An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System

Daniel Fett, Ralf Küsters, and Guido Schmitz
University of Trier, Germany
Email: {fett,kuesters,schmitzg}@uni-trier.de

***Abstract*—The web constitutes a complex infrastructure and, as demonstrated by numerous attacks, rigorous analysis of standards and web applications is indispensable.**

Inspired by successful prior work, in particular the work by Akhawe et al. as well as Bansal et al., in this work we propose a formal model for the web infrastructure. While unlike prior works, which aim at automatic analysis, our model so far is not directly amenable to automation, it is much more comprehensive and accurate with respect to the standards and specifications. As such, it can serve as a solid basis for the analysis of a broad range of standards and applications.

As a case study and another important contribution of our work, we use our model to carry out the first rigorous analysis of the BrowserID system (a.k.a. Mozilla Persona), a recently developed complex real-world single sign-on system that employs technologies such as AJAX, cross-document messaging, and HTML5 web storage. Our analysis revealed a number of very critical flaws that could not have been captured in prior models. We propose fixes for the flaws, formally state relevant security properties, and prove that the fixed system in a setting with a so-called secondary identity provider satisfies these security properties in our model. The fixes for the most critical flaws have already been adopted by Mozilla and our findings have been rewarded by the Mozilla Security Bug Bounty Program.

***Keywords*-Web Security; Formal Security Analysis; Web Model; Single Sign-on**

I. Introduction

The World Wide Web is a complex infrastructure, with a rich set of security requirements and entities, such as DNS servers, web servers, and web browsers, interacting using diverse technologies. New technologies and standards (for example, HTML5 and related technologies) introduce even more complexity and security issues. As illustrated by numerous attacks (see, e.g., [2], [6], [20], [27], [30]), rigorous analysis of the web infrastructure and web applications is indispensable.

Inspired by successful prior work, in particular the work by Akhawe et al. [2] and Bansal et al. [5], [6], one goal of our work is to develop an expressive formal model that precisely captures core security aspects of the web infrastructure, where we intend to stay as closely to the standards as possible, with a level of abstraction that is suitable for precise formal analysis. As further discussed in Section VI, while prior work aimed at automatic analysis, here our main focus is to obtain a comprehensive and more accurate model with respect to the standards and specifications. As such, our model constitutes a solid basis for the analysis of a broad range of standards and applications.

The standards and specifications that define the web are spread across many documents, including the HTTP standard RFC2616 (with its successor HTTPbis) and the HTML5 specification [18], with certain aspects covered in related documents, such as RFC6265, RFC6797, RFC6454, the WHATWG Fetch living standard [32], the W3C Web Storage specification [31], and the W3C Cross-Origin Resource Sharing specification [12], to name just a few. Specifications for the DNS system and communication protocols, such as TCP, are relevant as well. The documents often build upon each other, replace older versions or other documents, and sometimes different versions coexist. Some details or behaviors are not specified at all and are only documented in the form of the source code of a web browser.

Coming up with an accurate formal model is, hence, very valuable not only because it is required as a basis to precisely state security properties and perform formal analysis, but also because it summarizes and condenses important aspects in several specifications that are otherwise spread across different documents.

Another goal and important contribution of our work is to apply our model to the BrowserID system (also known under the marketing name *Mozilla Persona*), a complex real-world single sign-on system developed by Mozilla. BrowserID makes heavy use of several web technologies, including AJAX, cross-document messaging (postMessages), and HTML5 web storage, and as such, is a very suitable and practically relevant target to demonstrate the importance of a comprehensive and accurate model.

More precisely, the main contributions of our work can be summarized as follows.

Web model: We propose a formal model of the web infrastructure and web applications. Our model is based on a general Dolev-Yao-style communication model, in which processes have addresses (modeling IP addresses) and, as usual in Dolev-Yao-style models for cryptographic protocols (see, e.g., [1]), messages are modeled as formal terms, with properties of cryptographic primitives, such as encryption and digital signatures, expressed as equational theories on terms.

As mentioned before, our model is intended to be expressive and close to the standards and specifications, while providing a suitable level of abstraction. Our model includes web servers, web browsers, and DNS servers. We model HTTP(S) requests and responses, including several headers, such as host, cookie, location, strict-transport-security (STS), and origin headers. Our model of web browsers captures the concepts of windows, documents, and iframes as well as new technologies, such as web storage and cross-document messaging. It takes into account the complex security restrictions that are applied when accessing or navigating other windows. JavaScript is modeled in an abstract way by what we call scripting processes. These processes can be sent around and, among others, they can create iframes and initiate XMLHTTPRequests (XHRs). We also consider two ways of dynamically corrupting browsers. Altogether, our model is the most comprehensive model for the web infrastructure to date (see also Section VI).

Analysis of the BrowserID system: We use our model to perform the first rigorous security analysis of the BrowserID system, which supports both so-called primary and secondary identity providers. Our security analysis reveals a number of very critical and previously unknown flaws, most of which cannot be captured by previous models (see Section VI). The most severe attack allows an adversary to login to any service that supports authentication via BrowserID with the email address of *any* Gmail and Yahoo user (without knowing the Gmail/Yahoo credentials of these users), hence, breaking the system completely. Another critical attack allows an attacker to force a user to login with the attacker's identity. We confirmed that the attacks work on the actual BrowserID implementation. We propose fixes and formulate relevant security properties. For the BrowserID system with a secondary identity provider, we prove that the fixed system satisfies these properties in our model. By this, we provide the first rigorous formal analysis of the BrowserID system. Our attacks have been acknowledged by Mozilla, with the fixes for the most severe problems having been adopted by Mozilla already and other fixes being under discussion. Our findings have been rewarded by the Mozilla Security Bug Bounty Program.

Structure of this Paper: In Section II, we present the basic communication model. Our web model is introduced in Section III. For our case study, we first, in Section IV, provide a description of the BrowserID system. We then, in Section V, present the analysis of BrowserID using our model. Related work is discussed in Section VI. We conclude in Section VII. We refer the reader to [14] for the full version of this paper.

II. Communication Model

We now present a generic Dolev-Yao-style communication model on which our web model (see Section III) is based. While the model is stated in a concise mathematical fashion, instantiations, for example, using the applied pi-calculus [1] or multi-set rewriting [13], are conceivable.

The main entities in the communication model are what we call *atomic processes*, which in Section III are 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 we call 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. (In our web model, presented in Section III, only network attackers may listen to addresses of other atomic processes.)

Terms, Messages and Events: To define the communication model just sketched, we first define, as usual in Dolev-Yao models, messages, such as HTTP messages, as formal terms over a signature, and based on this notion of messages, we introduce events.

The signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols: (1) constants $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \Diamond\}$ where the three sets are pairwise disjoint, $\mathbb{S}$ is interpreted to be the set of ASCII strings (including the empty string ε), and IPs is interpreted to be a set of (IP) addresses, (2) function symbols for public keys, asymmetric/symmetric encryption/decryption, and digital signatures: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, $\mathsf{extractmsg}(\cdot)$, (3) n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and (4) projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

Let $X = \{x_0, x_1, \dots\}$ be a set of variables and $\mathcal{N}$ be an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \geq 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term. By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$. For example, $k \in \mathcal{N}$ and $\mathrm{pub}(k)$ are messages, where k typically models a private key and $\mathrm{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a, b, c\rangle, \mathrm{pub}(k))$ is interpreted to be the message $\langle a, b, c\rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathrm{pub}(k)$.

For strings, i.e., elements in $\mathbb{S}$, we use a specific font. For

example, HTTPReq and HTTPResp are strings. We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of domains, e.g., www.example.com $\in \mathsf{Doms}$. We denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of methods used in HTTP requests, e.g., GET, POST $\in \mathsf{Methods}$.

The equational theory associated with the signature Σ is given as follows: $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathrm{pub}(y)), y) = x$, $\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x$, $\mathsf{checksig}(\mathsf{sig}(x, y), \mathrm{pub}(y)) = \top$, $\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x$, and $\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i$ for $1 \leq i \leq n$. By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by this theory. For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b} \rangle, \mathrm{pub}(k)), k)) \equiv \mathsf{a}$.

An *event (over* IPs *and* $\mathcal{M}$*)* is of the form $(a{:}f{:}m)$, for $a, f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events.

Atomic Processes, Systems and Runs: We now define atomic processes, systems, and runs of systems.

An atomic process takes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

Definition 1. *A* (generic) atomic process *is a tuple* $p = (I^p, Z^p, R^p, s_0^p)$ *where* $I^p \subseteq \mathsf{IPs}$*,* Z^p *is a set of states,* $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}} \times Z^p)$*, and* $s_0^p \in Z^p$ *is the initial state of* p*. We write* $(e, z)R(E, z')$ *instead of* $((e, z), (E, z')) \in R$*.*

A system $\mathcal{P}$ *is a (possibly infinite) set of atomic processes.*

In order to define a run of a system, we first define configurations and processing steps.

