Browser Model for Security Analysis of Browser-Based Protocols

Thomas Groß[1], Birgit Pfitzmann[1], and Ahmad-Reza Sadeghi[2]

[1] IBM Zurich Research Lab, Rüschlikon, Switzerland
[2] Ruhr-University Bochum, Bochum, Germany
{tgr, bpf}@zurich.ibm.com
sadeghi@crypto.rub.de

Abstract. Currently, many industrial initiatives focus on web applications. In this context an important requirement is often that the user should only rely on a standard web browser. Hence the underlying security services also rely solely on a browser for interaction with the user. Browser-based identity federation is a prominent example of such a service. Very little is still known about the security of browser-based protocols, and they seem at least as error-prone as standard security protocols. In particular, standard web browsers have limited cryptographic capabilities and thus new protocols are used. Furthermore, these protocols require certain care by the user in person, which must be modeled. In addition, browsers, unlike normal protocol principals, cannot be assumed to do nothing but execute the given security protocol.

In this paper, we lay the theoretical basis for the rigorous analysis and security proofs of browser-based protocols. We formally model web browsers, secure browser channels, and the security-relevant browsing behavior of a user as automata. As a first rigorous security proof of a browser-based protocol we prove the security of password-based user authentication in our model. This is not only the most common stand-alone type of browser authentication, but also a fundamental building block for more complex protocols like identity federation.

1 Introduction

Browser-based services have received increasing attention in the last years. The idea is simple: users should be able to access the services by only using a standard web browser, which offers a set of basic functionalities. Thus the users send their requests for services as well as receive and view the results by means of the browser. This enables cost-efficient deployment of applications without specific user education. The requirement on such services not to need any special client software is also called *zero-footprint*. Consequently, the underlying security services must also be zero-footprint. Therefore, security services only use a standard browser for user authentication and for retaining a secure channel with the user. They may also request additional security-relevant attributes about the users and third-party confirmation via the browser. We first discuss how non-browser protocols establish such security services and how their methods differ from browser-based protocols. Then we consider research on browser-based protocols in the prominent area of identity federation and point out challenges in establishing rigorous security proofs, before summarizing our contributions.

Establishing Secure Channels. The security services for browser-based protocols mostly focus on establishing mutually authenticated secure channels. The typical approach in other security protocols is to perform a key exchange, based on local master keys, master keys shared with a third party, or public-key certificates, and to subsequently use the exchanged key to secure the communication. A large body of literature on such protocols exists. A seminal work were the Needham-Schroeder protocols [28], although a vulnerability in one protocol was later found by Lowe [20]. Tool-supported proofs were initiated in [25,15,23], based on abstractions of cryptographic primitives introduced in the Dolev-Yao model [6]. Recent tool-supported proofs concentrate on using existing general-purpose model checkers and theorem provers, first in [21,26,4,7,31]. Cryptographic proofs of key-exchange and authentication protocols were initiated by Bellare and Rogaway [1]. Cryptography also added interesting additional properties to pure authentication, e.g., see [17]. Modeling secure channels by a comparison to ideal secure channels, a technique that we will use for the underlying secure channels below, was introduced in [40,35,2]. Analyses specifically for SSL and TLS, and thus close to an underlying mechanism used in browsers, were made in [41,27,32,18].

However, standard browsers simply do not execute most of these protocols. Instead, a browser establishes a server-side authenticated secure channel leveraging a public-key certificate of the server and provides client authentication by other means. The only exception to this method would be SSL or TLS channels with client certificates for 2-party authentication. However, this is not considered truly zero-footprint because the users would have to obtain the certificates. Also, the method would not allow a user to easily use different browsers at different times. Thus it is very rarely used, and not used at all as a basis in larger browser-based security protocols. Hence browser-based protocols are different from all protocols for which prior security proofs exist.

