Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web

Daniel Fett, Ralf Küsters, and Guido Schmitz

University of Trier, Germany
{fett,kuesters,schmitzg}@uni-trier.de

Abstract. BrowserID is a complex, real-world Single Sign-On (SSO) System for web applications recently developed by Mozilla. It employs new HTML5 features (such as web messaging and web storage) and cryptographic assertions to provide decentralized login, with the intent to respect users' privacy. It can operate in a primary and a secondary identity provider mode. While in the primary mode BrowserID runs with arbitrary identity providers, in the secondary mode there is one identity provider only, namely Mozilla's default identity provider.

We recently proposed an expressive general model for the web infrastructure and, based on this web model, analyzed the security of the secondary identity provider mode of BrowserID. The analysis revealed several severe vulnerabilities, which have been fixed by Mozilla.

In this paper, we complement our prior work by analyzing the even more complex primary identity provider mode of BrowserID. We do not only study authentication properties as before, but also privacy properties. During our analysis we discovered new and practical attacks that do not apply to the secondary mode: an identity injection attack, which violates a central authentication property of SSO systems, and attacks that break the privacy promise of BrowserID and which do not seem to be fixable without a major redesign of the system. Interestingly, some of our attacks on privacy make use of a browser side channel that, to the best of our knowledge, has not gained a lot of attention so far.

For the authentication bug, we propose a fix and formally prove in a slight extension of our general web model that the fixed system satisfies all the authentication requirements we consider. This constitutes the most complex formal analysis of a web application based on an expressive model of the web infrastructure so far.

As another contribution, we identify and prove important security properties of generic web features in the extended web model to facilitate future analysis efforts of web standards and web applications.

1 Introduction

Single sign-on (SSO) systems have become an important building block for authentication in the web. Over the last years, many different SSO systems have been developed, for example, OpenID, OAuth, and proprietary solutions such as Facebook Connect. These systems usually allow a user to identify herself to a so-called relying party (RP), which provides some service, using an identity that is managed by an identity provider (IdP), such as Facebook or Google.

Given their role as brokers between IdPs and RPs, the security of SSO systems is particularly crucial: numerous attacks have shown that vulnerabilities in SSO systems compromise the security of many services and users at once (see, e.g., [3,7,23–26]).

BrowserID [21] is a relatively new complex SSO system which allows users to utilize any of their existing email addresses as an identity. BrowserID, which is also known by its marketing name *Persona*, has been developed by Mozilla and provides decentralized and federated login, with the intent to respect users' privacy: While in other SSO systems (such as OpenID), by design, IdPs can always see when and where their users log in, Mozilla's intention behind the design of BrowserID was that such tracking should not be possible. Several web applications support BrowserID authentication. For example, popular content management systems, such as Drupal and WordPress allow users to log in using BrowserID. Also Mozilla uses this SSO system on critical web sites, e.g., their bug tracker Bugzilla and their developer network MDN.

The BrowserID implementation is based solely on native web technologies. It uses many new HTML5 web features, such as web messaging and web storage. For example, BrowserID uses the postMessage mechanism for cross-origin inter-frame communication (i.e., communication within a browser between different windows) and the web storage concept of modern browsers to store user data on the client side.

There are two modes for BrowserID: For the best user experience, email providers (IdPs) can actively support BrowserID; they are then called *primary IdPs*. For all other email providers that do not support BrowserID, the user can register her email address at a default IdP, namely Mozilla's `login.persona.org`, the so-called *secondary IdP*.

In [13], we proposed a general and expressive Dolev-Yao style model for the web infrastructure. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards (mainly RFCs). It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as new technologies, such as web storage and cross-document messaging (postMessages). JavaScript is modeled in an abstract way by so-called scripting processes which can be sent around and, among others, can create iframes and initiate XMLHTTPRequests (XHRs). Browsers may be corrupted dynamically by the adversary.

Based on this general web model, we analyzed the security of the secondary IdP mode of BrowserID [13]. The analysis revealed several severe vulnerabilities, which have since been fixed by Mozilla.

Contributions of this Paper. The main contributions of this paper are that we i) analyze authentication and privacy properties for the primary mode of BrowserID, where in both cases the analysis revealed new attacks, ii) identify generic web security properties to ease future analysis efforts, and iii) slightly extend our web model.

As mentioned before, in [13], we studied the simpler secondary mode of BrowserID only. The primary model studied here is much more complex than the secondary mode (see also the remarks in Section 4.2). It involves more components (such as an arbitrary set of IdPs, more iframes), a much more complex communication structure, and requires weaker trust assumptions (for example, some IdPs, and hence, the JavaScript they deliver, might be malicious). Also, in our previous work, we have not considered privacy properties, but authentication properties only.

More specifically, the contributions of this paper can be summarized as follows.

Extension of the Web Model. We slightly extend our web model proposed in [13]. We complement the modeling of the web storage concept of modern browsers by adding sessionStorage [27], which is (besides the already modeled localStorage) heavily used by BrowserID in its primary mode. We also extend the model to include a set of user identities (e.g., user names or email addresses) in addition to user secrets.

Authentication Attack and Security Proof for BrowserID. The authentication properties we analyze are central to any SSO system and correspond to those considered in our previous work: i) the attacker should not be able to log in at an RP as an honest user and ii) the attacker should not be able to authenticate an honest user/browser to an RP with an ID not owned by the user (identity injection). While trying to prove these authentication properties for the primary mode of BrowserID, we discovered a new attack which violates property ii). Depending on the service provided by the RP, this could allow the attacker to track the honest user or to obtain user secrets. We confirmed the attack on the actual implementation and reported it to Mozilla, who acknowledged the attack. We note that this attack does not apply to the secondary mode.

We propose a fix and provide a detailed formal proof based on the (extended) web model which shows that the fixed system satisfies the mentioned authentication properties. This constitutes the most complex formal analysis of a web application based on an expressive model of the web infrastructure, in fact, as mentioned, the most comprehensive one to date. We note that other web models are too limited to be applied to BrowserID (see also Section 7).

Privacy Attacks on BrowserID. As pointed out before, BrowserID was designed by Mozilla with the explicit intention to respect users' privacy. Unlike in other SSO systems, when using BrowserID, IdPs should not learn to which RP a user logs in. When trying to formally prove this property, we discovered attacks that show that BrowserID cannot live up to this claim. Our attacks allow malicious IdPs to check whether or not a user is logged in at a specific RP with little effort. Interestingly, one variant of these attacks exploits a browser side channel which, to our knowledge, has not received much attention in the literature so far. Just as for authentication, we have confirmed the attacks on the actual implementation and reported them to Mozilla [10], who acknowledged the attacks. We have been awarded a bug bounty from the Mozilla Security Bug Bounty Program. Unfortunately, the attacks exploit a design flaw of BrowserID that does not seem to be easily fixable without a major redesign.