A *configuration of a system* $\mathcal{P}$ is a tuple (S, E) where S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$ and E is a (possibly infinite) multi-set of events waiting to be delivered.

A *processing step of the system* $\mathcal{P}$ is of the form $(S, E) \rightarrow (S', E')$ such that there exist $e = (a{:}f{:}m) \in E$, $E_{\text{out}} \subseteq E'$, and $p \in \mathcal{P}$ with $(e, S(p))R^p(E_{\text{out}}, S'(p))$, $a \in I^p$, $S'(p') = S(p')$ for all $p' \neq p$, and $E' = (E \setminus \{e\}) \cup E_{\text{out}}$ (multi-set operations).

Definition 2. *Let* $\mathcal{P}$ *be a system and* E_0 *be a multi-set of events. A* run ρ of a system $\mathcal{P}$ initiated by E_0 *is a finite sequence of configurations* $(S_0, E_0), \ldots, (S_n, E_n)$ *or an infinite sequence of configurations* $(S_0, E_0), \ldots$ *such that* $S_0(p) = s_0^p$ *for all* $p \in \mathcal{P}$ *and* $(S_i, E_i) \rightarrow (S_{i+1}, E_{i+1})$ *for all* $0 \leq i < n$ *(finite run) or for all* $i \geq 0$ *(infinite run).*

Atomic Dolev-Yao Processes: We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$, and $t_1, \ldots, t_n \in \mathcal{T}_N$. Then, by $\tau[t_1/x_1, \ldots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \ldots, n\}$. Let $M \subseteq \mathcal{M}$ be a set of messages. We say that *a message* m *can be derived from* M *with nonces* N if there exist $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_N(M)$ the set of all messages that can be derived from M with nonces N. For example, $a \in d_{\{k\}}(\{\mathsf{enc_a}(\langle a, b, c \rangle, \mathrm{pub}(k))\})$.

Definition 3. *An* atomic Dolev-Yao process (or simply, a DY process) *is a tuple* $p = (I^p, Z^p, R^p, s_0^p, N^p)$ *such that* (I^p, Z^p, R^p, s_0^p) *is an atomic process and (1)* $N^p \subseteq \mathcal{N}$ *is an (initial) set of nonces, (2)* $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ *(and hence,* $s_0^p \in \mathcal{T}_{\mathcal{N}}$*), and (3) for all* $a, a', f, f' \in \mathsf{IPs}$*,* $m, m', s, s' \in \mathcal{T}_{\mathcal{N}}$*, set of events* E *with* $((a{:}f{:}m), s)R(E, s')$ *and* $(a'{:}f'{:}m') \in E$ *it holds true that* $m', s' \in d_N(\{m, s\})$*. (Note that* $a', f' \in d_N(\{m, s\})$*.)*

In the rest of this paper, we will only consider DY processes and assume different DY processes to have disjoint initial sets of nonces.

We define a specific DY process, called an attacker 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 can carry out all attacks any DY process could possibly perform. The attacker process is parametrized by the set of sender addresses it may use.

Definition 4. *An* (atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$ *is an atomic DY process* $p = (I, Z, R, s_0, N)$ *such that for all* $a, f \in \mathsf{IPs}$*,* $m \in \mathcal{T}_{\mathcal{N}}$*, and* $s \in Z$ *we have that* $((a{:}f{:}m), s)R(E, s')$ *iff* $s' = \langle \langle a, f, m \rangle, s \rangle$ *and* $E = \{(a'{:}f'{:}m') \mid a' \in \mathsf{IPs}, f' \in A, m' \in d_N(\{m, s\})\}$*.*

III. Our Web Model

We now present our web model. We formalize the web infrastructure and web applications by what we call a web system. A web system, among others, contains a (possibly infinite) set of DY processes, which model web browsers, web servers, DNS servers as well as web and network attackers.

As already mentioned in the introduction, the model has been carefully designed, closely following published (de-facto) standards, for instance, the HTTP/1.1 standard, associated (proposed) standards (mainly RFCs), and the HTML5 W3C candidate recommendation. We also checked these standards against the actual implementations (primarily, Chromium and Firefox).

A. Web System

Before we can define a web system, we define scripting processes, which model client-side scripting technologies, such as JavaScript, in our browser model. Scripting processes are defined similarly to DY processes.

Definition 5. *A* scripting process *(or simply, a* script*) is a relation $R \subseteq (\mathcal{T}_{\mathcal{N}} \times 2^{\mathcal{N}}) \times \mathcal{T}_{\mathcal{N}}$ such that for all $s, s' \in \mathcal{T}_{\mathcal{N}}$ and $N \subseteq \mathcal{N}$ with $(s, N)\, R\, s'$ it follows that $s' \in d_N(s)$.*

A script is called by the browser which provides it with a (fresh, infinite) set N 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 Section III-D for details).

Similarly to an attacker process, we define the *attacker script* R^{att}. This script outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{((s, N), s') \mid s \in \mathcal{T}_{\mathcal{N}}, N \subseteq \mathcal{N}, s' \in d_N(s)\}$.

We can now define web systems, where we distinguish between web and network attackers. Unlike web attackers, network attackers can listen to addresses of other parties and can spoof the sender address, i.e., they can control the network. Typically, a web system has either one network attacker or one or more web attackers, as network attackers subsume all web attackers. As we will see later, web and network attacks may corrupt other entities, such as browsers.

Definition 6. *A* web system $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ *is a tuple with its components defined as follows:*

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon*,* Web*, and* Net *of honest, web attacker, and network attacker processes, respectively. We require that all DY processes in $\mathcal{W}$ have disjoint sets of nonces, i.e., $N^p \cap N^{p'} = \emptyset$ for every distinct $p, p' \in \mathcal{W}$.*

Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs*).*

Every $p \in \mathsf{Hon}$ is a DY process which models either a web server*, a* web browser*, or a* DNS server*, as further described in the following subsections. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.*

The second component, $\mathcal{S}$, is a finite set of scripts such that $R^{att} \in \mathcal{S}$. The third component, script*, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by* script *every $s \in \mathcal{S}$ is assigned its string representation* $\mathsf{script}(s)$.

Finally, E_0 is a multi-set of events, containing an infinite number of events of the form $(a{:}a{:}\texttt{TRIGGER})$ for every $a \in \bigcup_{p \in \mathcal{W}} I^p$.

A run *of $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E_0.*

In the definition above, the multi-set E_0 of initial events contains for every process and address an infinite number of `TRIGGER` messages in order to make sure that every process in $\mathcal{W}$ can be triggered arbitrarily often. In particular, by this it is guaranteed that an adversary (a dishonest server/browser) can be triggered arbitrarily often. Also, we use trigger events to model that an honest browser takes an action triggered by a user, who might, for example, enter a URL or click on some link.

The set $\mathcal{S} \setminus \{R^{\text{att}}\}$ specified in a web system as defined above is meant to describe the set of honest scripts used in the considered web application. These scripts are those sent out by an honest web server to a browser as part of a web application. In real web applications, possibly several dynamically loaded scripts may run in one document. However, if these scripts originate from honest sites, their composition can be considered to be one honest script (which is loaded right from the start into the document). In this sense, every script in $\mathcal{S} \setminus \{R^{\text{att}}\}$ models an honest script or a combination of such scripts in a web application. (In our case study, the combination is illustrated by the script running in RP-Doc.)

We model the situation where some malicious script was loaded into a document by the "worst-case" scenario, i.e., we allow such a script to be the script R^{att}. This script subsumes *everything* any malicious (and honest) script can do.

We emphasize that script representations being modeled as strings are public information, i.e., any server or attacker is free to send out the string representation for any script.

Since we do not model client-side or server-side language details, and hence details such as correct escaping of user input, we cannot analyze whether a server application (say, written in PHP) is vulnerable to Cross-Site-Scripting. However, we can model the effects of Cross-Site-Scripting by letting the (model of the) server output the script R^{att}, say, if it receives certain malicious input.

In the following subsections, (honest) DNS servers and web browsers are modeled as DY processes, including the modeling of HTTP messages. We also discuss the modeling of web servers.

B. DNS Servers

For the sake of brevity, in this paper we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover certain attacks on the DNS system itself.