Identity Federation. A prominent area of browser-based security protocols is identity federation, which aims at linking a user's (otherwise) distinct identities at several locations. The benefit is that the involved organizations can reduce user management costs, such as the cost of password helpdesks and user registration and deletion. In this area, concrete and complex browser-based security protocols were proposed, e.g., Microsoft's Passport [24], the Security Assertion Markup Language (SAML) standardized by OASIS [29], the Shibboleth project for university identity federation [3], the Liberty Alliance project [19], and WS-Federation [13,14]. Several papers discussed vulnerabilities of such protocols, in particular for Passport [16], the Liberty enabled-client protocol [37], and a SAML profile [10]. Others discussed privacy design principles and details [36,33,34,38]. Basic browser-based authentication without federated identity management is discussed in [9]. As far as the vulnerabilities found were removable security problems (in contrast to fundamental limitations of the browser-based protocol class or matters of taste like privacy), they were removed in the next version of the protocols. However, past experience in protocol design has shown that incorporating countermeasures against known attacks does not guarantee to eliminate all vulnerabilities. Hence it is desirable to devise security proofs.

Proving Browser-Based Protocols. It is not trivial to apply previous security proof techniques, both cryptographic techniques and formal-methods techniques, to browser-based protocols. The primary reason is that a browser represents a new party with its own, predefined behavior that impacts the security of the protocols executed across it. In usual security protocols, principals are assumed to execute precisely the security protocol under consideration (unless they are corrupted). A browser, in contrast, reacts on a number of predefined messages, adds information to responses automatically, and stores information such as histories in places that cannot always be assumed secure, e.g., if the browser is in an Internet kiosk. For instance, one of the SAML problems found in [10] is based on the HTTP Referer tag, i.e., a browser feature that is not mentioned at all on the level of the SAML protocol. Another usual issue is that browser-based protocols use a multitude of names for a principal, while other protocols typically assume a one-to-one mapping; for instance, there are URL addresses, identities used in SSL certificates, and identities used in higher protocols. It is easy to forget some name comparisons in protocols and thus to enable man-in-the-middle attacks. All this means that a detailed and rigorous browser model is a prerequisite for convincing security proofs of browser-based protocols, and no such model exists so far. For the resulting model, we currently assume that a real browser does not perform additional actions, because for most security protocols arbitrary additional actions could destroy the security. This could be replaced by more precise assumptions on forbidden additional actions in the future.

Another important special aspect is that due to the limited capabilities of browsers, the user at the browser is an active participant and certain assumptions must be made about the user, e.g., that the user verifies that a secure channel to a trusted server is used before entering an important password.

Our Contribution. In this paper, we lay the theoretical basis for research in this area by modeling the major building blocks for browser-based protocols. We present a rigorous and abstract model for a standard web browser as a principal for browser-based protocols. While our model is still extensible – in particular we do not model cookies and scripting but assume a browser with these features turned off – we believe that we have captured the major explicit and implicit browser features that play a role in typical browser-based protocols. In addition, we model the security-relevant browsing behavior of a user, i.e., a machine that implements the explicit constraints on a user that are needed for protocol proofs, but still allows arbitrary behavior apart from that. Furthermore, we model browser channels in order to capture, in particular, the naming issues across multiple protocol layers.

As a first security lemma for a browser-based protocol in our model, we study the security of the initial authentication of the user behind a browser by a password. Initial user authentication is an integral part of all browser-based protocols, and passwords are the standard technique used in the zero-footprint scenario.

A first step in the direction of proofs of browser-based protocols was taken in [11]. There, however, we only modeled exactly those parts of the user and browser behavior that we concretely needed for the protocol, and made assumptions that other things would not happen. These assumptions were made top-down for the needs of the protocol rather than bottom-up from a browser and user model. In this paper, we lay the bottom-up groundwork for such assumptions.

2 Notation

General Notation. We use a straight font for $\mathsf{constants}$, including $\mathsf{constant\ Sets}$ and Types, $\mathsf{functions}$, and $\mathsf{predicates}$, where Types are predefined constant sets. We use italics for *variables* and variable *Sets*. Let Σ be an alphabet without the symbols {"ϵ", "!", "?", "$^{\triangleleft}$", "[", "]", "//" }. Then Σ^* is the set of strings over Σ where ϵ denotes the empty string and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. For a set S, $\mathcal{P}(S)$ denotes the powerset of S and S^* the set of finite sequences over S. We define $S.\mathsf{add}\,(x)$ as $S := S \cup \{x\}$ and $S.\mathsf{remove}\,(x)$ as $S := S \setminus \{x\}$. Assignment by possibly probabilistic functions is written as $\leftarrow$. Assignment of a value to a tuple of variables means making correspondingly many projections; if one of these fails the entire assignment fails. We denote the set of URL host names, including protocol names such as "https", by $\mathsf{URLHost}$, the set of URL path names by $\mathsf{URLHostPath}$, and the set of URL host and path names by $\mathsf{URLHostPath}$. We write an address $adr \in \mathsf{URLHostPath}$ as a pair $(host, path)$ of a host name $host \in \mathsf{URLHost}$ and a path. The type $\mathsf{ChType} := \{\mathsf{secure}, \mathsf{insecure}\}$ contains the channel types available.

