protocol, attacks resulting from this confusion are out-of-scope because they are implementation or documentation errors. In this paper, we assume that there exists a correct and unique identifier for each user.
– *RP.* The identity of an RP can vary according to protocols imposed by different IdPs, but is a unique identifier nonetheless. For example, Facebook Connect uses the identifier “app_id” to identify an RP, while JanRain chose to adopt “AppName” and “settingsHandle” as the RP’s identifier [37].

Recent attacks show that the identity of a benign RP might be easily forgeable by a malicious RP. For instance, in the example involving Facebook and the NYTimes [37], the malicious RP spoofed/forged the identifier of the NYTimes. Because of this, we require an unforgeable identifier to represent the identity of an RP. In the same spirit as for the IdP, one can use the web origin as tracked by the

client browser. Since web origin tracking is already a basic security property that is enforced by all modern browsers, it is very hard for a malicious RP to forge it[3].

3.2 Communication between the RP and the IdP

Simply equipping the RP with an unforgeable identifier, however, does not mitigate existing vulnerabilities. During the execution of an SSO protocol, one must verify the identity of the RP at every step. To detail the problems caused by this, we carefully re-examine the communication between the RP and the IdP via a client's browser in existing protocols. We classify those interactions into two main categories: HTTP(s) requests to a third-party server and an in-browser communication channel.

HTTP(s) Requests to a Third-Party Server. In the first category, comprising of OpenID, Security Assertion Markup Language (SAML) [13], and AAuth [35], the RP and the IdP communicate with each other via HTTP(s) requests. The process of an RP trying to connect to an IdP is as follows. First, the RP's JavaScript code -running in the client's browser- sends a request to the RP. The RP sends a response containing an HTTP 3xx redirection (or, alternatively, some kind of other redirection, for instance, via automatic form submission) to the client. Based on this redirection, the RP's code in the client browser sends a request to the IdP. Finally, the browser communicates with the IdP and completes the authentication process.
The problem: The interaction via third-party HTTP(s) requests is a one-way channel, i.e., after an RP talks to an IdP, the IdP cannot send a response back. In order to actually receive a response, the RP needs to tell the IdP where to forward the client's browser upon authentication. Generally, this is done by utilizing a parameter in the request, such as the `next_url` parameter in Listing 1.1. The issue here is that this parameter can be modified by an attacker, and, in turn, can lead to an identity impersonation attack. To mitigate this vulnerability, a bi-directional communication channel is necessary.

In-browser Communication Channel. The second category describes protocols that communicate via an in-browser communication channel. This includes Facebook Connect[4] [23] and other protocols in which the RP and the IdP use JavaScript in the client's browser to communicate with each other. The different parties might communicate with each other through a number of mechanisms, such as postMessage [18], URI fragments [18], or even Flash objects [37].
The problem: Two issues remain with this approach. First, the in-browser communication channel is an undedicated, bi-directional channel without proper authentication. For each message exchanged between the RP and the IdP, the origin needs to be verified independently. Recent work has established that this verification step is frequently forgotten by developers [25, 31]. The omission of this crucial verification step can make the protocol vulnerable to impersonation attacks. For instance, Hanna et al. [25] determined that two prominent SSO protocols, Facebook Connect and Google Friend Connect, exhibit this problem. Further, Son et al. [31] identified 85 popular web sites that were using the postMessage API incorrectly. This demon-

[3] Impossible to forge, if web origin and its tracking is correctly implemented by the browser and configured properly by the web site operator, i.e., according to RFC6454

[4] Facebook Connect mixes the usage of HTTPS requests to a third-party server and an in-browser communication channel.

strates the requirement for a dedicated channel with built-in authentication for the RP-IdP communication. Second, the in-browser communication channel is insecure. In the vulnerability between Facebook and NYTimes [37], attackers exploited this fact to eavesdrop on the in-browser communication between the IdP and the RP and intercept users' access tokens while the authentication was taking place. Once the attacker obtains this access token, he can successfully impersonate the NYTimes to Facebook. Here, the channel's insecurity was introduced by the use of Flash objects. However, even if the communication channel would be changed from Flash objects to postMessage, the vulnerability would remain, as demonstrated by Barth et al. [18] and Yang et al. [40]. Furthermore, Cao et al. [21] show that the postMessage vulnerability originally proposed by Barth et al. still exists in modern browsers for requests from the same domain, such as different blogs or applications hosted under that same domain.

To mitigate all these threats, a *dedicated, bi-directional, secure channel with authentication* is required. In the next section, we discuss the steps involved in establishing such a channel.

4 Design

In the previous section, we argued for the necessity of a new, dedicated, bi-directional, secure channel between the IdP and the RP, along with a new approach to verify the identity of the RP. Following, we discuss the design of our SSO protocol, which addresses the shortcomings of prior protocol designs. We introduce two solutions: a clean-slate redesign and a legacy-compatible proxy. The clean-slate design (Section 4.1) must be deployed by the IdP as a new SSO service; the proxy design (Section 4.2) serves as a "fourth-party" IdP and is designed to be deployed by an RP that wants to protect its users from RP impersonation attacks on legacy SSO services, while still supporting authentication through existing, but insecure services.