A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as a DY process $(I^d, \{s_0^d\}, R^d, s_0^d, N^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form $s_0^d = \langle\langle \texttt{domain}_1, a_1\rangle, \langle \texttt{domain}_2, a_2\rangle, \ldots\rangle$. DNS queries are answered according to this table. DNS

Algorithm 1 Relation of a DNS server R^d

Input: $(a : f : m), s$

1: **let** $domain, n$ **such that** $\langle \texttt{DNSResolve}, domain, n\rangle \equiv m$ **if possible; otherwise stop** $\{\}, s$
2: **if** $domain \in s$ **then**
3: **let** $addr := s[domain]$
4: **let** $m' := \langle \texttt{DNSResolved}, addr, n\rangle$
5: **stop** $\{(f{:}a{:}m')\}, s$
6: **stop** $\{\}, s$

queries have the following form, illustrated by an example: $\langle$`DNSResolve`, `example.com`, $n\rangle$, where `example.com` is the domain name to be resolved and n is a nonce representing the random query ID and UDP source port number selected by the sender of the query. The corresponding response is of the form $\langle \texttt{DNSResolved}, a, n\rangle$, where $a \in \mathsf{IPs}$ is the IP address of the queried domain name and n is the nonce from the query.

In Algorithm 1, we specify the relation $R^d \subseteq (\mathcal{E}\times\{s_0^d\})\times(2^{\mathcal{E}} \times \{s_0^d\})$ of the DNS server d precisely, where **stop** E, s means that the process stops its execution at this point, that s is the new state of the process, and that it outputs all events in the set E. First, it is checked whether the input message m is a sequence of the form $\langle \texttt{DNSResolve}, domain, n\rangle$; if not, the process stops without changing the state and producing output. Then, it is checked whether $domain$ is recorded in s. If so, the corresponding address, denoted by $s[domain]$, is retrieved from s. Finally, the corresponding response message m' is constructed and this message is output as event $(f{:}a{:}m')$, with the state of d being unchanged.

C. HTTP Messages

In order to model web browsers and servers, we first need to model HTTP requests and responses.

HTTP requests and responses are encoded as messages (ground terms). An HTTP request (modeled as a message) contains a nonce, a method (for example, `GET` or `POST`), a domain name, a path, URL parameters, request headers (such as `Cookie` or `Origin`), and a message body. For example, an HTTP `GET` request for the URL http://example.com/show?page=1 is modeled as the term $r := \langle \texttt{HTTPReq}, n_1, \texttt{GET}, \texttt{example.com}, /\texttt{show}, \langle\langle \texttt{page}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$, where body and headers are empty. A web server that responds to this request is supposed to include the nonce n_1 contained in r in the response so that the browser can match the request to the corresponding response. More specifically, an HTTP response (modeled as a message) contains a nonce (matching the request), a status code (e.g., `200` for a normal successful response), response headers (such as `Set-Cookie` and `Location`), and a body. For example, a response to r could be $s := \langle \texttt{HTTPResp}, n_1, \texttt{200}, \langle\langle \texttt{Set-Cookie}, \langle \texttt{SID}, \langle n_2, \bot, \top, \bot\rangle\rangle\rangle, \langle \texttt{script1}, n_3\rangle\rangle$, where s contains (1) in the headers section, a cookie with the name `SID`, the value n_2, and the attributes `secure` and `httpOnly` not set but the attribute `session` set (see Section III-D for details on cookies) and (2) in the body section, the string representation `script1` of the scripting process $\texttt{script}^{-1}(\texttt{script1})$ (which should be an element of $\mathcal{S}$) and its initial state n_3.

For the HTTP request and response in the above examples, the corresponding HTTPS request would be of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathrm{pub}(k_{\text{example.com}}))$ and the response of the form $\mathsf{enc_s}(s, k')$ where k' is a fresh symmetric key (a nonce) which is typically generated by the sender of the request. The responder is supposed to use this key to encrypt the response.

D. Web Browsers

We think of an honest browser to be used by one honest user. However, we also allow browsers to be taken over by attackers. The honest user is modeled as part of the web browser model. Actions a user takes are modeled as non-deterministic actions of the web browser. For example, the web browser itself can non-deterministically follow the links provided by a web page. Secrets, such as passwords, typically provided by the user are stored in the initial state of a browser and are given to a web page when needed, similar to the AutoFill function in browsers (see below).

A web browser p is modeled as a DY process $(I^p, Z^p, R^p, s_0^p, N^p)$ where $I^p \subseteq \mathsf{IPs}$ is a finite set and $N^p \subseteq \mathcal{N}$ is an infinite set. The set of states Z^p, the initial state s_0^p, and the relation R^p are defined below (Sections III-D1 and III-D2). In the full version of this paper [14], R^p is formally defined as a (non-deterministic) algorithm in the style of Algorithm 1.

1) Browser State: Z^p and s_0^p: The set Z^p of states of a browser consists of terms of the form

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

Windows and documents. The most important part of the state are windows and documents, both stored in the subterm $windows$. A browser may have a number of windows open at any time (resembling the tabs in a real browser). Each window contains a list of documents of which one is "active". Being active means that this document is currently presented to the user and is available for interaction, similarly to the definition of active documents in the HTML5 specification [18]. The document list of a window represents the history of visited web pages in that window. A window may be navigated forward and backward (modeling forward and back buttons). This deactivates one document and activates its successor or predecessor.

A document is specified by a term which essentially contains (the string representing) a script, the current state of the script, the input that the script obtained so far (from XHRs and postMessages), the origin (domain name

plus HTTP or HTTPS) of the document, and a list of windows (called *subwindows*), which correspond to iframes embedded in the document, resulting in a tree of windows and documents. The (single) script is meant to model the static HTML code, including, for example, links and forms, and possibly multiple JavaScript code parts. When called by the browser, a script essentially outputs a command which is then interpreted by the browser, such as following a link, creating an iframe, or issuing an XHR. In particular, a script can represent a plain HTML document consisting merely of links, say: when called by the browser such a script would non-deterministically choose such a link and output it to the browser, which would then load the corresponding web page (see below for details).

We use the terms *top-level window* (a window which is not a subwindow itself), *parent window* (the window of which the current window is a direct subwindow) and *ancestor window* (some window of which the current window is a not necessarily direct subwindow) to describe the relationships in a tree of windows and documents.

A term describing a window or a document also contains a unique nonce, which we refer to by *reference*. This reference is used to match HTTP responses to the corresponding windows and documents from which they originate (see below).

Top-level windows may have been opened by another window. In this case, the term of the opened window contains a reference to the window by which it was opened (the *opener*). Following the HTML5 standard, we call such a window an *auxiliary window*. Note that auxiliary windows are always top-level windows.

We call a window *active* if it is a top-level window or if it is a subwindow of an active document in an active window. Note that the active documents in all active windows are exactly those documents a user can currently see/interact with in the browser.

The following is an example of a window term with reference n_1, two documents, and an opener (n_4):

$$\langle n_1, \langle \langle n_2, \langle \texttt{example.com}, \texttt{P} \rangle, \texttt{script1}, \langle\rangle, \langle\rangle, \langle\rangle, \bot \rangle, \\ \langle n_3, \langle \texttt{example.com}, \texttt{S} \rangle, \texttt{script2}, \langle\rangle, \langle\rangle, \langle\rangle, \top \rangle \rangle, n_4 \rangle$$

The first document has reference n_2. It was loaded from the origin $\langle \texttt{example.com}, \texttt{P} \rangle$, which translates into http://example.com. Its scripting process has the string representation `script1`, the last state and the input history of this process are empty. The document does not have subwindows and is inactive ($\bot$). The second document has the reference n_3, its origin corresponds to https://example.com, the scripting process is represented by `script2`, and the document is active ($\top$). All other components are empty.

Secrets. This subterm of the state term of a browser holds the secrets of the user of the web browser. Secrets (such as passwords) are modeled as nonces and they are indexed by origins. Secrets are only released to documents (scripts) with the corresponding origin, similarly to the AutoFill mechanism in browsers.

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. Cookies are stored together with their `secure`, `httpOnly`, and `session` attributes: If `secure` is set, the cookie is only delivered to HTTPS origins. If `httpOnly` is set, the cookie cannot be accessed by JavaScript (the script). According to the proposed standard RFC6265 (which we follow in our model) and the majority of the existing implementations, cookies that neither have the (real) "max-age" nor the "expires" attribute should be deleted by the browser when the session ends (usually when the browser is closed). In our model, such cookies carry the `session` attribute.

KeyMapping. This term is our equivalent to a certificate authority (CA) certificate store in the browser. Since, for simplicity, we currently do not formalize CAs in the model, this term simply encodes a mapping assigning domains $d \in \mathsf{Doms}$ to their respective public keys $\mathrm{pub}(k_d)$.