Automata. We represent our machines such as the browser model as I/O automata, in other words finite-state machines with additional variables. This is a very usual basis for specifying participants in distributed protocols; the first specific use for security is in [22]. Specifically we use the automata model proposed in [35], which has a well-defined realization by probabilistic interactive Turing machines and is therefore linked to more detailed cryptographic considerations where those become necessary in multi-layer proofs. In the following we give a brief overview of this machine model (see also Figure 2). Machines may have multiple fixed connections to other machines organized by means of uni-directional *ports*. We define two types of *simple ports* for message transmission: $\mathsf{n}?$ is an input port with name n and $\mathsf{n}!$ an output port, respectively. The machine model connects simple ports $\mathsf{n}?$ and $\mathsf{n}!$ with the same name n and opposite direction; these are called *complement* ports. We call ports without such a complement *free* ports. We define a *clock port* $p = (n,^{\triangleleft}, d) \in \Sigma^+ \times \{^{\triangleleft}\} \times \{!, ?\}$ as a port that schedules the connection between simple ports $\mathsf{n}?$ and $\mathsf{n}!$ with same name n, or is free itself if this connection does not exist.

A *machine* M is defined as a tuple $\mathsf{M} = (name_{\mathsf{M}}, Ports_{\mathsf{M}}, Vars_{\mathsf{M}}, States_{\mathsf{M}}, \delta_{\mathsf{M}}, Ini_{\mathsf{M}}, Fin_{\mathsf{M}})$ of a name $name_{\mathsf{M}} \in \Sigma^+$, a finite sequence $Ports_{\mathsf{M}}$ of ports, a finite sequence $Vars_{\mathsf{M}}$ of variables, a set $States_{\mathsf{M}} \subseteq \Sigma^*$ of major states, a probabilistic state-transition function δ_{M}, and sets $Ini_{\mathsf{M}}, Fin_{\mathsf{M}} \subseteq States_{\mathsf{M}}$ of initial and final states. The inputs are tuples $I = (I_i)_{i=1,\ldots,|\mathsf{in}(Ports_{\mathsf{M}})|}$, where $I_i \in \Sigma^*$ is the input for the i-th in-port, $\mathsf{in}(Ports_{\mathsf{M}})$ is the input ports of the machine and $|\mathsf{in}(Ports_{\mathsf{M}})|$ denotes the number of the input ports. Analogously, the outputs are tuples $O = (O_i)_{i=1,\ldots,|\mathsf{out}(Ports_{\mathsf{M}})|}$. The empty word, ϵ, denotes "no in- or output", respectively. The value assignments of variables are tuples $V = (V_i)_{i=1,\ldots,|Vars_{\mathsf{M}}|}$, where $V_i \in \Sigma^*$ is the value for the i-th variable in the sequence $Vars_{\mathsf{M}}$. In the following, we usually say "state" for "major state"; however, the state transition function changes the overall state consisting of the major state and the variables.

We define the state transition function δ_{M} of a machine M using a notation analogous to UML state diagrams [30], see Figure 1. This is a concise and user-friendly definition method that enables easy graph analysis of the command and information