4.1 IdP Deployment – Clean-slate Design

The core requirements of our clean-slate design of SSO are (i) the use of the web origin as an RP and IdP identifier, and (ii) the use of a dedicated, secure, bi-directional channel with authentication for all communication between the RP and the IdP (Section 3).

Identity Design. In our protocol, the identity of the RP is its web origin. There are two alternatives for choosing this identity:

- *Web Origin is defined by the RP.* If the RP has its own web origin, such as the NYTimes, the RP can simply define a sub-domain for its SSO communication. For example, the NYTimes could make use of *facebookconnect.nytimes.com* for all communication with Facebook Connect to authenticate users through SSO.
- *Web Origin is provided by the IdP.* If the RP does not have a web origin, such as in the case of an application designed by a third-party web developer, the RP can adopt a sub-domain origin of the IdP (for example, *application1.connect.facebook.com*) as its identity for communication with Facebook during the authentication of users through SSO.

In both cases, the RP can leverage the sandbox tag defined in HTML5 to ensure proper isolation of the web origin.

Communication Channel. Next, we will detail the life-cycle of a bi-directional, authenticated, secure communication channel over the following three steps: (i) establishing the channel, (ii) using the channel to communicate securely, and (iii) destroying the channel.

Establishing the Channel: Handshake Protocol. To establish a secure communication channel between the RP and the IdP, JavaScript first creates a secure socket that listens to connections from a given web origin (for example, the web origin belonging to the RP), optionally specifying a target window (such as *parent*), as shown in Listing 1.2. When the RP's web page is loaded, its JavaScript connects to the socket created by the IdP by specifying the target window (*e.g.*, a reference to an inline frame) and the target origin (*i.e.*, the IdP's web origin), c.f., Listing 1.2, line 11-14).

```
1  parameters = {
2   mWindow: parent, // optional when listening
3   mOrigin: IdP/RP Origin,
4   onMessage: function(m) { /* receiving message callback */ }
5   onConnect: function() { /* creating channel callback */ }
6   onDestroy: function() { /* destroying channel callback */ }
7  }
8  // Listening to a channel
9  var socket = new SecureSocket();
10 socket.listen(parameters)
11 // Creating a channel
12 var iframe = document.getElementById("myid");
13 var socket = new SecureSocket();
14 socket.connect(parameters)
```

Listing 1.2. JavaScript Primitive for the Channel.

The handshake protocol establishes the secure channel between the RP and the IdP, by exchanging their keys. The process includes the following five steps:

1. The RP verifies the identity of the IdP and sends its public key (PK_RP) to the IdP.
2. Upon receipt of the public key from the RP, the IdP verifies the identity (web origin) of the RP.
3. The IdP generates a session key (SK), encrypts it with the public key from the RP, and sends the encrypted session key and an encrypted partial channel number (PK_RP(N_IdP)) to the RP.
4. The RP decrypts the session key and partial channel number using its own private key and responds with an encrypted partial channel number (SK(N_RP)). This channel number, and the channel number generated by the IdP in (3) are then stored by the browser as an index to look up the session key.
5. With both parts of the channel number, the RP and the IdP can communicate with each other. Both N_IdP and N_RP are needed to send a message, and each message consists of a ControlByte (later used to determine the status of the channel) and the message content, encrypted with the session key.

In the aforementioned steps, N_IdP and N_RP are used to look up the session key by both the RP and the IdP, and the ControlByte is used to determine the status of the current channel, such as whether it has been destroyed. During the negotiation

period, both partial channel numbers (N_IdP and N_RP) are encrypted and protected. Later on, N_IdP and N_RP are not encrypted, and thus easily modifiable. If an attacker modifies N_IdP or N_RP, however, a different session key and message handler will be used to process the message, and, in turn, it will result in the message being delivered to an incorrect channel. In the absence of the correct keys, an entity listening on this channel will be unable to decrypt the message and fail.

Use of the Channel: Sending Messages (Figure 1). After the channel is created, messages can be sent. When either side wants to send a message, it calls the JavaScript primitive *socket.sendMessage(msg)*, which divides the input message into small chunks (to fit the key size), appends the control byte to indicate the status of current channel, encrypts all the divided chunks with the shared session key, and appends N_IdP+N_RP to the result. This message is then sent via the in-browser communication channel, which ensures the delivery of the message.

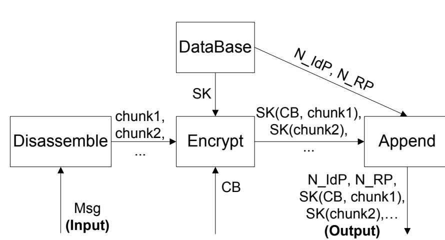

Fig. 1. Sending a message to the channel between the client-side RP and the client-side IdP.

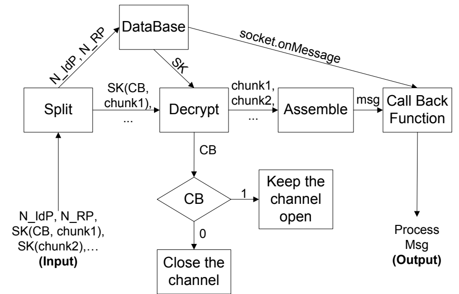

Fig. 2. Receiving a message from the channel between the client-side RP and the client-side IdP.