STS. Domains that are listed in this term are contacted by the web browser only over HTTPS. Connection attempts over HTTP are transparently rewritten to HTTPS requests. Web sites can issue the `Strict-Transport-Security` header to clients in order to add their domain to this list, see below.

DNSaddress. This term contains the address of the DY process that is to be contacted for DNS requests; typically a DNS server.

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 requests that await successful DNS resolution, and HTTP requests that await a response, respectively.

IsCorrupted. This term indicates whether the browser is corrupted ($\neq \bot$) or not ($= \bot$). A corrupted browser behaves like a web attacker (see Section III-D2).

Initial state s_0^p of a web browser. In the initial state, $keyMapping$, $DNSAddress$, and $secrets$ are defined as needed, $isCorrupted$ is set to $\bot$, and all other subterms are $\langle\rangle$.

2) Web Browser Relation R^p: Before we define the relation R^p, we first sketch the processing of HTTP(S) requests and responses by a web browser, and also provide some intuition about the corruption of browsers.

HTTP(S) Requests and Responses. An HTTP request, contains, as mentioned before, a nonce created by the browser. In the example in Section III-C, this nonce is n_1. A server is supposed to include this nonce into its HTTP response. By this, the browser can match the response to the request (a real web browser would use the TCP sequence number for this purpose). If a browser wants to send an

HTTP request, it first resolves the domain name to an IP address. (For simplicity, we do not model DNS response caching.) It therefore first records the HTTP request in $pendingDNS$ along with the reference of the window (in the case of HTTP(S) requests) or the reference of the document[1] (in the case of XHRs) from which the request originated and then sends a DNS request. Upon receipt of the corresponding DNS response it sends the HTTP request and stores it (again along with the reference as well as the server address) in $pendingRequests$. Before sending the HTTP request, the cookies stored in the browser for the domain of the request are added as cookie headers to the request. Cookies with attribute `secure` are only added for HTTPS requests. If an HTTP response arrives, the browser uses the nonce in this response to match it with the recorded corresponding HTTP request (if any) and checks whether the address of the sender is as expected. The reference recorded along with the request then determines to which window/document the response belongs. The further processing of a response is described below.

We note that before HTTPS requests are sent out, a fresh symmetric key (a nonce) is generated and added to the request by the browser. The resulting message is then encrypted using the public key corresponding to the domain in the request (according to $keyMapping$). The symmetric key is recorded along with the request in $pendingRequests$. The response is, as mentioned, supposed to be encrypted with this symmetric key.

Corruption of Browsers. We model two types of corruption of browsers, namely *full corruption* and *close-corruption*, which are triggered by special network messages in our model. In the real world, an attacker can exploit buffer overflows in web browsers, compromise operating systems (e.g., using trojan horses), and physically take control over shared terminals.

Full corruption models an attacker that gained full control over a web browser and its user. Besides modeling a compromised system, full corruption can also serve as a vehicle for the attacker to participate in a protocol using secrets of honest browsers: In our case study (Section V), the attacker starts with no user secrets in its knowledge, but may fully corrupt any number of browsers, so, in particular, he is able to impersonate browsers/users.

Close-corruption models a browser that is taken over by the attacker after a user finished her browsing session, i.e., after closing all windows of the browser. This form of corruption is relevant in situations where one browser can be used by many people, e.g., in an Internet café. Information left in the browser state after closing the browser could be misused by malicious users.

[1]As we will see later, in the case of XHRs this reference is actually a sequence of two elements, a document reference and a nonce that was chosen by the script that issued the XHR. For now, we will refer to this sequence simply as the document reference.

PROCESSING INPUT MESSAGE $\boldsymbol{m}$

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

Figure 1. The basic structure of the web browser relation R^p with an extract of the most important processing steps, in the case that $isCorrupted = \perp$.

The Relation R^p. To define R^p, we need to specify, given the current state of the browser and an input message m, the new state of the browser and the set of events output by the browser. Figure 1 provides an overview of the structure of the following definition of R^p. The input message m is expected to be `FULLCORRUPT`, `CLOSECORRUPT`, `TRIGGER`, a DNS response, or an HTTP(S) response.

If $isCorrupted \neq \perp$ (browser is corrupted), the browser, just like an attacker process, simply adds m to its current state, and then outputs all events it can derive from its state. Once corrupted, the browser stays corrupted. Otherwise, if $isCorrupted = \perp$, on input m the browser behaves as follows.

$\boldsymbol{m}$ = `FULLCORRUPT`**:** If the browser receives this message, it sets $isCorrupted$ to `FULLCORRUPT`. From then on the browser is corrupted as described above, with the attacker having full access to the browser's internal state, including all secrets.

$\boldsymbol{m}$ = `CLOSECORRUPT`**:** If the browser receives this message, it first removes the user secrets, open windows and documents, all *session* cookies, all sessionStorage data, and all pending requests from its current state; nonces used so far by the browser may not be used any longer. LocalStorage data and persistent cookies are not deleted. The browser then sets $isCorrupted$ to `CLOSECORRUPT` (and hence, from then on is corrupted). As already mentioned, this models that the browser is closed by a user and that then the browser is used by another, potentially malicious user (an attacker), such as in an Internet café.

$\boldsymbol{m}$ = `TRIGGER`**:** Upon receipt of this message, the browser

non-deterministically chooses one of two *actions*: (1) trigger a script or (2) request a new document.

m = TRIGGER, $action$ = **1:** Some active window (possibly an iframe) is chosen non-deterministically. Then the script of the active document of that window is triggered (see below).

m = TRIGGER, $action$ = **2:** A new HTTP(S) GET request (i.e., an HTTP(S) request with method GET) is created where the URL is some message derivable from the current state of the browser. However, nonces may not be used. This models the user typing in a URL herself, but we do not allow her to type in secrets, e.g., passwords or session tokens. A new window is created to show the response. (HTTP requests to domains listed in sts are automatically rewritten to HTTPS requests).

m = **DNS response:** DNS responses are processed as already described above, resulting in sending the corresponding HTTP(S) request (if any).

m = **HTTP(S) response:** The browser performs the steps (I) to (IV) in this order.

(I) The browser identifies the corresponding HTTP(S) request (if any), say q, and the window or document from which q originated. (In case of HTTPS, the browser also decrypts m using the recorded symmetric key.)

(II) If there is a `Set-Cookie` header in the response, its content (name, value, and if present, the attributes `httpOnly`, `secure`, `session`) is evaluated: The cookie's name, value, and attributes are saved in the browser's list of cookies. If a cookie with the same name already exists, the old values and attributes are overwritten, as specified in RFC6265.

(III) If there is a `Strict-Transport-Security` header in the response, the domain of q is added to the term sts. As defined in RFC6797, all future requests to this domain, if not already HTTPS requests, are automatically altered to use HTTPS.

(IV) If there is a `Location` header (with some URL u) in the response and the HTTP status code is 303 or 307, the browser performs a redirection (unless it is a non-same-origin redirect of an XHR) by issuing a new HTTP request to u, retaining the body of the original request. Rewriting POST to GET requests for 303 redirects and extending the origin header value are handled as defined in RFC2616 and in the W3C Cross-Origin Resource Sharing specification [12].

Otherwise, if no redirection is requested, the browser does the following: If the request originated from a window, a new document is created from the response body. For this, the response body is expected to be a term of the form $\langle sp, stat\rangle$ where sp is a string such that $\mathsf{script}^{-1}(sp) \in \mathcal{S}$ is a script and $stat$ is a term used as its initial state. The document is then added to the window the reference points to, it becomes the active document, and the successor of the currently active document. All previously existing successors are removed. If the request originated from a document (and hence, was the result of an XHR), the body of the response is appended to the script input term of the document. When later the script of this document is activated, it can read and process the response.

Triggering the Script of a Document (m = TRIGGER, $action$ = 1). First, the script of the document is called with the following input:

- all active windows[2] and their active documents (with limited information about non-same-origin documents),
- the last state and the input history (i.e., previous inputs from postMessages and XHRs) of the script as recorded in the document,
- cookies (names and values only) indexed with the document's domain, except for `httpOnly` cookies,
- localStorage data and secrets indexed with the document's origin, and
- sessionStorage data indexed with the document's origin and top-level window reference.

In addition, the script is given an infinite set of fresh nonces from the browser's set of (unused) nonces.

Now, given the above input, according to the definition of scripts (Definition 5), the script outputs a term. The browser expects terms of the form

$$\langle state, cookies, localStorage, sessionStorage, cmd\rangle$$

(and otherwise ignores the output) where $state$ is an arbitrary term describing the new state of the script, $cookies$ is a sequence of name/value pairs, $localStorage$ and $sessionStorage$ are arbitrary terms, and cmd is a term which is interpreted as a command which is to be processed by the browser. The old state of the script recorded in the document is replaced by the new one ($state$), the local/session storage recorded in the browser for the document's origin (and top-level window reference) is replaced by $localStorage$/$sessionStorage$, and the old cookie store of the document's origin is updated using $cookies$ similarly to the case of HTTP(S) responses with cookie headers, except that now no httpOnly cookies can be set or replaced, as defined by the HTML5 standard [18] and RFC6265.