Generic Web Security Properties. Our security analysis of BrowserID and the case study in [13] show that certain security properties of the web model need to be established in most security proofs for web standards and web applications. As another contribution, we therefore identify and summarize central security properties of generic web features in our extension of our model and formalize them in a general way such that they can be used in and facilitate future analysis efforts of web standards and web applications.

Structure of this Paper. In Section 2, we outline the basic communication model and the web model, including our extensions. We deduce general properties of this model, which are independent of specific web applications, in Section 3. For our security analysis, we first, in Section 4, provide a description of the BrowserID system, focusing on the primary mode. We then, in Section 5, present our attack and the formal analysis of the authentication properties of the (fixed) BrowserID system in primary mode. In Section 6, we present our attacks on privacy of BrowserID. Related work is discussed in Section 7. We conclude in Section 8. In the appendix, we present more details on our web model and some privacy attack variants. Full details of our models and proofs can be found in our technical report [14].

2 The Web Model

In this section, we present a brief overview of our model of the web infrastructure as proposed in [13], along with our extensions (sessionStorage and user identities) mentioned in the introduction. Full details are provided in [14]. We first present the generic Dolev-Yao style communication model which the model is based on.

2.1 Communication Model

The main entities in the communication model are *atomic processes*, which will be used to model web browsers, web servers, DNS servers as well as web and network attackers. Each atomic process has a list of addresses (representing IP addresses) it listens to. A set of atomic processes forms what is called a *system*. The different atomic processes in such a system can communicate via events, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from the current "pool" of events and is delivered to an atomic process that listens to the receiver address of that event; if different atomic processes can listen to the same address, the atomic process to which the event is delivered is chosen non-deterministically among the possible processes. The (chosen) atomic process can then process the event and output new events, which are added to the pool of events, and so on. More specifically, messages, processes, etc. are defined as follows.

Terms, Messages and Events. As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature. The signature Σ for the terms and messages considered in our web model contains, among others, constants (such as (IP) addresses, ASCII strings, and nonces), sequence and projection symbols, and further function symbols, including those for (a)symmetric encryption/decryption and digital signatures. The equational theory associated with the signature Σ is defined as usual in Dolev-Yao models. Message are defined to be ground terms (terms without variables) and events are of the form $(a{:}f{:}m)$ where a and f are receiver/sender (IP) addresses, and m is a message.

To provide an example of a message, in our web model an HTTP request is represented as a ground term containing a nonce, a method (e.g., GET or POST), a domain name, a path, URL parameters, request headers (such as Cookie), and a message body. Now, for example, an HTTP GET request for the URL http://ex.com/show?p=1 is modeled as the term $r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{ex.com}, \mathtt{/show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$, where headers and body are empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

Atomic Processes, Systems and Runs. Atomic Dolev-Yao processes, systems, and runs of systems are defined as follows.

An *atomic Dolev-Yao (DY) process* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ where I^p is a set of addresses (the set of addresses the process listens to), Z^p is a set of states (formally, terms), $s_0^p \in Z^p$ is an initial state, and R^p is a relation that takes an event and a state as input and (non-deterministically) returns a new state and a set of events. This relation models a computation step of the process, which upon receiving an event in a given state non-deterministically moves to a new state and outputs a set of messages (events). It is required that the events and states in the output can be computed (more formally, derived in the usual Dolev-Yao style) from the current input event and state.

The so-called *attacker process* is an atomic DY process which records all messages it receives and outputs all messages it can possibly derive from its recorded messages. Hence, an attacker process is the maximally powerful DY process. It carries out all attacks any DY process could possibly perform and is parametrized by the set of sender addresses it may use. Attackers may corrupt other DY processes (e.g., a browser).

A *system* is a (possibly infinite) set of atomic processes. Its state (i.e., the states of all atomic processes in the system) together with a multi-set of waiting events is called a *configuration*.

A *run* of a system for an initial set E_0 of events is a sequence of configurations, where each configuration (except for the first one, which consists of E_0 and the initial states of the atomic processes) is obtained by delivering one of the waiting events of the preceding configuration to an atomic process p (which listens to the receiver address of the event), and which in turn performs a computation step according to its relation R^p.

Scripting Processes. We also define scripting processes, which model client-side scripting technologies, such as JavaScript.

A *scripting process* (or simply, a *script*) is defined similarly to a DY process. It is called by the browser in which it runs. The browser provides it with a (fresh, infinite) set of nonces and state information s. The script then outputs a term s', which represents the new internal state and some command which is interpreted by the browser (see Appendix A). Again, it is required that a script's output is derivable from its input.

Similarly to an attacker process, the so-called *attacker script* R^{att} may output everything that is derivable from the input.

2.2 Web System

A web system formalizes the web infrastructure and web applications. Formally, a *web system* is a tuple $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ with the following components:

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and contains honest processes, web attacker, and network attacker processes. While a web attacker can listen to and send messages from its own addresses only, a network attacker may listen to and spoof all addresses. Hence, it is the maximally powerful attacker. Attackers may corrupt other parties. In the analysis of a concrete web system, we typically have one network attacker only and no web attackers (as they are subsumed by the network attacker), or one or more web attackers but then no network attacker. Honest processes can either be web browsers, web servers, or DNS servers. The modeling of web servers heavily depends on the specific application (for BrowserID see the modeling in Section 5.1). In our security analysis of authentication properties, DNS servers will be subsumed by the attacker, and hence, do not need to be modeled explicitly in this work. The web browser model, which is independent of a specific web application, is presented below.

The second component, $\mathcal{S}$, is a finite set of scripts, including the attacker script R^{att}. In a concrete model, such as our BrowserID model (see Section 5.1), the set $\mathcal{S} \setminus \{R^{\text{att}}\}$ describes the set of honest scripts used in the application under consideration while malicious scripts are modeled by the "worst-case" malicious script, R^{att}.

The third component, script, is an injective mapping from a script in $\mathcal{S}$ to its string representation $\mathsf{script}(s)$ (a constant in Σ). Finally, E_0 is a multi-set of events, containing an infinite number of events of the form $(a{:}a{:}\mathtt{TRIGGER})$ for every process a in the web system. A *run* of the web system is a run of $\mathcal{W}$ initiated by E_0.

2.3 Web Browsers

We now sketch the model of the web browser, with more details provided in Appendix A. A web browser is modeled as a DY process $(I^p, Z^p, R^p, s_0^p, N^p)$.

An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions are modeled as non-deterministic actions of the web browser. For example, the browser itself non-deterministically follows the links in a web page. User data (i.e., passwords and identities) is stored in the initial state of the browser and is given to a web page when needed, similar to the AutoFill feature in browsers.

Besides the user identities and passwords, the state of a web browser (modeled as a term) contains a tree of open windows and documents, lists of cookies, localStorage and sessionStorage data, a DNS server address, and other data (see Appendix A). We note that identities and sessionStorage were not considered in [13].

In the browser state, the *windows* subterm is the most complex one. It contains a window subterm for any open window (which may be many at a time), and inside each window, a list of documents opened in that window (which, again, may contain windows, modeling iframes). A document contains a script loaded from a web server and represents one loaded HTML page.