Use of the Channel: Receiving Messages (Figure 2). When a message is sent by one of the parties, the system first uses the N_IdP and N_RP to retrieve the session key and socket from the browser. The session key is used to decrypt the encrypted message chunks and the ControlByte. These chunks are then stitched back together accordingly and delivered to the processing function on the other end of the corresponding socket.

Destroying a Channel: Releasing Resources. In the process of destroying a channel, *socket.close()* is called. When either the IdP or RP calls the close method, the other side is notified and must close the channel as well.

For example, if the RP wants to close the channel during the communication, it sends a message with the ControlByte set to 0. When the IdP receives this message, it sends an equivalent response (with the ControlByte of 0) and releases resources such as the channel number, session key, and socket. After the RP receives the response message, it also releases its resources. This is a process analogous to the closing handshake of a TCP network socket.

SSO Protocol over the Channel. Our secure communication channel between the RP and IdP can now be used in the design of a secure SSO protocol. Traditionally, SSO is classified into two categories: OAuth-like protocols for authorization and OpenID-like protocols for authentication. In the former, the RP asks the IdP for an access token to fetch a user's information from the IdP; in the latter, the

RP asks the IdP to verify the identity of a user. We describe both paradigms in the context of our secure channel.

OAuth-like Protocol:
1. Client-side JavaScript code served by the RP initiates a connection with the JavaScript code served by the IdP by establishing the secure channel.
2. The RP JavaScript requests a token, which can be used to access a user's data, from the IdP JavaScript.
3. The IdP authenticates the user, usually by having them log into the IdP's service.
4. When the user authenticates successfully, the IdP asks the user for permission to allow the RP to access their information.
5. If permission is granted, the IdP then sends the token to the IdP JavaScript, which forwards the token to the RP JavaScript.
6. The RP JavaScript sends the token to the RP server, which then uses that token to request the user's information from the IdP service.

OpenID-like Protocol:
1. JavaScript served by the RP establishes a secure channel to JavaScript served by the IdP through our protocol.
2. The RP JavaScript asks the IdP JavaScript to authenticate the user.
3. The IdP server authenticates the user, usually by having them log into the IdP service.
4. Upon successful authentication, the IdP server sends an authentication proof (typically a token encrypted or signed with the IdP's private key) to the IdP JavaScript.
5. The IdP JavaScript relays the authentication proof to the RP JavaScript.
6. The RP JavaScript relays the authentication proof to the RP server.

4.2 RP Deployment – Proxy Design

Many existing, unprotected protocol SSO implementations are currently deployed, and migrating them to our secure design will take time since it requires manual development effort. To assist in securing these legacy implementations, we introduce a proxy that integrates them into our design, and allows for a seamless transition.

The proxy mediates the communication between an RP and the IdP (Figure 3). It acts as a legacy RP to the legacy IdP, authenticating against it like any other currently deployed RP, but with sufficient isolation to protect it from identity impersonation attacks. Additionally, this proxy acts as a secure IdP to the RP using our secure protocol.

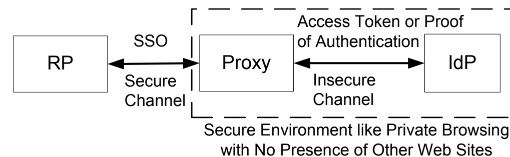

Fig. 3. Overview of the proxy design. The secure RP is talking to our proxy through the secure channel, and our proxy is talking to the legacy IdP using a legacy SSO protocol, but contained in a secure, controlled environment.

Communication with the legacy IdP. To communicate with the legacy IdP, the proxy needs to act as a legacy RP. For the IdPs of most existing SSO implementations (for example, Facebook Connect), we must register an application for each user, and keep the application ID secret from attackers to avoid any impersonation attacks. Specifically, only the legacy IdP and our proxy must know the application ID.

After registering an application with the IdP, the proxy authenticates normally with the IdP and acquires the access token (for an OAuth-like protocol) or proof of authentication (for an OpenID-like protocol). Since these tokens incorporate the *privileges* that an RP has over the authenticating user's account, and since some subset of these privileges will be ultimately used by the secure RPs that communicate through our proxy, the proxy must request the total set of permissions that will be needed by all the secure RPs. To accommodate different privileges levels, the proxy stores this confidential information inside its own database on the proxy server side.

Since the communication with the legacy IdP takes place using an insecure SSO implementation, the process should ideally be done in private browsing mode on a secure machine with no other web sites open. This is to prevent details such as the application ID of the proxy from being leaked to potential attackers.

Communication with the secure RP. After the initial setup, the secure RP communicates only with our proxy, instead of the legacy IdP. Communication between the RP and our proxy is done over the secure channel (Section 4.1). We first authenticate the user, and then, for OAuth-like protocols, generate our own token for the RP with a corresponding subset of the privileges granted to us by the legacy IdP. For OpenID-like protocols, the proxy can simply forward the authentication proof directly to the RP.

Fetching the User's Information. In an OAuth-like protocol, when the RP asks for the user's information using the token issued by us, the proxy sends a request for this information to the legacy IdP with the token issued by that legacy IdP. When the legacy IdP returns the requested information, the proxy forwards the information to the RP.