Subsequently, cmd (if not empty) is interpreted by the browser, as described briefly next. We note that commands may contain parameters.

cmd = HREF (parameters: URL u, window reference w): A new GET request to u is initiated. If w is _BLANK, the response to the request will be shown in a new *auxiliary* window. Otherwise, if w is not _BLANK, the window with reference w is navigated (upon receipt of the response and only if it is active) to the given URL. Navigation is subject

[2]Note that we overapproximate here: In real-world browsers, only a limited set of window handles are available to a script. Our approach is motivated by the fact that in some cases windows can be navigated by names (without a handle). However, as we will see, specific restrictions for navigating windows and accessing/changing their data apply.

to several restrictions.[3]

cmd = IFRAME: Similar to HREF, but opens the document in a new subwindow of the given window (when same origin).

cmd = FORM: Similar to HREF, but allows for methods other than GET and request body data. For this request, an Origin header is set if the method is POST. Its value is the origin of the document.

cmd = SETSCRIPT, SETSCRIPTSTATE, BACK, FORWARD, CLOSE: These commands change the browser's state such that the script (state) in a document is changed or the window is navigated back/forward or closed (if the document is same origin or the window is navigable, respectively).

cmd = XMLHTTPREQUEST (parameters: URL u, method md, data d, nonce $xhrreference$): Initiate a request with method md and data d for u, if u is same origin. The reference (for $pendingRequests$) used for this request is $\langle r, xhrreference\rangle$, where r is the reference of the script's document and $xhrreference$ is a nonce chosen by the script (for later correlation). The Origin header is set as in the case of FORM.

cmd = POSTMESSAGE (parameter: message msg, window reference w, origin o): msg, the origin of the sending document, and a reference to its window are appended to the input history of the active document in w if that document's origin matches o or if $o = \bot$.

E. Web Servers

While the modeling of DNS servers and browsers is independent of specific web applications, and hence, forms the core of the model of the web infrastructure, the modeling of a *web server* heavily depends on the specific web application under consideration. Conversely, the model of a specific web application is determined by the model of the web server. We therefore do not and cannot fix a model for web servers at this point. Such a model should be provided as part of the analysis of a specific web application, as illustrated by our case study (see Section IV and following).

F. Limitations

We now briefly discuss main limitations of the model. As will be illustrated by our case study, our model is formulated on a level of abstraction that is suitable to capture many security relevant features of the web, and hence, a relevant class of attacks. However, as with all models, certain attacks are out of the scope of our model. For example, as already mentioned, we currently cannot reason about language details (e.g., how two JavaScripts running in the same document interact). Also, we currently do not model user interface details, such as frames that may overlap in Clickjacking attacks. Being a Dolev-Yao-style model, our model clearly does not aim at lower-level cryptographic attacks. Also, byte-level attacks, such as buffer overflows, are out of scope.

[3]We follow the rules defined in [18]: A window A can navigate a window B if the active documents of both are same origin, or B is an ancestor window of A and B is a top-level window, or if there is an ancestor window of B whose active document has the same origin as the active document of A (including A itself). Also, A may navigate B if B is an auxiliary window and A is allowed to navigate the opener of B.

IV. The BrowserID System

BrowserID [23] is a new 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 code for Sideshow/BigTent, see below, and some libraries). It allows web sites to delegate user authentication to email providers, where users use their email addresses as identities. The BrowserID implementation makes use of a broad variety of browser features, such as XHRs, postMessage, local- and sessionStorage, cookies, etc.

We first, in Section IV-A, provide a high-level overview of the BrowserID system. A more detailed description of the BrowserID implementation is then given in Sections IV-B to IV-D.

A. Overview

The BrowserID system knows three distinct parties: the user, which 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@eyedee.me), and the identity/email address provider IdP. If the email provider (eyedee.me) supports BrowserID directly, it is called a *primary IdP*. Otherwise, a Mozilla-provided service, a so-called *secondary IdP*, takes the role of the IdP. In what follows, we describe the case of a primary IdP, with more information on secondary IdPs given in Section IV-D.

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

A user who wants to log in at an RP with an email address for some IdP has to present two signed documents: A *user certificate* (UC) and an *identity assertion* (IA). The UC contains the user's email address and a public key. It is signed by the IdP. The IA contains the origin of the RP and is signed with the private key corresponding to the user's public 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 claimed 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 wants to log in. Altogether, given a valid CAP, RP would consider the user (with the email address mentioned in the CAP) to be logged in.

The BrowserID authentication process (with a primary IdP) consists of three phases (see Figure 2 for an overview):

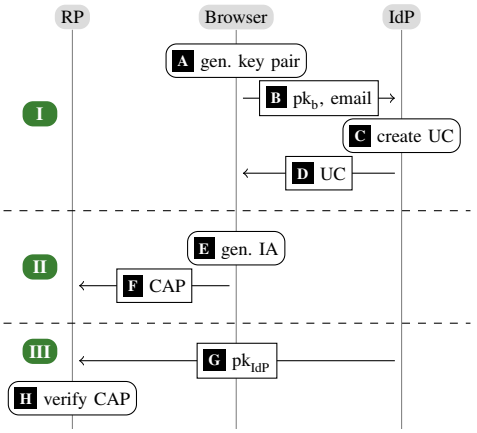

Figure 2. BrowserID authentication: basic overview

(I) provisioning of the UC, (II) CAP creation, and (III) verification of the CAP.

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

With the user having received the UC, Phase (II) can start. The user wants to authenticate to an RP, so she creates the IA [E]. The UC and the IA are concatenated to a CAP, which is then sent to the RP [F].

In Phase (III), the RP checks the authenticity of the CAP. For this purpose, the RP could use an external verification service provided by Mozilla or check the CAP itself as follows: First, the RP fetches the public key of IdP [G], which is contained in the support document. Afterwards, the RP checks the signatures of the UC and the IA [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 session ID), which we refer to as an *RP service token*.

B. Implementation Details

We now provide a more detailed description of the BrowserID implementation (see also Figure 3). 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.

In addition to the parties mentioned in the rough overview in Section IV-A, the actual implementation uses another party, login.persona.org (LPO). The role of LPO is as follows: First, LPO provides the HTML and JavaScript files of the implementation. Thus, the BrowserID implementation

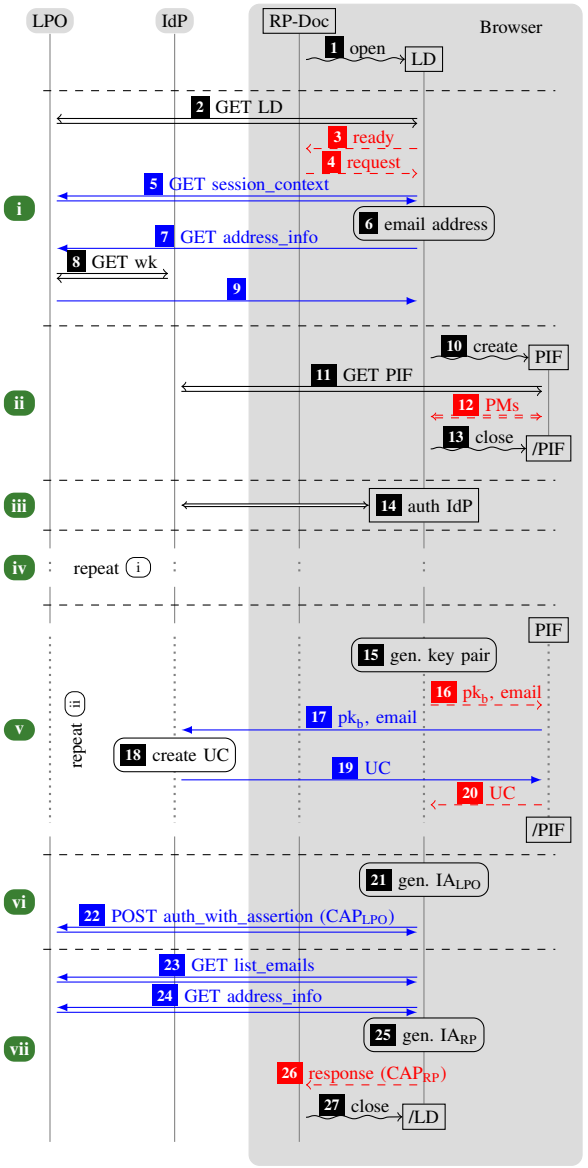

Figure 3. BrowserID implementation overview. Black arrows (open tips) denote HTTPS messages, blue arrows (filled tips) denote XHRs (over HTTPS), red (dashed) arrows are postMessages, snake lines are commands to the browser.

mainly runs under the origin of LPO.[4] When the JavaScript implementation running in the browser under the origin of LPO needs to retrieve information from the IdP (support document), LPO acts as a proxy to circumvent cross-origin restrictions.