Scripts may, for example, navigate or create windows, send XHRs and postMessages, submit forms, set/change cookies, localStorage, and sessionStorage data, and create iframes. When activated, the browser provides a script with all data it has access to, such as certain cookies as well as localStorage and sessionStorage.

Browsers can become corrupted, i.e., be taken over by web and network attackers. We model two types of corruption: *close-corruption*, modeling that a browser is closed by the user, and hence, certain data is removed (e.g., session cookies and opened windows), before it is taken over by the attacker, and *full corruption*, where no data is not removed in advance. Once corrupted, the browser behaves like an attacker process.

3 General Security Properties

We have identified central application independent security properties of web features in the web model and formalized them in a general way such that they can be used in and facilitate future analysis efforts of web standards and web applications. In this section, we provide a brief overview of these properties, with precise formulations and proofs presented in [14].

The first set of properties concerns encrypted connections (HTTPS): We show that HTTP requests that were encrypted by an honest browser for an honest receiver cannot be read or altered by the attacker (or any other party). This, in particular, implies correct behavior on the browser's side, i.e., that browsers that are not fully corrupted never leak a symmetric key used for an HTTPS connection to any other party. We also show that honest browsers set the host header in their requests properly, i.e., the header reflects an actual domain name of the receiver, and that only the designated receiver can successfully respond to HTTPS requests.

The second set of properties concerns origins and origin headers. Using the properties stated above, we show that browsers cannot be fooled about the origin of an (HTTPS) document in their state: If the origin of a document in the browser's state is a secure origin (HTTPS), then the document was actually sent by that origin. Moreover, for requests which contain an origin header with a secure origin we prove that such requests were actually initiated by a script that was sent by that origin to the browser. In other words, in this case, the origin header works as expected.

4 The BrowserID System

BrowserID [22] is a decentralized single sign-on (SSO) system developed by Mozilla for user authentication on web sites. It is a complex full-fledged web application deployed in practice, with currently ~47k LOC (excluding some libraries). It allows web sites to delegate user authentication to email providers, identifying users by their email addresses. BrowserID makes use of a broad variety of browser features, such as XHRs, postMessage, local- and sessionStorage, cookies, various headers, etc.

We first, in Section 4.1, provide a high-level overview of the BrowserID system. A more detailed description of the BrowserID implementation is then given in Section 4.2. The description of the BrowserID system presented in the following as well as our BrowserID model (see Section 5.1) is extracted mainly from the BrowserID source code [20] and the (very high-level) official BrowserID documentation [22].

4.1 Overview

The BrowserID system knows three distinct parties: the user, who wants to authenticate herself using a browser, the relying party (RP) to which the user wants to authenticate (log in) with one of her email addresses (say, `user@idp.com`), and the identity/email address provider, the IdP. If the IdP (`idp.com`) supports BrowserID directly, it is called a *primary IdP*. Otherwise, a Mozilla-provided service, the so-called *secondary IdP*, takes the role of the IdP. As mentioned before, here we concentrate on the primary IdP mode as the secondary IdP mode was described in detail in [13]. However, we briefly discuss the differences between the two modes at the end of Section 4.2.

A primary IdP provides information about its setup in a so-called *support document*, which it provides at a fixed URL derivable from the email domain, e.g., `https://idp.com/.well-known/browserid`.

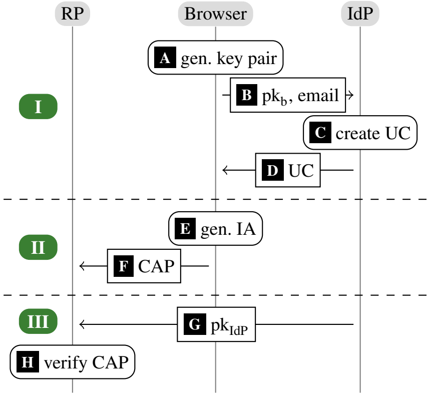


Fig. 1. BrowserID login: basic overview

A user who wants to log in at an RP with an email address for some IdP has to present two signed documents to the RP: A *user certificate* (UC) and an *identity assertion* (IA). The UC contains the user's email address and the user's public key. It is signed by the IdP. The IA contains the origin of the RP and is signed with the user's private key. Both documents have a limited validity period. A pair consisting of a UC and a matching IA is called a *certificate assertion pair* (CAP) or a *backed identity assertion*. Intuitively, the UC in the CAP tells the RP that (the IdP certified that) the owner of the email address is (or at least claims to be) the owner of the public key. By the IA contained in the CAP the RP is ensured that the owner of the given public key (i.e., the one who knows the corresponding private key) wants to log in. Altogether, given a valid CAP, RP would consider the user (identified by the email address in the CAP) to be logged in.

The BrowserID authentication process (with a primary IdP) consists of three phases (see Figure 1): $\boxed{\text{I}}$ UC provisioning, $\boxed{\text{II}}$ CAP creation, and $\boxed{\text{III}}$ CAP verification.

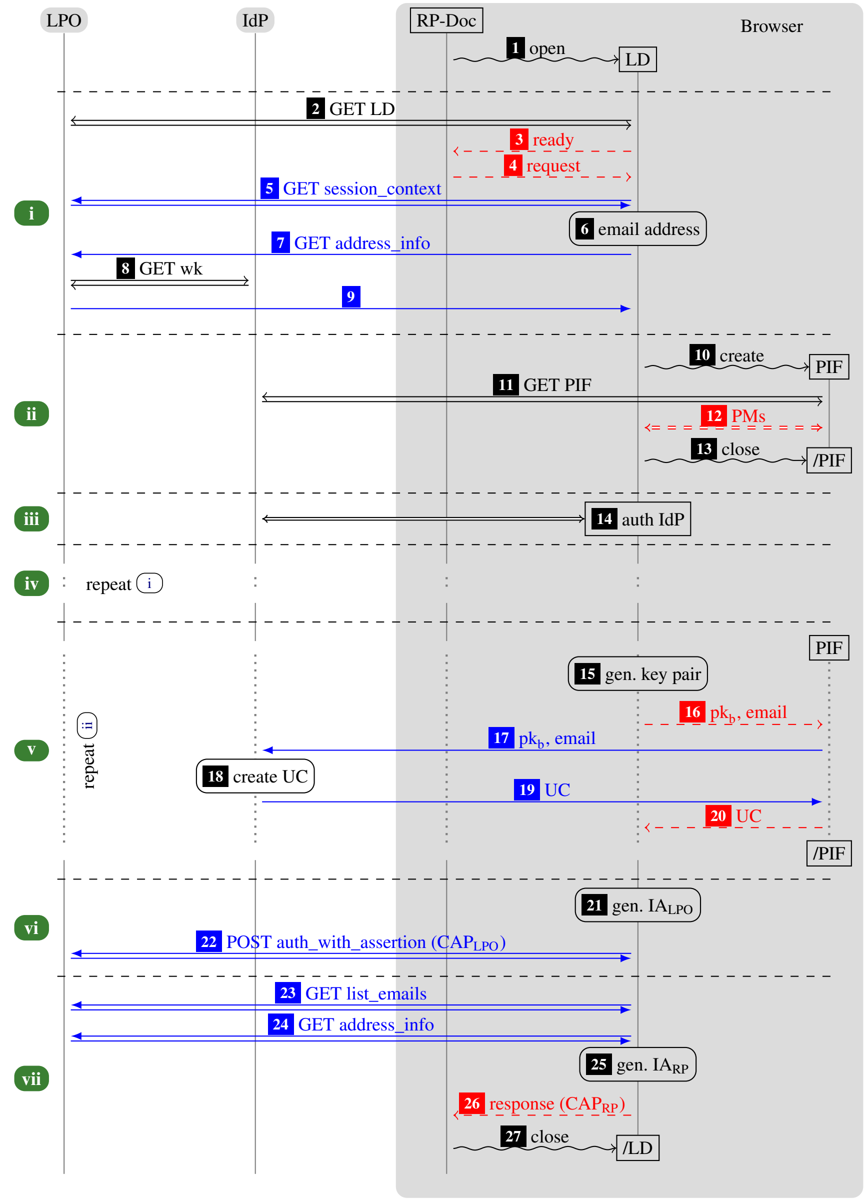
In Phase $\boxed{\text{I}}$, (the browser of) the user creates a public/private key pair $\boxed{\text{A}}$. She then sends her public key as well as the email address she wants to use to log in at some RP to the respective IdP $\boxed{\text{B}}$. The IdP now creates the UC $\boxed{\text{C}}$, which is then sent to the user $\boxed{\text{D}}$. The above requires the user to be logged in at IdP.