5 Implementation

In this section, we present the implementation details of our system. First, we introduce the implementation of our clean-slate design (Section 5.1). We implemented the authenticated, bi-directional, secure channel as a layer on top of the existing *postMessage* channel available in modern browsers, and then leveraged this channel to implement our SSO protocol. Next, we discuss a prototype implementation of our proxy design (Section 5.2).

5.1 IdP Deployment

Our implementation of the clean-slate design is very lightweight: it consists of only 252 lines of JavaScript code (excluding external libraries), 264 lines of HTML code, and 243 lines of PHP code. The bi-directional secure channel with authentication is implemented exclusively in JavaScript; the RP-side and IdP-side code is implemented in HTML and PHP.

Our JavaScript implementation of the secure channel uses two external libraries, the JavaScript Cryptography Toolkit [3] and the Stanford JavaScript Crypto

Library [14], both for cryptographic purposes. The former is used for public and private key generation and asymmetric encryption and decryption; the latter is used for session key generation and symmetric encryption and decryption. The public and private keys in the implementation are 512 bits, and the session key is 128 bits. However, these are implementation details, and can be modified to address different security requirements. As mentioned before, we implemented the protocol by adding our security layer on top of the *postMessage* functionality available in modern browsers, which provides a reliable, but insecure, communication channel that guarantees in-order delivery of messages.

We use a socket pool for session tracking. For each incoming message, the system looks in the socket pool for a channel with the corresponding N_IdP and N_RP and, if one is found, fetches the socket and session key. Otherwise, a new socket is created (Section 4.1), and kept track of in the socket pool.

To evaluate the practicality of our design, we created reference IdP and RP implementations. A login interface for the IdP prompts the user to input his username and password. The IdP's server-side code looks up the credentials in a database and, if the username and password match, it generates an access token and transfers the token to the IdP JavaScript. Since the authentication process involves multiple server-side pages, we load an iframe to embed the authentication process and communicate with the RP. This iframe talks to the IdP iframe, which is responsible for authentication through another secure channel.

5.2 Proxy RP Deployment

To demonstrate the feasibility of our proxy design, we implemented a prototype RP that authenticates against Facebook through our proxy. The proxy process works as follows: first, a user who wants to authenticate via SSO through our proxy needs to register on our site. We enable the Facebook developer account for that user and let the user register a Facebook application under his account. This process is manual since it involves solving a simple captcha and acknowledging the terms of service, but could theoretically be automated if endorsed by Facebook. The user then provides the proxy's application's ID and secret key, so that the proxy can act as the application. While technically not necessary, we use the user's account to register an application because it lets the user maintain full control over the application, and the overall security is tied to the user's account. The whole process is done only once per user, as would happen when installing a normal application. A similar system can be implemented for Google's SSO implementation, which also adopts a user application key and secret to authenticate an application. In the OAuth case, the user will then grant the proxy permissions to access his/her user data. As mentioned previously, these permissions are a superset of those that the proxy can provide to an RP.

Afterwards, when a user visits an RP supported by our proxy, the SSO process will be the same as with the clean-slate design in the IdP deployment. We use Facebook's SDK [1], safely isolated from the RP, on the server-side to fetch the user's data and forward them to the RP.

There are, however, some implementation concerns unique to the proxy deployment:

Secrecy of the legacy IdP application ID. To guarantee the security of the proxy design, the application ID used by the proxy to identify itself at the legacy

IdP must remain secret. In the case of the legacy IdP being Facebook Connect, we take two measures to ensure this secrecy. First, we register a unique Facebook application ID for each user. This is done so that an attacker cannot learn the application ID by using the proxy service himself. Second, certain legacy IdPs, such as Facebook, provide a sandbox mode [11] for developer use. This sandbox mode isolates the application from all users except for the developer. In our case, the "developer" is the user of the proxy implementation, and this functionality allows us to keep the application ID secret.

Legacy IdP Terms of Service. When implementing the proxy design, care must be taken to avoid violating the legacy IdP's terms of service. We have reviewed Facebook's terms of service and determined that encouraging every user to enable Facebook Developer Mode is acceptable. Additionally, we made the legacy IdP portion of the proxy as lightweight as possible to avoid putting any significant load on Facebook's infrastructure.

Legacy IdP access token expiration. Tokens acquired from legacy IdPs, such as Facebook Connect, have an expiration time past which they cease granting access to a user's data. For Facebook Connect, this expiration is 60 days from the token's issuance time. After these 60 days, a reacquisition of the token by our proxy has to occur. This requires the user to repeat the initial setup process, which should be done in a private browsing mode to guarantee security. Tokens granted by the proxy to the RP do not expire in the current implementation, but introducing the functionality is straight-forwarding and requires only some engineering effort.

6 Evaluation

To evaluate our design, we first formally verify the channel (Section 6.1). Following, we empirically examine existing vulnerabilities (Section 6.2). Finally, we study the performance of our implementation to show that our protocol incurs only a reasonable overhead for the security guarantees it provides (Section 6.3).

6.1 Formal Protocol Verification

We verify the correctness of our protocol with ProVerif [6, 20], an automatic cryptographic protocol verifier based on the Dolev-Yao model [22]. ProVerif can prove properties including secrecy, authentication, strong secrecy, and equivalences of a protocol. For our formal verification, we require only the former two properties.