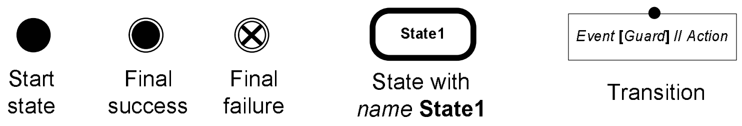


Fig. 1. Key to the state diagrams

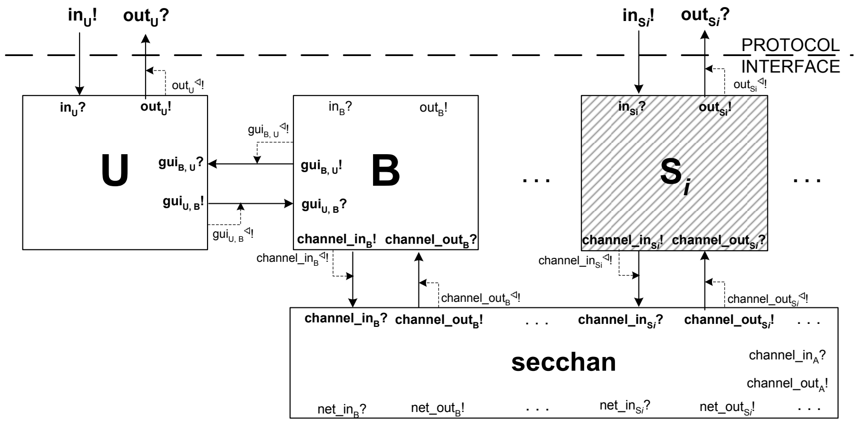

in_U! out_U?
in_Si! out_Si?
PROTOCOL INTERFACE
out_U◁!
out_Si◁!
U
B
S_i
secchan
in_U? out_U!
gui_B,U? gui_U,B!
in_B? out_B!
gui_B,U! gui_U,B?
gui_B,U◁! gui_U,B◁!
channel_in_B! channel_out_B?
in_Si? out_Si!
channel_in_Si! channel_out_Si?
channel_in_B◁! channel_out_B◁!
channel_in_Si◁! channel_out_Si◁!
channel_in_B? channel_out_B! . . . channel_in_Si? channel_out_Si! . . .
channel_in_A? channel_out_A!
net_in_B? net_out_B! . . . net_in_Si? net_out_Si! . . .

Fig. 2. System architecture for browser-based protocols with a browser B, a user U, servers S_i, channel abstraction secchan and their ports

flow in security proofs. We define a *transition* in a state diagram as an arrow from a state s to a state s' with label $Event[Guard]//Action$, where $Event$ is a sequence of non-empty inputs to the input ports, $Guard$ is a predicate over $Event$ and the machine's current variable allocation V, and $Action$ specifies the computations and outputs of the transition.

System Overview for Browser-based Protocols. Figure 2 gives an overview of the automata in our model. We call the generic browser machine B, the user machine that implements the minimum assumptions on secure user behavior U, and the machine that models the behavior of secure channels as implemented within HTTPS (see [39]) by secchan. To analyze and prove browser-based security protocols one complements these general-purpose machines with one or more server machines, here denoted by S_i, that jointly execute the browser-based protocol. Furthermore, one configures the user machine U with suitable initial information about trusted parties.

3 Ideal Web Browser B

In this section, we describe our model of a web browser. Before the actual definitions, we give an overview of real web browsers.

A web browser acts as the client in transactions of the Hypertext Transfer Protocol (HTTP) [8]. A browser acts on behalf of one single user in a browsing session. However, it may display multiple windows that render different HTTP transactions in parallel. We will model a window by a window identifier wid in the communication between U and B; however, the window management of the browser machine is omitted for brevity.

A browser accepts user inputs specifying addresses and retrieves and renders the content associated with these addresses or error messages. It also renders the status of the channel to the server and, if a secure channel is used, the identity of the server. Furthermore, it initiates dialogs with the user to negotiate changes of the channel state, verify a server's authentication or request for user authentication.

HTTP is a client-server protocol in the application layer of the TCP/IP protocol stack. In order to initiate an HTTP transaction, a browser establishes a connection to a server; here it may leverage various transport protocols, in particular TCP/IP, SSL 3.0, or TLS1.0 [5]. Having established a channel, the browser issues an HTTP request to the server. Such a request specifies the resource that the browser intends to retrieve and may contain additional parameters. The server evaluates the request and issues a response using the same channel. We call such an interaction an HTTP transaction. In principle, browsers do not need to hold state beyond such a single transaction. However, real web browser do hold persistent state, e.g., a cache and a browsing history, and they let a transaction influence the subsequent one. Our browser model reflects this behavior.