With the user having received the UC, Phase $\boxed{\text{II}}$ can start. The user wants to authenticate to an RP, so she creates the IA $\boxed{\text{E}}$. The UC and the IA are concatenated to a CAP, which is then sent to the RP $\boxed{\text{F}}$.

In Phase $\boxed{\text{III}}$, the RP checks the authenticity of the CAP. For this purpose, the RP fetches the public key of the IdP $\boxed{\text{G}}$, which is contained in the support document. Afterwards, the RP checks the signatures of the UC and the IA $\boxed{\text{H}}$. If this check is successful, the RP can, as mentioned before, consider the user to be logged in with the given email address and send her some token (e.g., a cookie with a session ID), which we refer to as an *RP service token*.

4.2 Implementation Details

We now provide a more detailed description of the BrowserID implementation. Since the system is very complex, with many HTTPS requests, XHRs, and postMessages sent between different entities (servers as well as windows and iframes within the browser), we here describe mainly the phases of the login process without explaining every single message exchange done in the implementation. A more detailed step-by-step description can be found in [14]. Note that BrowserID's specification of IdPs fixes the interface to BrowserID only, but otherwise does not further detail the specification of IdPs. Therefore, in what follows, we consider a typical IdP, namely the example implementation provided by Mozilla [20].



⟶ HTTPS messages, ⟶ XHRs (over HTTPS), - -> postMessages, ⤳ browser commands

Fig. 2. Simplified BrowserID implementation overview. CIF omitted for brevity.

In addition to the parties mentioned so far, the actual BrowserID implementation uses another party, Mozilla's `login.persona.org` (LPO). Among others, LPO provides HTML and JavaScript files that, for security and privacy reasons, cannot be delivered by either IdP or RP. An overview of the implementation is given in Figure 2. For brevity of presentation, several messages and components, such as the CIF (see below), are omitted in the figure (but not in our analysis).

Windows and iframes in the Browser. By *RP-Doc* we denote the window containing the document loaded from some RP, at which the user wants to log in with an email address hosted by some IdP. RP-Doc typically includes JavaScript from LPO and contains a button "Login with BrowserID". The LPO JavaScript running in RP-Doc opens an auxiliary window called the *login dialog* (LD). Its content is provided by LPO and it handles the interaction with the user. During the login process, a temporary invisible iframe called the *provisioning iframe* (PIF) can be created in the LD. The PIF is loaded from IdP. It is used by LD to communicate (cross-origin) with the IdP via postMessages: As the BrowserID implementation mainly runs under the origin of LPO, it cannot directly communicate with the IdP, thus it uses the PIF as a proxy. Temporarily, the LD may navigate itself to a web page at IdP to allow for direct user interaction with the IdP. We then call this window the *authentication dialog* (AD).

Login Process. To describe the login process, for the sake of presentation we assume for now that the user uses a "fresh" browser, i.e., the user has not been logged in before. As mentioned, the process starts by the user visiting a web site of some RP. After the user has clicked on the login button in RP-Doc, the LD is opened and the interactive login flow is started. We can divide this login flow into seven phases: In Phase $\boxed{\text{i}}$, the LD is initialized and the user is prompted to provide her email address. Also, LD fetches the support document (see Section 4.1) of the IdP via LPO. In Phase $\boxed{\text{ii}}$, LD creates the PIF from the *provisioning URL* provided in the support document. As (by our assumption) the user is not logged in yet, the PIF notifies LD that the user is not authenticated to the IdP. In Phase $\boxed{\text{iii}}$, LD navigates itself away to the *authentication URL* which is also provided in the support document and links to the IdP. Usually, this document will show a login form in which the user enters her password to authenticate to the IdP. After the user has been authenticated to IdP (which typically implies that the IdP sets a session cookie in the browser), the window is navigated back to LPO.

Now, the login flow continues in Phase $\boxed{\text{iv}}$, which basically repeats Phase $\boxed{\text{i}}$. However, the user is not prompted for her email address (it has previously been saved in the localStorage under the origin of LPO along with a nonce, where the nonce is stored in the sessionStorage). In Phase $\boxed{\text{v}}$, which essentially repeats Phase $\boxed{\text{ii}}$, the PIF detects that the user is now authenticated to the IdP and the provisioning phase is started ($\boxed{\text{I}}$ in Figure 1): The user's keys are created by LD and stored in the localStorage under the origin of LPO. The PIF forwards the certification request to the IdP, which then creates the UC and sends it back to the PIF. The PIF in turn forwards it to the LD, which stores it in the localStorage under the origin of LPO.

In Phases $\boxed{\text{vi}}$ and $\boxed{\text{vii}}$, mainly the IA is generated by LD for the origin of RP-Doc and sent (together with the UC) to RP-Doc ($\boxed{\text{II}}$ in Figure 1). In the localStorage, LD stores that the user's email address is logged in at RP. Moreover, to log the user in at LPO, LD generates an IA for the origin of LPO and sends the UC and IA to LPO.