Channel Verification. We model the channel and the attacker, and then verify the channel using the attacker model.

Channel Model. Using ProVerif, we model our secure channel by using two processes and one free channel according to our threat model (Section 2). These two processes model the client-side and the server-side. The free channel, exposed to the attacker by definition, is used to communicate between the client and the server.

First, the RP sends its public key to the IdP. Second, the server verifies the web origin (defined as a bitstring transmitted together with the public key in the first step). This happens because browsers always send the web origin as part of a message via the postMessage functionality. If the origin matches, we generate a symmetric session key. Following, on the IdP side, the session key and partial channel number N_IdP are generated and encrypted. After receiving the encrypted session key,

the RP decrypts the message. It then encrypts its own, generated partial channel number N_RP with the session key. In the end, the IdP can send messages securely by encrypting messages with the session key and appending N_IdP and N_RP.

Attacker Model. In ProVerif, an attacker in the context of a secure channel is an active and passive adversary who is capable of sending messages to either party and also capable of eavesdropping on messages sent across the channel.

Results. After modeling the channel and attacker, ProVerif determined that an attacker would be unable to obtain the plaintext of an encrypted message sent over the secure channel. To further validate this assessment, we introduced an intentional vulnerability into the protocol and verified that ProVerif was able to produce a valid attack scenario in which the adversary was able to read the message in the clear.

For this vulnerability, we simply removed the origin check, of the RP, that is performed by the IdP. Once removed, ProVerif produced a counterexample showing the attack scenario. The produced attack works as follows: a malicious RP directly sends its own PK_RP to the IdP, and, in turn, acquires the session key SK and channel number N_IdP. Since the IdP does not verify the RP's identity, the malicious RP can impersonate the legitimate RP and talk to the IdP as if it is the legitimate RP.

SSO Protocol Verification. We model the SSO protocol and the attacker in ProVerif, and we verify its security guarantees in respect to our threat model.

SSO Protocol Model. We model the SSO protocol with two secure channels and three processes. The three processes represent the client-side RP, the client-side IdP and the server-side IdP. The client-side RP and the client-side IdP communicate with each other over the secure channel (Section 6.1). The client-side IdP and the server-side IdP communicate through HTTPS, a well-known secure channel.

Attacker Model. In respect to our threat-model, attackers are either network attackers or web attackers, as defined by Akhawe et al [15]. A network attacker is a passive adversary capable of eavesdropping on the network traffic, while a web attacker can also control malicious web sites and clients.

Results. Both channels are verified as secure with properly authenticated peers. Because of this, an attacker cannot acquire any useful information from either channel. ProVerif confirms this result.

Proxy Design Verification. To validate our proxy design, we model and verify the proxy in ProVerif.

Proxy Design Model. We model the proxy design with one secure channel, one insecure channel, and three processes. These three processes correspond to the RP, our proxy (the new, secure, fourth-party IdP), and the real, legacy IdP. The channel between the RP and our proxy is secure (as established in Section 6.1), and the channel between our proxy and the legacy IdP is insecure.

Attacker Model. The attacker model follows the attacker model definition from the Channel Verification (Section 6.1).

Results. Since all communication of our proxy and the real IdP occurs over insecure channel, a malicious RP can intercept any ongoing communication. Unsurprisingly, ProVerif yields that the information transmitted over the channel between our proxy and the real IdP can be obtained by an attacker. Thus, as discussed in Section 4.2, it is crucial that we need to keep the identity of our proxy at the real IdP confidential (to prevent impersonation attacks on the proxy) and

that the initial setup, during which communication between the proxy and the real IdP occurs, is performed in a secure environment, such as in private browsing mode with no other web sites open to which information of the proxy might leak.

6.2 Security Analysis

Following, we study several existing RP impersonation vulnerabilities [37] and show how our design mitigates such vulnerabilities effectively.

Facebook and NYTimes. The first vulnerability we discuss was found in the interaction between the NYTimes as the RP and Facebook as the IdP. In this vulnerability, a malicious RP impersonates the NYTimes by spoofing its app_id. In turn, Facebook continues to authenticate the user and generate an access token that is supposed to be delivered to the NYTimes. However, due to nature of cross-domain communication in Adobe Flash, i.e., its "unpredictability" [4], the in-browser channel can be eavesdropped on and the access token is being leaked to the malicious RP, the actual initiator of the authentication request.
Mitigation. The vulnerability has two major components: a malicious RP impersonating the NYTimes and the insecure communication channel between the NYTimes and Facebook. Both vulnerabilities are mitigated in our protocol by design. First, we leverage web origins to verify the identity of the RP, thus a malicious RP cannot impersonate the NYTimes. Second, the communication channel between the NYTimes and Facebook is over a secure channel with authentication, therefore, even if a malicious RP would acquire the messages transmitted, it cannot easily decrypt and retrieve the clear message.