The most important types of HTTP requests are GET and POST requests, used by the browser to retrieve and send data, respectively. Servers may not only respond with content but also instigate a behavior change in the browser. In particular, HTTP responses with scripted form POST and redirect messages direct the browser to another address of the server's choice. Another HTTP response asks for user authentication by a username-password pair. We model these HTTP messages as abstract formats, i.e., we do not model how the bitstrings are parsed. We focus on the mentioned subset of HTTP messages and their core parameters, because these are the ones that are actively used in browser-based protocols or may have impact on their security.

An important aspect of real web browsers is that they do more than browser-based protocols typically intend. Most prominent is the problem of information flow. On the one hand, this may occur within HTTP requests, e.g., by the HTTP Referer tag. On the other hand, persistent state such as history, cache and password storage provide data flow to the underlying operating system and thus potentially to other parties. Hence the user's log-off from a browser session that removes browser state is also security-relevant. We dedicate Section 3.4 to information-flow aspects.

At present, we do not consider cookies and scripts as many browser-based protocols do not use them directly and it is possible to switch them off.

In Section 3.1 we define the interface of the browser machine B, i.e., its ports and the syntax of permitted in- and output messages. Details of the abstract HTTP messages are treated in Section 3.2. Section 3.3 defines the variables of the browser model. In Section 3.4 we define functions governing the information-flow properties. Finally, Section 3.5 describes the transition function.

3.1 Interface of Browser Machine B

The ports of B were already shown in Figure 2. We now discuss the messages exchanged over these ports. We list them all in Table 3 of Appendix A. Here we only explain them as far as it is useful to understand the upcoming state diagram.

The ports $\mathsf{gui}_{\mathsf{U,B}}?$ and $\mathsf{gui}_{\mathsf{B,U}}!$ model the browser's user interface. The input messages enter_address, trigger_address and submit_form issue a request for an address to B. Here enter_address represents an input in the browser's address field, trigger_address models clicking a link and submit_form defines the submission of an HTML form. The output messages established and error inform the user of the channel status. The message channel_change notifies the user of a change of the security level of the channel. The browser uses request_uauth and authenticate in the password-based user authentication dialog. The remaining messages organize a certificate verification dialog with the user. The security of browser-based protocols builds upon the browser machine reliably presenting secure channels and the server identity sid to their users. Thus, if an HTTP transaction uses a secure channel, B includes the channel's server identity in each message to U. The user machine U confirms the server identity in each message to B.[1]

The ports $\mathsf{channel\_out}_{\mathsf{B}}?$ and $\mathsf{channel\_in}_{\mathsf{B}}!$ connect the browser to the underlying channel abstraction secchan. We introduce the ports $\mathsf{self}_{\mathsf{B}}!$ and $\mathsf{self}_{\mathsf{B}}?$ to reduce the complexity of the state diagrams. Using these ports, we allow the browser to delegate trigger_address and submit_form commands to itself and to treat them on the same command path as the user inputs. The ports $\mathsf{in}_{\mathsf{B}}?$ and $\mathsf{out}_{\mathsf{B}}!$ model information flow to the operating system and may be connected to a higher protocol layer or the adversary. We discuss in Section 3.4 how B explicitly leaks information about its state. Loosely speaking, upon an input do_leak at $\mathsf{in}_{\mathsf{B}}?$ the browser outputs its full persistent state to $\mathsf{out}_{\mathsf{B}}!$.

3.2 Abstract HTTP Messages

The interface of the secure channel abstraction secchan allows the browser to send very general messages to other machines, but correct browsers only send HTTP messages and only accept messages parsable according to HTTP.

We model an HTTP GET request by an abstract message with format $\mathsf{GET}(path, query, login, info\_leak)$. Here $path \in \mathsf{URLPath}$ denotes the path of the address to be retrieved and $query \in \Sigma^*$ the query string of the URL. The parameter $login \in \Sigma^*$ contains the credentials of a password-based user authentication, including the account name. The parameter $info\_leak$ is a list of name-value pairs from $\Sigma^* \times \Sigma^*$. It models potential additional parameters, e.g., the preceding address in the HTTP Referer tag. We discuss them specifically under the aspect of information flow in Section 3.4. For HTTP POST requests we define $\mathsf{POST}(path, query, login, info\_leak)$ analogously.