Automatic CAP Creation. In addition to the interactive login presented above, BrowserID also contains an automatic, non-interactive way for RPs to obtain a freshly generated CAP: During initialization within RP-Doc, an invisible iframe called the *communication iframe* (CIF) is created inside RP-Doc. The CIF's JavaScript is loaded from LPO and behaves similar to LD, but without user interaction. The CIF automatically issues a fresh CAP and sends it to RP-Doc under specific conditions: among others, the email address must be marked as logged in at RP in the localStorage. If necessary, a new key pair is created and a corresponding new UC is requested at the IdP. For this purpose, a PIF is created inside the CIF.

Differences to the Secondary IdP Mode. In the secondary IdP mode there are three parties involved only: RP, Browser, and LPO, where LPO also takes the role of an IdP; LPO is the only IdP that is present, rather than an arbitrary set of (external) IdPs. Consequently, in the secondary IdP mode the PIF and the AD do not exist. Moreover, in the primary mode, the behavior of the CIF and the LD is more complex than in the secondary mode. For example, in the primary mode, just like the LD, the CIF might contain a PIF (iframe in iframe) and interact with it via postMessages. Altogether, the secondary IdP case requires much less communication between parties/components and trust assumptions are simpler: in the secondary IdP mode LPO (which is the only IdP in this mode) has to be trusted, in the primary IdP mode some external IdPs might be malicious (and hence, also the scripts they deliver for the PIF and the AD). In [14], Appendix I, we illustrate the differences between the two modes.

5 Analysis of BrowserID: Authentication Properties

In this section, we present the analysis of the BrowserID system with primary IdPs and with respect to authentication properties. As already mentioned, in [13], we analyzed the simpler case with a secondary IdP. Due to the many differences between the secondary and primary mode as described above, the model for the primary case had to be written from scratch in most parts, and hence, the proof is new and much more complex.

We first, in Section 5.1, describe our model of BrowserID with primary IdPs, with two central authentication properties one would expect any SSO system to satisfy formalized in Section 5.2. As mentioned in the introduction, during the analysis of BrowserID it turned out that one of the security properties is not satisfied and that in fact there is an attack on BrowserID. We confirmed that this attack, which was acknowledged by Mozilla, works on the actual implementation of BrowserID. In Section 5.3, the attack is presented along with a fix. In Section 5.4, we prove that the fixed BrowserID system with primary IdPs satisfies both authentication properties.

5.1 Modeling of BrowserID with Primary IdPs

We model the BrowserID system with primary IdPs as a web system (in the sense of Section 2). Note that, while in Section 4 we give only a brief overview of the BrowserID system, our modeling and analysis considers the complete system with primary IdPs, where we have extracted the model from the BrowserID source code [20].

We call a web system $\mathcal{BID} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ a *BrowserID web system* if it is of the form precisely described in [14] and briefly outlined here.

The system $\mathcal{W}$ consists of the (network) attacker process $\mathsf{attacker}$, a finite set B of (initially honest) web browsers, the web server for LPO, a finite set RP of web servers for the relying parties, and a finite set IDP of web servers for the identity providers. (DNS servers are assumed to be dishonest, and hence, are subsumed by $\mathsf{attacker}$.) IdPs and RPs are initially honest and can become corrupted (similar to browsers, by a special message); LPO is assumed to be honest. The definition of the processes in $\mathcal{W}$ follows the description in Section 4.2. For RP, we explicitly follow the security considerations in [22] (Cross-site Request Forgery protection, e.g., by checking origin headers and HTTPS only with STS enabled). When RP receives a valid CAP, RP responds with a fresh *RP service token for ID i* where *i* is the ID (email address) for which the CAP was issued. Intuitively, a client having such a token can use the service of the RP.

The set $\mathcal{S}$ of $\mathcal{BID}$ contains six scripts, with their string representations defined by script: the honest scripts running in RP-Doc, CIF, LD, AD, and PIF, respectively, and the malicious script R^{att}. The scripts for CIF and LD (issued by LPO) are defined in a straightforward way following the implementation outlined in Section 4. The scripts for RP-Doc, AD, and PIF follow the example implementation provided by Mozilla [20].

5.2 Authentication Properties of the BrowserID System

While the documentation of BrowserID does not contain explicit security goals, here we state two fundamental authentication properties every SSO system should satisfy. These properties are adapted from [13].

Informally, these properties can be stated as follows: **(A)** *The attacker should not be able to use a service of RP as an honest user.* In other words, the attacker should not get hold of (be able to derive from his current knowledge) an RP service token for an ID of an honest user (browser), even if the browser was closed and then later used by a malicious user (i.e., after a CLOSECORRUPT). **(B)** *The attacker should not be able to authenticate an honest browser to an RP with an ID that is not owned by the browser (identity injection).* We refer the reader to [14] for the formal definitions.

We call a BrowserID web system *$\mathcal{BID}$ secure (w.r.t. authentication)* if the above conditions are satisfied in all runs of the system.

5.3 Identity Injection Attack on BrowserID with Primary IdPs

While trying to prove the above mentioned authentication properties of BrowserID with primary IdPs in our model, we discovered a serious attack, which is sketched below and does not apply to the case with secondary IdPs. We confirmed the attack on the actual implementation and reported it to Mozilla [9], who acknowledged it.

During the provisioning phase $\boxed{\text{v}}$ (see Figure 2), the IdP issues a UC for the user's identity and public key provided in $\boxed{16}$. This UC is sent to the LD by the PIF in $\boxed{20}$.

If the IdP is malicious, it can issue a UC with different data. In particular, it could replace the email address by a different one, but keep the original public key. This (malicious) UC is then later included in the CAP by LD. The CAP will still be valid, because the public key is unchanged. Now, as the RP determines the user's identity by the UC contained in the CAP, RP issues a service token for the spoofed email address. As a result, the honest user will use RP's service (and typically will be logged in to RP) under an ID that belongs to the attacker, which, for example, could allow the attacker to track actions of the honest user or obtain user secrets. This violates Condition **(B)**.

To fix this problem, upon receipt of the UC in $\boxed{20}$, LD should check whether it contains the correct email address and public key, i.e., the one requested by LD in $\boxed{16}$.

The same is true for the CIF, which behaves similarly to the LD. Our formal model of BrowserID presented in [14] contains these fixes.

5.4 Security of the Fixed System

For the fixed BrowserID system with primary IdPs, we have proven the following theorem, which says that a fixed BrowserID web system (i.e., the system where the above described fix is applied) satisfies the security properties **(A)** and **(B)**.

Theorem 1. *Let $\mathcal{BID}$ be a fixed BrowserID web system. Then, $\mathcal{BID}$ is secure (w.r.t. authentication).*

We prove Conditions **(A)** and **(B)** separately. For both conditions, we assume that they are not satisfied and lead this to a contradiction. In our proofs, we make use of the general security properties of the web model presented in Section 3, which helped a lot in making the proof for the primary IdP model more modular and concise. The complete proof with all details is provided in [14].