Before explaining the login process, we provide a quick overview of the windows and iframes in the browser. By RP-Doc we denote the window (see Figure 3) containing the document loaded from some RP, a web page on which the user wants to log in with an email address of some IdP. This document typically includes JavaScript from LPO

[4]It is envisioned by Mozilla to integrate the part of LPO directly into the browser in the future.

and contains a button "Login with BrowserID". (Loading of *RP-Doc* from the RP and the JavaScript from LPO is not depicted in Figure 3). 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 IdP. Temporarily, the LD may navigate itself to a web page at IdP to allow for direct user interaction with the IdP.

Now, in order to describe the login process, for the time being we assume 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 (i), the LD is initialized and the user is prompted to provide her email address. Then LD fetches the support document (see Section IV-A) of IdP via LPO. In Phase (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 IdP yet. In Phase (iii), LD navigates itself away to the *authentication URL* which is also provided in the support document and links to 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 IdP sets a session cookie in the browser), the window is navigated to LPO again. (This is done by JavaScript loaded from LPO that the IdP document is supposed to include.)

Now, the login flow continues in Phase (iv), which basically repeats Phase (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 (v), which basically repeats Phase (ii), the PIF detects that the user is now authenticated to IdP and the provisioning phase is started ((I) in Figure 2): 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 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 (vi) and (vii), mainly the IA is generated by LD for the origin of RP-Doc and sent (together with the UC) to RP-Doc ((II) in Figure 2). In the localStorage, LD stores that the user's email is logged in at RP. Moreover, the user's email is recorded at LPO (see the explanation on LPO Sessions below). For this purpose, LD generates an IA for the origin of LPO and sends the UC and IA to LPO.

LPO Session. LPO establishes a session with the browser by setting a cookie `browserid_state` (in Step [5] in Figure 3) on the client-side. LPO considers such a session authenticated after having received a valid CAP (in Step [22] in Figure 3). In future runs, the user is presented a list of her email addresses (which is fetched from LPO) in order to choose one address. Then, she is asked if she trusts the computer she is using and is given the option to be logged in for one month or "for this session only" (*ephemeral* session). In order to use any of the email addresses, the user is required to authenticate to the IdP responsible for that address to get an UC issued. If the localStorage (under the origin LPO) already contains a valid UC, then, however, authentication at the IdP is not necessary.

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 of the BrowserID code included by 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 IdP.

Logout. We have to differentiate between three ways of logging out: an RP logout, an LPO logout, and an IdP logout. An RP logout is handled by the CIF after it has received a *logout* postMessage from RP-Doc. The CIF then changes the localStorage such that no email address is recorded to be logged in at RP.

An LPO logout essentially requires to logout at the web site of LPO. The LPO logout removes all key pairs and certificates from the localStorage and invalidates the session on the LPO server.

An IdP logout depends on the IdP implementation and usually cancels the user's session with IdP. This entails that IdP will not issue new UCs for the user without re-authentication.

C. Sideshow and BigTent

Since several email providers, such as gmail.com and yahoo.com, already use OpenID [24], a widely employed SSO system, Mozilla implemented IdPs called Sideshow and BigTent which use an OpenID backend for user authentication: Sideshow/BigTent are put between BrowserID and an email provider running OpenID. That is, BrowserID uses Sideshow/BigTent as an IdP. Sideshow/BigTent translate requests from BrowserID to requests to the email provider's OpenID interface. Currently, Sideshow and BigTent are used to provide BrowserID support for gmail.com and yahoo.com, respectively. In what follows, we describe Sideshow in more detail; BigTent is similar.

All BrowserID protocol steps that would normally be carried out by the IdP are now handled by Sideshow (i.e., the Sideshow server). For this purpose, Sideshow serves the provisioning URL (for the PIF) and the authentication URL used in (iii). It maintains a session with the user's browser. This session is considered to be authenticated if the user has successfully authenticated to Sideshow using OpenID. In this case, Sideshow's PIF document may send public keys to Sideshow. Sideshow then creates a UC for the identity it believes to be logged in. If the session at Sideshow is not authenticated, the user will first be redirected to the Sideshow authentication URL. Sideshow's authentication document will redirect the user further to the OpenID URL at Gmail. This URL contains an authentication request encoding that Sideshow requests an *OpenID assertion* that contains an email address. In general, such an assertion is a list of attribute name/value pairs (partially) MACed by Gmail with a temporary symmetric key known only to Gmail; an additional attribute, `openid.signed`, in such an assertion encodes which attribute name/value pairs have actually been MACed and in which order. The user now authenticates to Gmail. Then, Gmail issues the requested OpenID assertion and redirects the browser to Sideshow with the assertion in the URL parameters. Sideshow then sends the OpenID assertion to Gmail in order to check its validity. If the OpenID assertion is valid, i.e. the MAC over the attributes listed in `openid.signed` verifies, Sideshow considers its session with the user's browser to be authenticated for the email address contained in the OpenID assertion.

D. Secondary Identity Provider

If an email provider (IdP) does not directly support BrowserID, LPO can be used as a so-called *secondary IdP* (sIdP), i.e., it replaces the IdP completely. For this, the user has to register at LPO. That is, she creates an account at LPO where she can register one or more email addresses to be used as identities. She has to prove ownership of all email addresses she registers. (LPO sends URLs to each email address, which then have to be opened by the user.)

When the sIdP is used, the phases (ii) – (vi) are not needed as now LPO replaces the IdP and the actions previously performed by IdP and LPO are now carried out by LPO alone. The user is prompted to enter her password directly into LD. If the password is correct, LPO now considers the session with the browser to be authenticated. LPO will then issue UCs on behalf of the email provider. We note that, for automatic CAP creation, the CIF (see Section IV-B) is still used.

V. Analysis of BrowserID

In this section, we present the analysis of the BrowserID system. We first formulate fundamental security properties for the BrowserID system. We then present attacks that show that these properties are not satisfied and propose fixes. For the case of BrowserID with sIdP and the fixes applied, we then prove that the security properties are satisfied in our web model. We note that we also incorporate the automated CAP creation with the CIF in our model of BrowserID (see Section IV-B). Our web model is expressive enough to also formally model the BrowserID system with primary IdPs (and Sideshow/BigTent) in a straightforward way. However, we leave the detailed formulation of such a model and the proof of the security of the fixed system with *primary* IdPs to future work.

A. Security Properties for BrowserID

While the documentation of BrowserID does not contain explicit security goals, we deduce two fundamental security properties that can be informally described as follows (see Section V-C for a formal description): **(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.*

B. Attacks on BrowserID

Our analysis of BrowserID w.r.t. the above security properties revealed several attacks (as sketched next). We confirmed the attacks on the actual implementation and also reported them to Mozilla. The first two fixes proposed below have been adopted by Mozilla already and the others are currently under discussion at Mozilla.

1) Identity Forgery: There are two problems in Sideshow that lead to identity forgery attacks for Gmail addresses; analogously in BigTent with Yahoo email addresses.[5]

a) It is not checked if all requested attributes in the OpenID assertion are MACed, which allows for the following attack: A (web) attacker may choose any Gmail address to impersonate, say victim@gmail.com. He starts a BrowserID login with this email address. When he is then redirected to the OpenID URL at Gmail, he removes the email attribute from Sideshow's authentication request. The attacker authenticates himself at Gmail with his own account (say, attacker@gmail.com). Upon receipt of the OpenID assertion, he appends the email attribute with value victim@gmail.com and forwards it to Sideshow. The assertion is declared valid by Gmail since the MAC is correct (the email attribute is not listed in `openid.signed`). Since Sideshow does not require the email attribute to be in `openid.signed`, it accepts the OpenID assertion, considers the attacker's session to be authenticated for victim@gmail.com, and issues UCs for this address to the attacker. This violates Condition **(A)**.

[5]See https://bugzilla.mozilla.org/show_bug.cgi?id=920030 and https://bugzilla.mozilla.org/show_bug.cgi?id=920301.

b) Sideshow uses the first email address in the OpenID assertion (based on the attribute type information), which is not necessarily the MACed email address. This allows for an attack similar to the above, except that the attacker does not need to change Sideshow's authentication request but only prepends the victim's email address to the OpenID assertion in an additional attribute.