We now proceed to describe the HTTP responses. The abstract messages $\mathsf{Page}(m, close, nocache)$ and $\mathsf{Error}(m, close)$ model HTTP 200 OK responses and HTTP 40x error responses. Both contain a page $m \in \Sigma^*$ as the payload and a flag $close \in \mathsf{Bool}$ that directs the browser to close the underlying channel or to keep it alive for further HTTP transactions. This parameter models the token close of the Connection header of HTTP1.1 [8]. The parameter $nocache \in \mathsf{Bool}$ of Page models the cache-response directive nostore of HTTP1.1, which forces a browser not to store any part of this response and the request that elicited it.

[1] We only model that the user sees the server identity, not a channel identifier, because he or she will not notice if a channel is interrupted. Usually, however, a user can distinguish different channels with one partner by different windows.

The abstract message $\mathsf{Redirect}(adr, path, query, close)$ models a redirect (HTTP 302 or 303) to $adr/path?querystring$, where $querystring$ is an encoding of the abstract $query \in \Sigma^*$. Here $adr \in \mathsf{URLHost}$ and $path \in \mathsf{URLPath}$. Similarly, $\mathsf{POSTForm}(adr, path, query, close, nocache)$ models a form containing a script that will POST a message whose body encodes the abstract $query$ to the address $adr/path$. The parameters $close$ and $nocache$ are defined as above. In consequence of both messages the browser establishes a channel to the address adr and then sends $path$ and $query$ over that channel. The channel type is implied by the HTTP protocol name "http" or "https" in adr.

The abstract message $\mathsf{Authenticate}()$ queries the browser for a user authentication.

3.3 Variables of Browser Machine B

We now define the browser's variables $Vars_{\mathsf{B}}$. We distinguish *volatile* and *persistent* variables. Volatile variables belong to one HTTP transaction and are deleted in the final state of this transaction, while persistent variables survive individual transactions. We describe the variables in more detail in the full version of this paper [12].

We start by describing the volatile variables. The variable adr contains the address to be retrieved in the given HTTP transaction. It implies the value of $host$ and ch_type. The value of ch_type specifies the type of the channel the browser establishes to the server with hostname $host$. The variable $method$ contains the HTTP method used in this HTTP transaction, and $source\_uri \in \mathsf{Bool}$ states whether the entity issuing the request for adr has a URI of its own. This implies whether a Referer Tag is included in the request. The variable $form$ contains a form compiled from user inputs given by the variable $form\_in$ and a form obtained in the preceding HTTP transaction. The variable $ch \in \mathsf{Channel}$ contains the browser's local representation of a channel established to a server, i.e., the data the browser has acquired about the channel. An element of Channel is a tuple $(cid, host, sid, type, free)$ from the domain $\mathbb{N} \times \mathsf{URLHost} \times \Sigma^* \times \mathsf{ChType} \times \mathsf{Bool}$. Here cid is a channel identifier, $host$ contains the hostname of the server to which the browser channel is connected, and sid names the server's identity in a secure channel and is ϵ for an insecure channel. The element $type$ represents the channel type (secure or insecure). The element $free$ is used to organize the reuse of existing channels and flags that the channel is currently not associated to a HTTP transaction. The variable m contains the payload of an HTTP response, whereas variable $store$ flags the user's decision whether to store login data in the browser's state. The variable $auto\_req$ determines whether the browser issues the following request automatically to itself.

The first two persistent variables exist for each window: wid is the window identifier, and $prev\_run = (ch\_type, adr, form)$ contains data about the preceding HTTP transaction in this window: the channel type, the address retrieved, and the structure of an HTML form together with hidden value fields already included in the form. The other persistent variables are global for B. The sequence $Channels$ contains representations of type Channel of all channels the browser has established. The set $UAuth$ contains a user's login information the user decided to store in the browser's state, $History$ is a sequence of addresses successfully retrieved by the browser, and $Cache$ models the browser cache as a sequence of pairs of addresses and page contents retrieved from these addresses.