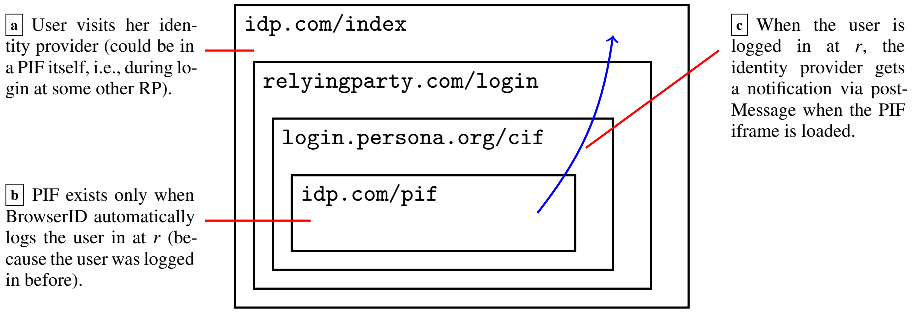
6 Privacy of BrowserID

In this section, we study the privacy guarantees of the BrowserID system with primary IdPs. Regarding privacy, Mozilla states that "…the BrowserID protocol never leaks tracking information back to the Identity Provider." [5] and "Unlike other sign-in systems, BrowserID does not leak information back to any server […] about which sites a user visits." [19].[1] While this is not a formal definition of the level of privacy that BrowserID is supposed to provide, these and other statements[2] make it certainly clear that, unlike for other SSO systems, IdPs should not be able to learn to which RPs their users log in.

In the process of formalizing this intuition in our model of BrowserID and trying to prove this property, we found severe attacks against the privacy of BrowserID which made clear that BrowserID does not provide even a rather weak privacy property in the presence of a malicious IdP. Intuitively, the property says that a malicious IdP (which acts as a web attacker) should not be able to tell whether a user logs in at an honest RP r or some other honest RP r'. In other words, a run in which the user logs in at r at some point should be indistinguishable (from the point of view of the IdP) from the run in which the user logs in at r' instead. Indistinguishability means that the two sequences of messages received by the web attacker in the two runs are statically equivalent in the usual sense of Dolev-Yao models (see [1]), i.e., a Dolev-Yao attacker cannot distinguish between the two sequences. Details of the privacy definition are not important here since our attacks clearly show that privacy is broken for any reasonable definition of privacy. Unfortunately, our attacks are not caused by a simple implementation error,

[1] Clearly, in the current state of BrowserID a malicious LPO server could gather information about users' log in history. However, an integration of the code currently delivered by LPO into the browser, as envisioned, would avoid this issue. Currently, Mozilla's LPO needs to be trusted.

[2] see, for example, `https://developer.mozilla.org/en-US/Persona/Why_Persona` and `http://identity.mozilla.com/post/7669886219`.



$\boxed{\text{a}}$ User visits her identity provider (could be in a PIF itself, i.e., during login at some other RP).

$\boxed{\text{b}}$ PIF exists only when BrowserID automatically logs the user in at *r* (because the user was logged in before).

$\boxed{\text{c}}$ When the user is logged in at *r*, the identity provider gets a notification via postMessage when the PIF iframe is loaded.

Fig. 3. The three main steps of the privacy attack. Using a specially crafted PIF document, a malicious IdP can notify itself via postMessage when the user is logged in at some RP *r*.

but rather a fundamental design flaw in the BrowserID protocol. Fixes for this flaw are conceivable, but not without major changes to the design of BrowserID as discussed in Section 6.2. Such a redesign of BrowserID and a proof of privacy of the redesigned system are therefore out of the scope of this paper, which focuses on the existing and deployed version of BrowserID.

6.1 Privacy Attacks on BrowserID

For our attacks to work, it suffices that the IdP is a web attacker. They work even if all DNS servers, RPs, and LPO are honest, and all parties use encrypted connections. In what follows, we present two variants of the attacks on privacy with three additional interesting variants presented in Appendix B.

PostMessage-Based Attack. The adversary is a malicious IdP that is interested to learn whether a user is logged in at RP *r*. Figure 3 illustrates the main steps:

Step $\boxed{\text{a}}$. First, the victim visits her IdP. In BrowserID, email providers serve as IdPs, and therefore it is not unlikely that a user visits this web site (e.g., for checking email). As the IdP usually has some cookie set at the user's browser, it learns the identity of the victim. The IdP now creates a hidden iframe containing the login page of *r*.

Step $\boxed{\text{b}}$. The login page of *r* (now loaded as an iframe within IdP's web site) includes and runs the BrowserID script. As defined in the BrowserID protocol, the script creates the communication iframe (see "Automatic CAP Creation" in Section 4.2), which in turn checks whether the email address is marked as logged in at *r* in the localStorage of the user's browser. Only then it will try to create a new CAP, for which it needs a PIF (the same as in Phase $\boxed{\text{ii}}$ in Figure 2).

Step $\boxed{\text{c}}$. The PIF is loaded from the IdP. (From this action alone, the IdP does not learn where the user wants to log in.) However, instead of the original (honest) PIF document, the IdP can send a modified one that sends a postMessage to the parent of the parent of the parent of its own window, which in this setting is the IdP document that was opened by the user in Step $\boxed{\text{a}}$. When the IdP receives this message in the document from Step $\boxed{\text{a}}$, it knows that the PIF was loaded, and therefore, that the user is currently logged in at *r*.

Note that the IdP can repeatedly apply the above as long as the user stays on the IdP's web site. During this period, the IdP can see whether or not the user is logged in at the targeted RP. Clearly, the IdP can simultaneously run the attack for different RPs. In particular, the IdP can distinguish whether a user is logged in at RP r or r', which violates the privacy property sketched above. In our formal model, the malicious IdP would run the attacker script R^{att} in `idp.com/index` and in `idp.com/pif` (see Figure 3) in order to carry out the attack.

Variant 1: Waiting for UC requests. The IdP first acts as in Step $\boxed{\text{a}}$. Now, it could passively wait for incoming requests for the PIF document or UC requests on its server, which tell the IdP that a provisioning flow (probably initiated by Step $\boxed{\text{a}}$) was started. This variant cannot be executed in parallel and is less reliable in practice, though.

We verified (all variants of) the attacks in our model as well as in a real-world BrowserID setup. Implementing proofs-of-concept required only a few lines of (trivial) JavaScript. In most attack variants, we directly or indirectly use the structure of the windows inside the web browser as a side channel. To our knowledge, this is the first description of this side channel for breaking privacy in browsers. The attacks have been reported to and confirmed by Mozilla [10].

6.2 Fixing the Privacy of BrowserID

Fixing the privacy of BrowserID seems to require a substantial redesign of the system. Regarding the presented attacks, BrowserID's main weakness is the window structure. The most obvious mitigation, modifying the CIF such that it always creates the PIF (even if the user has not logged in before), does not work: To open the PIF, the CIF looks up (in the localStorage) the user's identity at the current RP to derive the address of the PIF. If the user has not logged in before, this information is not available.

Another approach would be to use cross-origin XHRs to replace the features of the PIF. This solution would require a major revision in the inner workings of BrowserID and would not protect against Variant 1.