JanRain Wrapping GoogleID. A second interesting vulnerability was found in how JanRain wraps GoogleID [37]. Here, a malicious RP registers itself with JanRain. Initially, the malicious RP initiates the communication. Then, once JanRain redirects the communication to Google, the malicious RP impersonates the victim RP and sets the return URL to its own URL. Since Google is using the HTTP redirection method (c.f., Section 3.2) to communicate with the RP, confidential information will be leaked to the malicious RP.
Mitigation. In our design, when the RP talks to the IdP, all communication occurs over a bi-directional secure channel. As such, when the real RP talks to the IdP, the IdP will respond to the original, legitimate RP rather than to a different, malicious RP. Thus, the vulnerability is mitigated by design.

Facebook and Zoho. Similar to the vulnerability found in how JanRain wraps GoogleID, a vulnerability in the use of HTTP redirection was discovered between Facebook and Zoho.com [37]. To exploit this vulnerability, after receiving the authorization code from the `redirect_url` of Zoho, the attacker sends a request to Zoho and sets the `service_url` to a malicious URL. Zoho fails the SSO, but still redirects the user to the malicious URL.
Mitigation. The root cause of this vulnerability is very similar to 2). Since our protocol leverages a bi-directional secure channel, a return URL is not required in our design, thus preventing this vulnerability by design.

JanRain Wrapping Facebook. In this vulnerability, JanRain wraps Facebook as the IdP [37]. Here, *sears.com* incorrectly set its whitelist to **.sears.com*

rather than to the more restrictive *rp.sears.com*, thus exposing it to an attack similar to the *document.domain* attack by Singh et al. [30].
Mitigation. Since our design specifically uses web origins as the identity of the RP, the concept of a whitelist is irrelevant, thus preventing the misconfiguration of any such whitelist by design.

Facebook Legacy Canvas Auth. Lastly, a third vulnerability that allows *IdP* impersonation rather than *RP* impersonation was discovered in how Facebook is being used as the IdP. In this case, the vulnerability lies in in the fact that, for Facebook's legacy canvas authentication [37], the generated signature is not properly verified by a Facebook app (specifically, *FarmVille.com*). Because of this, the RP renders itself vulnerable to IdP impersonation attacks, leading to arbitrary user impersonation attacks and resulting leaks of private or confidential user data.
Mitigation. Since we transmit every message in an established secure channel that provides authentication, it is not necessary for the RP to verify any signature itself. For this reason, it reduces the risk of missing or omitted signature verification steps and preventing the impersonation attacks that might arise.

6.3 Performance Analysis

Environment and Methodology. Our client-side experiment was performed on a 2.67GHz Intel(R) i7 CPU with four physical cores and 8GB of memory running Ubuntu 13.04 64-bit. The browser at the client-side was Firefox 22.0 in the 32-bit version. The RP server was deployed on a CentOS 64-bit server with a 2.50GHz Intel(R) Xeon(R) CPU with eight physical cores and 16GB memory. The IdP server was deployed on a CentOS 64-bit server with a 2.80GHz Intel(R) Xeon(R) CPU with four cores and 16GB memory. The average round-trip network latency between the client and the two servers was measured to be about 50ms.

To measure the delay of each step of the secure channel and an instantiation of our SSO protocol, we make use of the JavaScript primitive *Date()* and subtract the monitored value at start and end points to calculate the delay. Each experiment was repeated ten times to calculate the average delay and to account for outliers and deviations.

Table 1. Breakdown of the authentication performance of our prototype implementation.

Operation	Delay
(1) Creating the Channel between RP and IdP	164±11ms
(2) Creating IdP Iframe	57±3ms
(3) Sending the First Message from RP to IdP	32±2ms
(4) Creating IdP Iframe for Authentication	57±3ms
(5) Creating the Second Channel inside IdP	165±11ms
(6) Authenticating the User	56±4ms
(7) Getting the User's Permission	57±3ms
(8) Sending the Token inside IdP Iframe	32±2ms
(9) Sending the Token to RP	33±2ms
Total	653±21ms

Note: Step (2), (4), (6) and (7) are extremely depended on the network latency.

Performance of SSO Implementation. We measure the performance of our prototype implementation of our SSO protocol. The results are shown in Table 1 and they are divided into three categories: channel creation, network delay, and message passing. (1) and (5) correspond to channel creation, which takes about 164ms. (2),

(4), (6) and (7) are almost exclusively driven by the network delay. (2) and (4) are to fetch content from the IdP, and (6) and (7) are used to communicate with the IdP. In the end, (3), (8), and (9) are used to communicate between iframes within the browser through our established secure channel, which is about 32ms. Overall, the total overhead of our prototype implementation is only 653ms, i.e., acceptable from a user's perspective given the strong security guarantees of our SSO protocol.

7 Related Work

We present related work that looks for vulnerabilities in existing SSO protocol designs and work investigating protection mechanisms for existing SSO protocols.

7.1 Vulnerability Identification

Much research has been carried out to identify vulnerabilities in existing SSO protocols. The work falls into three types: manual to mostly manual analysis, automatic analysis, and user studies.

Table 2. Comparison and positioning of our work with related work.