3.4 Information Flow Functions

Even correct browsers produce information flow beyond the core parameters of HTTP requests and responses to both communication partners and the underlying operating system. As this information flow may lead to vulnerabilities in browser-based security protocols, we model it explicitly.

We already provided the parameter $info\_leak$ for additional information in abstract HTTP requests. For generating the content of this parameter, we define a function $\mathsf{leak2server}()$. It implicitly works on the current variables of the given window and browser and computes a list of name-value pairs of information to be disclosed. A real browser primarily sends such information in HTTP header tags such as Referer, From or $\mathsf{Accept\_Language}$. However, most of these tags do not contain data that we modeled. Hence our default implementation of $\mathsf{leak2server}$ only includes the Referer tag, which contains the preceding address if the request was issued by an entity that has a URI of its own: $\mathsf{leak2server}() = (\mathsf{Referer}, prev\_run.adr)$ if $source\_uri$, else ϵ.

Another potential for information flow exists from the persistent variables, which are stored in the underlying operating system. This introduces security and privacy risks especially in kiosk scenarios. We already described that we model leakage of these variables by an input $\mathsf{do\_leak}$ at port $\mathsf{in_B}?$. Upon this input, the browser outputs its entire persistent state as a string $info$ in a message $\mathsf{leak}(info)$ at port $\mathsf{out_B}!$. These ports are free by default, so that the adversary can connect to them. Alternatively, they may be made so-called specified ports, which define the interface to a higher protocol. This allows for flexibility in the assumptions about the security of the persistent variables.