7 Related Work

The formal treatment of the security of the web infrastructure and web applications based on this infrastructure is a young discipline. Of the few works in this area even less are based on a general model that incorporates essential mechanisms of the web.

Early works in formal web security analysis (see, e.g., [3,11,16,17,25]) are based on very limited models developed specifically for the application under scrutiny. The first work to consider a general model of the web, written in the finite-state model checker Alloy, is the work by Akhawe et al. [2]. Inspired by this work, Bansal et al. [6,7] built a more expressive model, called WebSpi, in ProVerif [8], a tool for symbolic cryptographic protocol analysis. These models have successfully been applied to web standards and applications. Recently, Kumar [18] presented a high-level Alloy model and applied it to SAML single sign-on. However, compared to our model in [13] and its extensions considered here, on the one hand, all above mentioned models are formulated in the specification languages of specific analysis tools, and hence, are tailored towards automation (while we perform manual analysis). On the other hand, the models considered in these works are much less expressive and precise. For example, these models do not incorporate a precise handling of windows, documents, or iframes; cross-document messaging (postMessages) or session storage are not included at all. In fact, several general web features and technologies that have been crucial for the analysis of BrowserID are not supported by these models, and hence, these models cannot be applied to BrowserID. Moreover, the complexity of BrowserID exceeds that of the systems analyzed in these other works in terms of the use of web technologies and the complexity of the protocols. For example, BrowserID in primary mode is a protocol consisting of 48 different (network and inter-frame) messages compared to typically about 10–15 in the protocols analyzed in other models.

The BrowserID system in the primary mode has been analyzed before using the AuthScan tool developed by Bai et al. [4]. Their work focusses on the automated extraction of a model from a protocol implementation. This tool-based analysis did not reveal the identity injection attack, though; privacy properties have not been studied there. Dietz and Wallach demonstrated a technique to secure BrowserID when specific flaws in TLS are considered [12].

8 Conclusion

In this paper, we slightly extended our existing web model, resulting in the most comprehensive model of the web so far. It contains many security-relevant features and is designed to closely mimic standards and specifications for the web. As such, it constitutes a solid basis for the analysis of a broad range of web standards and applications.

Based on this model, we presented a detailed analysis of the BrowserID SSO system in the primary IdP mode. During the security proof of the fundamental authentication requirements **(A)** and **(B)**, we found a flaw in BrowserID that does not apply to its secondary mode and leads to an identity injection attack, and hence, violates property **(B)**. We confirmed the attack on the actual BrowserID implementation and reported it to Mozilla, who acknowledged it. We proposed a fix and formally proved that the fixed system fulfills both **(A)** and **(B)**. Among the so far very few efforts on formally analyzing web applications and standards in expressive web models, our analysis constitutes the most complex formal analysis of a web application to date. It illustrates that (manual) security analysis of complex real-world web applications in a detailed web model, while laborious, is feasible and yields meaningful and practically relevant results.

During an attempt to formally analyze the privacy promise of the BrowserID system, we again found practical attacks. These attacks have been reported to and confirmed by Mozilla and, unfortunately, show that BrowserID would have to undergo a substantial redesign in order to fulfill its privacy promise. Interestingly, for our attacks we use a side channel that exploits information about the structure of windows in a browser. To the best of our knowledge, such side channel attacks have not gained much attention so far in the literature.

Finally, we have identified and proven important security properties of general application independent web features in order to facilitate future analysis efforts of web standards and web applications in the web model.

References

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL 2001*, pages 104–115. ACM Press, 2001.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF 2010*, pp. 290–304. IEEE Computer Society, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In *FMSE 2008*, pp. 1–10. ACM, 2008.

[4] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS'13*. The Internet Society, 2013.

[5] W. Bamberg et al. Persona FAQ. Mozilla Developer Network Wiki. Last edited Sept. 29, 2013. https://developer.mozilla.org/en-US/Persona/FAQ.

[6] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *POST 2013*, vol. 7796 of *LNCS*, pp. 126–146. Springer, 2013.

[7] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF 2012*, pp. 247–262. IEEE Computer Society, 2012.

[8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW-14*, pp. 82–96. IEEE Computer Society, 2001.

[9] Bugzilla@Mozilla. Bug 1064254 - Identity Injection Attack on Persona by Malicious IdP, September 2014. https://bugzilla.mozilla.org/show_bug.cgi?id=1064254 (access restricted).

[10] Bugzilla@Mozilla. Bug 1120255 - Privacy leak in Persona, January 2015. https://bugzilla.mozilla.org/show_bug.cgi?id=1120255 (access restricted).

[11] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[12] M. Dietz and D. S. Wallach. Hardening Persona – Improving Federated Web Login. In *NDSS 2014*. The Internet Society, 2014.

[13] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *S&P 2014*, pp. 673–688. IEEE Computer Society, 2014.

[14] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. Technical Report arXiv:1411.7210, arXiv, 2014. Available at http://arxiv.org/abs/1411.7210.

[15] HTML5, W3C Recommendation. Oct. 28, 2014.

[16] D. Jackson. Alloy: A New Technology for Software Modelling. In *TACAS 2002*, vol. 2280 of *LNCS*, p. 20. Springer, 2002.

[17] F. Kerschbaum. Simple Cross-Site Attack Prevention. In *SecureComm 2007*, pp. 464–472. IEEE Computer Society, 2007.

[18] A. Kumar. A Lightweight Formal Approach for Analyzing Security of Web Protocols. In *RAID 2014*, vol. 8688 of *LNCS*, pp. 192–211. Springer, 2014.

[19] C. Mills. Introducing BrowserID: A better way to sign in. Identity at Mozilla. Jul. 14, 2011. http://identity.mozilla.com/post/7616727542/.

[20] Mozilla Identity Team. BrowserID Source Code. BrowserID Repository. https://github.com/mozilla/browserid.

[21] Mozilla Identity Team. Persona. https://login.persona.org.

[22] Mozilla Identity Team. Persona. Mozilla Developer Network. Last visited October 15, 2014. https://developer.mozilla.org/en/docs/persona.

[23] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *USENIX Security '12*, pp. 397–412. USENIX Association, 2012.

[24] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *CCS'12*, pp. 378–390. ACM, 2012.

[25] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically Breaking and Fixing OpenID Security: Formal Analysis, Semi-Automated Empirical Evaluation, and Practical Countermeasures. *Computers & Security*, 31(4):465–483, 2012.

[26] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *S&P 2012*, pp. 365–379. IEEE Computer Society, 2012.

[27] Web Storage - W3C Recommendation 30 July 2013. http://www.w3.org/TR/webstorage/.

A Browser Model

Here, we provide a compact overview of our browser model, with full details presented in [14]. A web browser p is modeled as a DY process $(I^p, Z^p, R^p, s_0^p, N^p)$ where I^p is a finite set of (IP) addresses p may listen to and N^p is an infinite set of nonces p may use. The set of states Z^p, the initial state s_0^p, and the relation R^p are sketched next.