Proposed fix. Sideshow/BigTent must ensure to use the correct and MACed attribute for the email address.

2) Login Injection Attack: During the login process, the origin of the response postMessage (26 in Figure 3), which contains the CAP, is not checked. An attacker (e.g., in a malicious advertisement iframe within RP-Doc), can continuously send postMessages to the RP-Doc with his own CAP in order to log the user into his own account. This attack violates Condition **(B)**.[6]

Proposed fix. To fix the problem, the sender's origin of the postMessage 26 must be checked to match LPO.

3) Key Cleanup Failure Attack: When LD creates a key pair (15 in Figure 3), it stores the keys in the localStorage (even in ephemeral sessions). When a user quits a session (e.g, by clicking on RP's logout button and closing the browser) the key pair (and the UC) remain in the localStorage, unlike session cookies. Hence, users of shared terminals can read the localStorage (in our model, a `CLOSECORRUPT` allows an attacker to do this) and then, using the key pair and the UC, create valid CAPs to log in at any RP under the identity of the previous user, which violates Condition **(A)**.[7]

Proposed fix. We propose to use the localStorage for this data only in non-ephemeral sessions.

4) Cookie Cleanup Failure Attack (for the case of secondary IdP only): The LPO session cookie is not deleted when the browser is closed, even in ephemeral sessions and even if a user logged out at RP beforehand. (In our model, if the attacker issues a `CLOSECORRUPT`, he can therefore still access the LPO session cookie.) Hence, another user of the same browser could request new UCs for *any* ID registered at LPO for that user, and hence, log in at any RP under this ID, which violates Condition **(A)**.[8]

Proposed fix. In ephemeral sessions, LPO should limit the cookie lifetime to the browser session.

C. Analysis of BrowserID with sIdP

We now present our formal model and analysis of BrowserID with sIdP. We consider ephemeral sessions (the default), which are supposed to last until the browser is closed. We assume that users are already registered at LPO, i.e., they have accounts at LPO with one or more email addresses registered in each account.

[6]See https://bugzilla.mozilla.org/show_bug.cgi?id=868967

[7]See https://github.com/mozilla/browserid/issues/3770

[8]See https://github.com/mozilla/browserid/issues/3769

More specifically, we first model the BrowserID system as a web system (in the sense of Section III), then precisely formalize the security properties already sketched in Section V-A in this model, and finally prove, for the BrowserID model with the fixes proposed in Section V-B applied (otherwise the proof would not go through), that these security properties are satisfied.

1) Our BrowserID Model: 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 described in what follows.

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$ consists of the (network) attacker process $\mathtt{attacker}$, the web server for LPO, a finite set B of web browsers, and a finite set RP of web servers for the relying parties, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \{\mathsf{LPO}\}$, $\mathsf{Web} := \emptyset$, and $\mathsf{Net} := \{\mathtt{attacker}\}$. DNS servers are assumed to be dishonest, and hence, are subsumed by $\mathtt{attacker}$. More details on the processes in $\mathcal{W}$ are provided below.

The set $\mathcal{N}$ of nonces is partitioned into three sets, an infinite set $N^{\mathcal{W}}$, an infinite set K_{private}, and a finite set $\mathsf{Secrets}$. The set $N^{\mathcal{W}}$ is further partitioned into infinite sets of nonces, one set $N^p \subseteq N^{\mathcal{W}}$ for every $p \in \mathcal{W}$.

The set IPs contains for LPO, $\mathtt{attacker}$, every relying party in RP, and every browser in B one address each. By $\mathtt{addr}$ we denote the corresponding assignment from a process to its address. The set Doms contains one domain for LPO, one for every relying party in RP, and a finite set of domains for $\mathtt{attacker}$. Each domain is assigned a fresh private key (a nonce). Additionally, LPO has a fresh signing key k^{LPO}, which it uses to create UCs.

Each browser $b \in \mathsf{B}$ owns a finite set of secrets ($\subseteq \mathsf{Secrets}$) for LPO and each secret is assigned a finite set of email addresses (IDs) of the form $\langle name, d\rangle$, with $name \in \mathbb{S}$ and $d \in \mathsf{Doms}$, such that browsers have disjoint sets of secrets and secrets have disjoint sets of IDs. An *ID i is owned by a browser b* if the secret associated with i belongs to b.

The set $\mathcal{S}$ contains four scripts, with their string representations defined by script: the honest scripts running in RP-Doc, CIF, and LD, respectively, and the malicious script R^{att} (see below for more details).

The set E_0 contains only the trigger events specified in Definition 6.

Before we specify the processes in $\mathcal{W}$, we first note that a UC uc for a user u with email address i and public key (verification key) $\mathrm{pub}(k_u)$, where k_u is the private key (signing key) of u, is modeled to be a message of the form $uc = \mathsf{sig}(\langle i, \mathrm{pub}(k_u)\rangle, k^{\mathsf{LPO}})$, with k^{LPO} as defined above. An IA ia for an origin o (e.g., $\langle \mathtt{example.com}, \mathsf{S}\rangle$) is a message of the form $ia = \mathsf{sig}(o, k_u)$. Now, a CAP is of the form $\langle uc, ia\rangle$. Note that the time stamps are omitted both from the UC and the IA. This models that both certificates are valid indefinitely. In reality, as explained in Section IV, they are valid for a certain period of time, as indicated by the time stamps. So our modeling is a safe overapproximation.

We are now ready to define the processes in $\mathcal{W}$ as well as the scripts in $\mathcal{S}$ in more detail. We note that in our full version [14], we provide a detailed formal specification of the processes and scripts in the style of Algorithm 1.

All processes in $\mathcal{W}$ contain in their initial states all public keys and the private keys of their respective domains (if any). We define $I^p = \{\mathsf{addr}(p)\}$ for all $p \in \mathsf{Hon}$.

Attacker. The attacker process is a network attacker (see Section III-A), who uses all addresses for sending and listening. All parties use the attacker as a DNS server.

Browsers. Each $b \in \mathsf{B}$ is a web browser as defined in Section III-D. The initial state contains all secrets owned by b, stored under the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$ of LPO; sts is $\langle \mathsf{dom}(\mathsf{LPO}) \rangle$.

LPO. The initial state of LPO contains its signing key k^{LPO}, all secrets in Secrets and the corresponding IDs. The definition of R^{LPO} closely follows the description of LPO in Section IV-D. Sessions of LPO expire non-deterministically. UCs are signed using k^{LPO}.

Relying Parties. A relying party $r \in \mathsf{RP}$ is a web server. The definition of R^r follows the description in Section IV and the security considerations in [23].[9] RP answers any `GET` request with the script `script_RP_index` (see below). When receiving an HTTPS `POST` message, RP checks (among others) if the message contains a valid CAP. If successful, RP responds with an *RP service token for ID* i of the form $\langle n, i \rangle$, where $i \in \mathsf{ID}$ is the ID for which the CAP was issued and n is a freshly chosen nonce. The RP r keeps a list of such tokens in its state. Intuitively, a client having such a token can use the service of r for ID i.

BrowserID Scripts. The set $\mathcal{S}$ consists of the following scripts: R^{att}, $script\_RP\_index$, $script\_LPO\_cif$, and $script\_LPO\_ld$, with their string representations being `att_script`, `script_RP_index`, `script_LPO_cif`, and `script_LPO_ld`. The latter two scripts (issued by LPO) are defined in a straightforward way following the implementation outlined in Section IV. The script $script\_RP\_index$ (issued by RP) also includes the script that is (in reality) loaded from LPO. In particular, this script creates the CIF and the LD iframes/subwindows, whose contents (scripts) are loaded from LPO.

2) Formal Security Properties: The security properties for BrowserID, informally introduced in Section V-A, are formally defined as follows. First note that every RP service token $\langle n, i \rangle$ recorded in an RP was created by the RP as the result of a unique HTTPS `POST` request m with a valid CAP for ID i. We refer to m as the *request corresponding to* $\langle n, i \rangle$.

[9]Mozilla recommends to (1) protect against Cross-site Request Forgery (R^r checks the Origin header, which is always set in our model), (2) verify CAPs on the server (rather than in the browser), (3) check if the CAP is issued for the correct RP, and (4) verify SSL certificates.

Definition 7. *Let $\mathcal{BID}$ be a BrowserID web system. We say that $\mathcal{BID}$* is secure *if for every run ρ of $\mathcal{BID}$, every state (S_j, E_j) in ρ, every $r \in \mathsf{RP}$, every RP service token of the form $\langle n, i \rangle$ recorded in r in the state $S_j(r)$, the following two conditions are satisfied:*

***(A)** If $\langle n, i \rangle$ is derivable from the attackers knowledge in S_j (i.e., $\langle n, i \rangle \in d_{N^{\mathsf{attacker}}}(S_j(\mathsf{attacker}))$), then it follows that the browser owning i is fully corrupted in S_j, i.e., the value of $isCorrupted$ is* `FULLCORRUPT`.