	Deployment at	**Protecting**	**Preventing Impersonation Attacks**	**Proactive Deployment**
InteGuard [39]	IdP and Gateway	IdP users and in-network devices	✓	✗
AUTHSCAN [17]	IdP	IdP users	✓	✗
Explicating SDKs [38]	IdP	IdP users	✓	✗
Defensive JavaScript [19]	IdP and RP	IdP and RP users	✗	✓
Our Work	IdP and RP	IdP and RP users	✓	✓

- *Manual analysis to mostly manual analysis.* Wang et al. [37] propose a browser relay message analyzer and identify eight vulnerabilities in protocol implementations through by manually analyzing them. Sun et al. [32] perform an empirical study to identify vulnerabilities in three major OAuth identity providers (IdP) (Facebook, Microsoft, and Google). Uruena et al. [36] manually identify privacy leaks in SSO protocols, which are beyond the scope of the paper, and they propose short-term and long-term solutions. Pfitzmann et al. [29] empirically analyze SSO protocols and identify several vulnerabilities.
- *Automatic Analysis.* Bai et al. [17] introduce AUTHSCAN, a tool that automatically extracts a specification from SSO implementation and identifies vulnerabilities from the extracted specification through formal analysis. Similarly, Explicating SDKs by Wang et al. [38] extracts the underlying assumptions of SSO protocols from their respective SDKs and detects corresponding vulnerability patterns. On the other hand, Armando et al. [16] formally analyze the SAML 2.0 protocol and identify vulnerabilities in the protocol. Lastly, much work based on formal analysis [24, 26–28] has been carried out focusing on the Facebook Connect protocol, SAML, and OAuth respectively.
- *User Studies.* A different categories of vulnerabilities, namely denial of service, such as single point-of-failure related issues, have been highlighted by Sun et al. [34] in an empirical user study to gauge the reaction and opinion on SSO protocols by its respective users. Overall, their study shows that 26% of users express concerns about denial of service attacks on the identity provider, i.e., concerns about attacks preventing them from authenticating to the RP.

7.2 Defense Mechanism

InteGuard [39] describes a proxy in between the client's browser and the IdP. InteGuard extracts a set of invariant relations among HTTP messages it observes and deduces their relation to the security of the protocol. Due to its nature, InteGuard is deployed at the server-side together with a server's load balancer or as process at client's browser. Defensive JavaScript [19] on the other hand is a subset of the JavaScript language that guarantees that scripts integrity is being kept even in an adversarial environment. Alternatively, an OpenIDemail enabled browser [33] modifies the client browser to support OpenID natively and hides the OpenID identifiers from users by using their existing email accounts.

Prior work compares to our protocol as follows:

- *Deployment by the IdP.* All existing approaches can be deployed by the IdP. However, except for Defensive JavaScript [19], they reactively identify and mitigate vulnerabilities in existing and deployed SSO implementations. Instead, our framework proactively mitigates those vulnerabilities from the start by addressing the root cause (analogously to how memory-safe languages address the root cause of memory corruption vulnerabilities). It is important to note that Defensive JavaScript [19] is complementary to our approach. First, it uses a different threat model than we do, i.e., a benign RP might be compromised. In our threat model, however, a benign RP must remain uncompromised. Second, while the communication is secured in our protocol, in the case of Defensive JavaScript the communication between the RP and the IdP remains vulnerable. Thus, a malicious party can sniff or modify messages, as in the case of the NYTimes and Facebook [37] attack.
- *Deployment by non-IdP.* InteGuard [39] can also be deployed by the other entities, essentially acting as a gateway or firewall, thus protecting a set of physical machines with different users behind it. For example, if InteGuard would be deployed at a university, it protects all machines at the university, but a student connecting to a RP from home would not be protected. In contrast, our protocol protects users regardless of where they are connecting from.
  Alternatively, in the case of OpenIDemail, a compatible browser [33] must be used by the user, which is generally considered an impractical burden to the user, and hard to enforce on a large scale. Our design, regardless of its actual deployment scenario, does not require any such modification to the user's browser.

8 Conclusion

In this paper, rather than identifying individual vulnerabilities in SSO protocols, we determined the root cause of why RP impersonation attacks exist: an undedicated, insecure, one-way channel between the RP and the IdP. Based on our findings, we propose to abandon simple HTTP redirection or the raw in-browser communication channel currently used when designing SSO protocols. Instead, a dedicated bi-directional secure channel is needed that can be built on top of existing, in-browser communication channels.

We introduced a technique to establish such a channel securely, proposed a SSO protocol that uses a channel established with our method, and verified the correctness of our protocol formally with ProVerif [6], which was unable to create

an attack scenarios in respect to our threat model. Additionally, we provide an example of a vulnerability introduced into the protocol deliberately, for which ProVerif identified the vulnerability correctly and generated a valid attack scenario.

In addition to our clean-slate and secure SSO protocol design, we detailed a proxy that allows and supports a smooth transition from existing SSO protocols to our new and secure protocol. The design of this proxy is formally verified in respect to our threat model.

Finally, we evaluated the performance and overhead of our SSO protocol implementation. With an overall small latency overhead of about 650ms, and a clear bottleneck in our prototype implementation in the generation of the private and public key that can be remedied with some engineering effort, our new SSO protocol proves to have a clearly acceptable overhead given its strong security guarantees.