A.1 Browser State: Z^p and s_0^p.

The set Z^p of states of a browser consists of terms of the form

$$\langle \mathit{windows}, \mathit{ids}, \mathit{secrets}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{keyMapping},$$
$$\mathit{sts}, \mathit{DNSaddress}, \mathit{nonces}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{isCorrupted}\rangle.$$

Windows and documents. The most important part of the state are windows and documents, both stored in the subterm *windows*. A browser may have several windows open at any time (resembling the tabs and windows in a real browser), each containing a list of documents (the history of visited web pages) of which one is "active", namely the one currently presented to the user in that window. A window may be navigated forward and backward (modeling navigation buttons), deactivating one document and activating its successor or predecessor. Intuitively, a document represents a loaded HTML page. More formally, a document contains (the string representation of) a script, which is meant to model both the static HTML code (e.g., links and forms) as well as JavaScript code. When called by the browser, a script outputs a command which is then interpreted by the browser, such as following a link or issuing an XHR (see below). Documents may also contain iframes, which are represented as windows (*subwindows*) nested inside of document terms. This creates a tree of windows and documents.

Secrets and IDs. This subterm holds the secrets and the identities of the user of the web browser. Secrets (such as passwords) are modeled as nonces and they are indexed by origins (where an origin is a domain name plus the information whether the connection to this domain is via HTTP or HTTPS). Secrets are only released to documents (scripts) with the corresponding origin, similarly to the AutoFill mechanism in browsers. Identities are arbitrary terms that model public information of the user's identity, such as

```
PROCESSING INPUT MESSAGE $m$
$m =$ FULLCORRUPT: $isCorrupted :=$ FULLCORRUPT
$m =$ CLOSECORRUPT: $isCorrupted :=$ CLOSECORRUPT
$m =$ TRIGGER: non-deterministically choose $action$ from $\{1,2\}$
    $action = 1$: Call script of some active document. Outputs new state and $command$.
        $command =$ HREF: $\rightarrow$ Initiate request
        $command =$ IFRAME: Create subwindow, $\rightarrow$ Initiate request
        $command =$ FORM: $\rightarrow$ Initiate request
        $command =$ SETSCRIPT: Change script in given document.
        $command =$ SETSCRIPTSTATE: Change state of script in given document.
        $command =$ XMLHTTPREQUEST: $\rightarrow$ Initiate request
        $command =$ BACK or FORWARD: Navigate given window.
        $command =$ CLOSE: Close given window.
        $command =$ POSTMESSAGE: Send postMessage to specified document.
    $action = 2$: $\rightarrow$ Initiate request to some URL in new window
$m =$ DNS response: send corresponding HTTP request
$m =$ HTTP(S) response: (decrypt,) find reference.
    reference to window: create document in window
    reference to document: add response body to document's script input
```

Fig. 4. The basic structure of the web browser relation R^p with an extract of the most important processing steps, in the case that the browser is not already corrupted.

email addresses. Identities are released to any origin. As mentioned in the introduction, identities were not considered in [13].

Cookies, localStorage, and sessionStorage. These subterms contain the cookies (indexed by domains), localStorage data (indexed by origins), and sessionStorage data (indexed by origins and top-level window references) stored in the browser. As mentioned in the introduction, sessionStorage was not modeled in [13].

KeyMapping. This term is the equivalent to a certificate authority (CA) certificate store in the browser. Since, for simplicity, the model currently does not formalize CAs, this term simply encodes a mapping assigning domains to their respective public keys.

STS. Domains that are listed in this term are contacted by the web browser over HTTPS only. Connection attempts over HTTP are transparently rewritten to HTTPS. Servers can employ the `Strict-Transport-Security` header to add their domain to this list.

DNSaddress. This term defines the address of the DNS server used by the browser.

Nonces, pendingDNS, and pendingRequests. These terms are used for bookkeeping purposes, recording the nonces that have been used by the browser so far, the HTTP(S) requests that await successful DNS resolution, and HTTP(S) requests that await a response, respectively.

IsCorrupted. This term indicates whether the browser is corrupted ($\neq \perp$) or not ($= \perp$). A corrupted browser behaves like a web attacker.

Initial state s_0^p. In the browser's initial state, *keyMapping*, *DNSAddress*, *secrets*, and *ids* are defined as needed, *isCorrupted* is set to $\perp$, and all other subterms are $\langle\rangle$.

A.2 Web Browser Relation R^p.

This relation, outlined in Figure 4, specifies how the web browser processes incoming messages. The browser may receive special messages that cause it to become corrupted (first two lines in Figure 4), in which case it acts like the attacker process. As explained in Section 2.3, there are two types of corruption: close-corruption and full corruption.

If the browser receives a special trigger message `TRIGGER`, it non-deterministically chooses one of two actions: i) Select one of the current documents, trigger its JavaScript, and evaluate the output of the script. Scripts can change the state of the browser (e.g., by setting cookies) and can trigger specific actions (e.g., following a link or creating an iframe), which are modeled as *commands* issued by the script (see the list in Figure 4). ii) Follow some URL, with the intuition that it was entered by the user.

As mentioned, some of the above actions can cause the browser to generate new HTTP(S) requests. In this case, the browser first asks the configured DNS server for the IP address belonging to the domain name in the HTTP(S) request. As soon as the DNS response arrives, the browser sends the HTTP(S) request to the respective IP address.

If the HTTP(S) response arrives, its headers are evaluated and the body of the request becomes the script of a newly created document that is then inserted at an appropriate place in the window/document tree. However, if the HTTP(S) response is a response to an XHR (triggered by a script in a document), the body of the response is given to the script of that document for processing when it is called next.

B Additional Privacy Attack Variants

We here present three additional variants of the privacy attack introduced in Section 6.1.

Variant 2: PIF as Attack Source. Step $\boxed{a}$ can also be launched from within a PIF itself (i.e., the PIF also takes the role of `idp.com/index` above). This way, while the user logs in at some r_1, the IdP could check whether the user is logged in at r_2, for any r_2.

Variant 3: Scanning the Window Structure (I). Instead of using a postMessage to alert the IdP's outer document about the existence of the inner PIF document, the outer document could as well repeatedly scan the window tree of the iframe containing r's web site: While the IdP sees almost no information about r's document in the iframe (as it is not same origin), it can see the list of subwindows (i.e., the CIF, and possibly other iframes). For these frames, again, it would see the subwindows, especially the PIF, which it could identify uniquely by checking whether it is same origin with the IdPs outer window.

Variant 4: Scanning the Window Structure (II). In Variant 2, using a same-origin check, the malicious IdP can uniquely identify the PIF in the window structure. This same-origin check could be skipped and it could only be checked whether a PIF is generated, based on the window structure alone. While this is less reliable, this attack could be launched by *any* third party web attacker (not only the IdP to which the user's email address belongs) to check whether the victim is logged in at r or not.