***(B)** If the request corresponding to $\langle n, i \rangle$ was sent by some $b \in \mathsf{B}$ which is honest in S_j, then b owns i.*

3) Security of the Fixed System: We call a BrowserID web system $\mathcal{BID}$ with the fixes proposed in Section V-B a *fixed BrowserID web system*. We now obtain the following theorem, which says that such a system satisfies the security properties **(A)** and **(B)**.

Theorem 1. *Let $\mathcal{BID}$ be a fixed BrowserID web system. Then, $\mathcal{BID}$ is secure.*

The complete proof with all details is provided in the full version of this paper [14]. Due to space limitations, here we only provide a very rough sketch of how security property **(A)** is proved: We assume that **(A)** is not satisfied and lead this to a contradiction. To do so, we first prove a sequence of (twelve) lemmas. To provide an example, in one lemma we show that in every run of $\mathcal{BID}$ if a CAP c is created by $script\_LPO\_ld$, then the origin for which c is issued is the origin of the script that receives the postMessage containing c (26 in Figure 3). Using these lemmas, we distinguish between two (main) cases to lead the assumption that **(A)** does not hold to a contradiction: the attacker, in state S_j, knows (or does not know) the key used to encrypt the service token $\langle n, i \rangle$ recorded in and issued by r.

VI. Related Work

Early work in the direction of formal web security analysis includes work by Kerschbaum [21], in which a Cross-Site Request Forgery protection proposal is formally analyzed using a simple model expressed using Alloy, a finite-state model checker [19].

In seminal work, Akhawe et al. [2] initiated a more general formal treatment of web security. Again the model was provided in the Alloy modeling language. Inspired by this work, Bansal et al. [5], [6] built the WebSpi model for the web infrastructure, which is encoded in the modeling language (a variant of the applied pi-calculus [1]) of ProVerif, a specialized tool for cryptographic protocol analysis [8]. Both models have successfully been applied to find attacks in standards and web applications.

We see our work as a complement to these models: On the one hand, the above models support (fully) automated analysis. On the other hand, our model is much more comprehen-

sive and accurate, but not directly suitable for automation.[10] We think that, similarly to the area of cryptography, both approaches, automated analysis and manual analysis, are very valuable. Clearly, it is highly desirable to push automated analysis as much as possible, given that manual proofs are laborious and error-prone. Conversely, automated approaches may miss important problems due to the less accurate models they consider. Moreover, a "service" more comprehensive and accurate models provide, even if they are manually driven, is that they summarize and condense relevant aspects in the various standards and specifications for the web. As such, they are an important basis for the formal foundation and discourse on web security and can serve as reference models (for tool-supported analysis, web security researchers, for developers of web technologies and standards, and maybe for teaching basic web security concepts).

The BrowserID system 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. Their analysis of BrowserID is not very detailed; only two rather trivial attacks are identified, for example, CAPs that are sent unencrypted can be replayed by the attacker to an RP. There is also work on the analysis of other web-based single sign-on systems, such as SAML-based single sign-on, OpenID, and OAuth (see, e.g., [3], [7], [11], [15], [17], [22], [26]–[29]). However, none of these works are based on a model of the web infrastructure.

In [16], [25], [26], [29], potentially problematic usage of postMessages and the OpenID interface are discussed. While very useful, these papers do not consider BrowserID or formal models, and they do not formalize security properties for web applications or establish formal security guarantees.

Bohannon and Pierce propose a formal model of a web browser core [9]. The scope and goal of the model is different to ours, but some mechanisms can be found in both models. Börger et al. present an approach for the analysis of web application frameworks, focussing on the server [10].

VII. Conclusion

We presented an expressive model of the web infrastructure and web applications, the most comprehensive model for the web infrastructure to date. 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.

In our case study, we analyzed the BrowserID system, found several very critical attacks, proposed fixes, and proved the fixed system for the case of secondary IdP case secure w.r.t. the security properties we specified. The analysis of this system is out of the scope of other models for the web infrastructure.

As for future work, it is straightforward to incorporate further features, such as subdomains, cross-origin resource sharing, and finer-grained settings for cookie paths and domains, which we have left out mainly for brevity of presentation for now. Our model could serve as a basis and a reference for automated approaches, where one could try to extend the existing automated approaches or develop new ones (e.g., based on theorem provers, where higher accuracy is typically paid by more interaction). Finally, BrowserID is being used by more and more web sites and it will continue to be an interesting object of study. An obvious next step is to analyze BrowserID for the case of primary IdPs. The model is already expressive enough to carry out such an analysis. We also plan to apply our model to other web applications and web standards.

[10]The tool-based models are necessarily tailored to and limited by constraints of the tools. For example, models for Alloy are necessarily finite state. Terms (messages) need to be encoded in some way as they are not directly supported. Due to the analysis method employed in ProVerif, the WebSpi model is of a monotonic nature. For instance, cookies and localStorage entries can only be added, but not deleted or modified. Also, the number of cookies per request is limited. Several features (that have been crucial for the analysis of BrowserID) are not supported by the tool-based models, including the precise handling of windows, documents, and iframes as well as cross-document messaging (postMessages), and the ability for an attacker to take over a browser after it has been closed. Dealing with such features in an automated tool is indeed challenging.

Acknowledgement

The first author is supported by the *Studienstiftung des Deutschen Volkes* (German National Academic Foundation).

References

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (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 *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 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 V. Shmatikov, editor, *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, pages 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 *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In D. A. Basin and J. C. Mitchell, editors, *Principles of Security and Trust - Second International Conference, POST 2013*, volume 7796 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2013.

[6] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 247–262. IEEE Computer Society, 2012.

[7] J. Bellamy-McIntyre, C. Luterroth, and G. Weber. OpenID and the Enterprise: A Model-Based Analysis of Single Sign-On Authentication. In *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2011*, pages 129–138. IEEE Computer Society, 2011.

[8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001.

[9] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 11–11. USENIX Association, 2010.

[10] E. Börger, A. Cisternino, and V. Gervasi. Contribution to a Rigorous Analysis of Web Application Frameworks. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012*, volume 7321 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012.

[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] Cross-Origin Resource Sharing - W3C Recommendation 29 January 2013. Available at http://www.w3.org/TR/2013/CR-cors-20130129/.

[13] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[14] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. Technical Report arXiv:1403.1866, arXiv, 2014. Available at http://arxiv.org/abs/1403.1866.

[15] T. Groß. Security Analysis of the SAML Single Sign-on Browser/Artifact Profile. In *19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 298–307. IEEE Computer Society, 2003.

[16] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. The emperor's new apis: On the (in)secure usage of new client side primitives. In *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP), 2010*, 2010.

[17] S. M. Hansen, J. Skriver, and H. R. Nielson. Using static analysis to validate the SAML single sign-on protocol. In C. Meadows, editor, *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005*, pages 27–40. ACM, 2005.

[18] HTML5, W3C Candidate Recommendation. Dec. 17, 2012.

[19] D. Jackson. Alloy: A new technology for software modelling. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, page 20. Springer, 2002.

[20] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007*, pages 58–71. ACM, 2007.

[21] F. Kerschbaum. Simple cross-site attack prevention. In *Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.

[22] A. Kumar. Using automated model analysis for reasoning about security of web protocols. In R. H. Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012*, pages 289–298. ACM, 2012.

[23] Mozilla Identity Team. Persona. Mozilla Developer Network. Last visited May 1, 2013. https://developer.mozilla.org/en/docs/persona.

[24] OpenID Foundation website. http://openid.net.

[25] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[26] P. Sovis, F. Kohlar, and J. Schwenk. Security Analysis of OpenID. In *Sicherheit*, volume 170 of *LNI*, pages 329–340. GI, 2010.

[27] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security, CCS'12*, pages 378–390. ACM, 2012.

[28] 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.

[29] 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 *IEEE Symposium on Security and Privacy (S&P 2012), 21-23 May 2012, San Francisco, California, USA*, pages 365–379. IEEE Computer Society, 2012.

[30] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *32nd IEEE Symposium on Security and Privacy, S&P 2011*, pages 465–480. IEEE Computer Society, 2011.

[31] Web Storage - W3C Recommendation 30 July 2013. http://www.w3.org/TR/2013/REC-webstorage-20130730/.

[32] whatwg.org. Fetch. http://fetch.spec.whatwg.org/.