Acknowledgment

This material is based upon work supported by the Department of Homeland Security under grant 2009-ST-061-CI0001; the National Science Foundation under grants CNS-0845559, CNS-0905537, CNS-1116967, and CNS-1255546; by the Office of Naval Research through grants N00014-09-1-1042, and N00014-12-1-0165; by the Army Research Office under grant W911NF-09-1-0553; and Secure Business Austria. This work is also sponsored by DARPA under agreement number FA8750-12-2-0101 and N66001-13-2-4039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Getting started with the facebook sdk for php. https://developers.facebook.com/docs/php/gettingstarted/.
2. JanRain Engage. http://janrain.com/products/engage/.
3. JavaScript Cryptography Toolkit. http://ats.oka.nu/titaniumcore/js/crypto/readme.txt.
4. LocalConnection. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/LocalConnection.html.
5. OpenID. http://openid.net/.
6. ProVerif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.
7. RFC 4252: The Secure Shell (SSH) Authentication Protocol. http://tools.ietf.org/html/rfc4252.
8. RFC 4301: Security Architecture for the Internet Protocol. http://tools.ietf.org/html/rfc4301.
9. RFC 5246: The Transport Layer Security (TLS) Protocol. http://tools.ietf.org/html/rfc5246.
10. RFC 6101: The Secure Sockets Layer (SSL) Protocol. http://tools.ietf.org/html/rfc6101.
11. Sandbox mode of facebook application. https://developers.facebook.com/docs/ApplicationSecurity/.
12. Secure electronic transaction. http://goo.gl/2SpMbF.
13. Security assertion markup language. http://en.wikipedia.org/wiki/Security_Assertion_Markup_Language.
14. The Stanford Javascript Crypto Library. http://crypto.stanford.edu/sjcl/.

15. Akhawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a formal foundation of web security. In *CSF* (2010).
16. Armando, A., Carbone, R., Compagna, L., Cuellar, J., and Tobarra, L. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *FMSE: the ACM workshop on Formal methods in security engineering* (2008).
17. Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS* (2013).
18. Barth, A., Jackson, C., and Mitchell, J. C. Securing frame communication in browsers. In *USENIX Security Symposium* (2008).
19. Bhargavan, K., Delignat-Lavaud, A., and Maffeis, S. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium* (2013).
20. Blanchet, B. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW* (2001).
21. Cao, Y., Rastogi, V., Li, Z., Chen, Y., and Moshchuk, A. Redefining web browser principals with a configurable origin policy. In *DSN* (2013).
22. Dolev, D., and Yao, A. C. On the security of public key protocols. Tech. rep., Stanford, CA, USA, 1981.
23. Facebook. Facebook connect. `http://goo.gl/ZUyBXF`.
24. Gro, T. Security Analysis of the SAML Single Sign-on Browser/Artifact Profile. In *ACSAC* (2003).
25. Hanna, S., Shin, R., Akhawe, D., Saxena, P., Boehm, A., and Song, D. The emperor's new APIs: On the (in)secure usage of new client-side primitives. In *W2SP* (2010).
26. Hansen, S. M., Skriver, J., and Nielson, H. R. Using static analysis to validate the saml single sign-on protocol. In *WITS: the workshop on Issues in the theory of security* (2005).
27. Miculan, M., and Urban, C. Formal Analysis of Facebook Connect Single Sign-On Authentication Protocol. In *SOFSEM* (2011).
28. Pai, S., Sharma, Y., Kumar, S., Pai, R. M., and Singh, S. Formal verification of oauth 2.0 using alloy framework. In *CSNT: the International Conference on Communication Systems and Network Technologies* (2011).
29. Pfitzmann, B., and Waidner, M. Analysis of liberty single-sign-on with enabled clients. *Internet Computing, IEEE 7*, 6 (2003), 38–44.
30. Singh, K., Moshchuk, A., Wang, H., and Lee, W. On the Incoherencies in Web Browser Access Control Policies. In *SP: IEEE Symposium on Security and Privacy* (2010).
31. Son, S., and Shmatikov, V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS* (2013).
32. Sun, S.-T., and Beznosov, K. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *CCS* (2012).
33. Sun, S.-T., Hawkey, K., and Beznosov, K. OpenIDemail enabled browser: towards fixing the broken web single sign-on triangle. In *DIM* (2010).
34. Sun, S.-T., Pospisil, E., Muslukhov, I., Dindar, N., Hawkey, K., and Beznosov, K. What makes users refuse web single sign-on?: an empirical investigation of openid. In *SOUPS* (2011).
35. Tassanaviboon, A., and Gong, G. Oauth and abe based authorization in semi-trusted cloud computing: aauth. In *DataCloud-SC: the international workshop on Data intensive computing in the clouds* (2011).
36. Urueña, M., Muñoz, A., and Larrabeiti, D. Analysis of privacy vulnerabilities in single sign-on mechanisms for multimedia websites. *Multimedia Tools and Applications* (2012).
37. Wang, R., Chen, S., and Wang, X. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy* (2012).
38. Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security Symposium* (2013).
39. Xing, L., Chen, Y., Wang, X., and Chen, S. InteGuard: Toward Automatic Protection of Third-party web service integrations. In *NDSS* (2013).
40. Yang, E. Z., Stefan, D., Mitchell, J., Mazieres, D., Marchenko, P., and Karp, B. Toward principled browser security. In *HotOS* (2013).