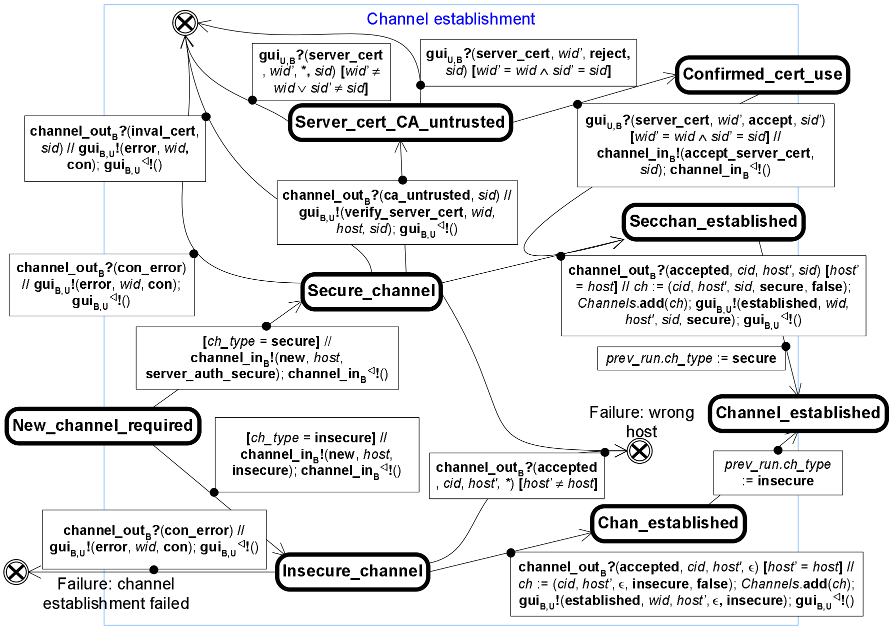

Channel establishment
gui_U,B?(server_cert, wid', *, sid) [wid' ≠ wid ∨ sid' ≠ sid]
gui_U,B?(server_cert, wid', reject, sid) [wid' = wid ∧ sid' = sid]
Confirmed_cert_use
Server_cert_CA_untrusted
channel_out_B?(inval_cert, sid) // gui_B,U!(error, wid, con); gui_B,U◁!()
gui_U,B?(server_cert, wid', accept, sid') [wid' = wid ∧ sid' = sid] // channel_in_B!(accept_server_cert, sid); channel_in_B◁!()
channel_out_B?(ca_untrusted, sid) // gui_B,U!(verify_server_cert, wid, host, sid); gui_B,U◁!()
Secchan_established
channel_out_B?(con_error) // gui_B,U!(error, wid, con); gui_B,U◁!()
Secure_channel
channel_out_B?(accepted, cid, host', sid) [host' = host] // ch := (cid, host', sid, secure, false); Channels.add(ch); gui_B,U!(established, wid, host', sid, secure); gui_B,U◁!()
[ch_type = secure] // channel_in_B!(new, host, server_auth_secure); channel_in_B◁!()
prev_run.ch_type := secure
New_channel_required
Failure: wrong host
Channel_established
[ch_type = insecure] // channel_in_B!(new, host, insecure); channel_in_B◁!()
channel_out_B?(accepted, cid, host', *) [host' ≠ host]
prev_run.ch_type := insecure
Chan_established
channel_out_B?(con_error) // gui_B,U!(error, wid, con); gui_B,U◁!()
Insecure_channel
Failure: channel establishment failed
channel_out_B?(accepted, cid, host', ϵ) [host' = host] // ch := (cid, host', ϵ, insecure, false); Channels.add(ch); gui_B,U!(established, wid, host', ϵ, insecure); gui_B,U◁!()

Fig. 3. Channel establishment phase of an HTTP transaction

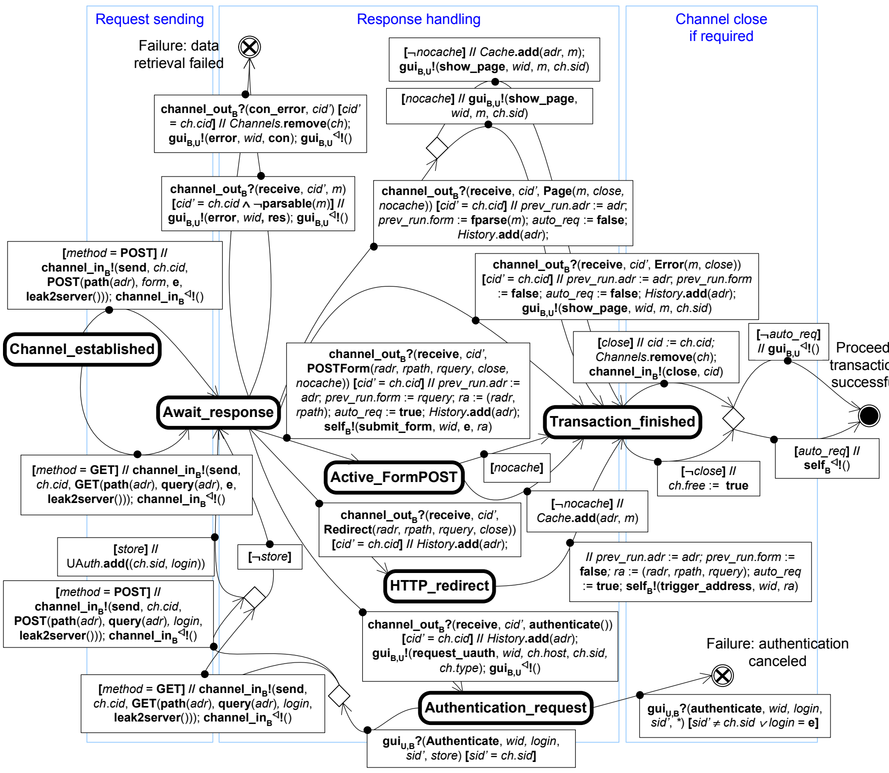

Request sending
Response handling
Channel close if required
Failure: data retrieval failed
Channel_established
Await_response
Active_FormPOST
HTTP_redirect
Transaction_finished
Authentication_request
Proceed: transaction successful
Failure: authentication canceled
[nocache]
[¬store]
[store] // UAuth.add((ch.sid, login))
[¬nocache] // Cache.add(adr, m)
[close] // cid := ch.cid; Channels.remove(ch); channel_in_B!(close, cid)
[¬close] // ch.free := true
[¬auto_req] // gui_B,U◁!()
[auto_req] // self_B◁!()

Fig. 4. Request handling phase of an HTTP transaction

3.5 State-Transition Function

We now define the browser's state transition function δ_B. A browser handles several classes of user actions asynchronously, such as enter_address and log_off. Upon enter_address, trigger_address or submit_form the window with the corresponding window identifier *wid* starts a new HTTP transaction. If there exists an ongoing HTTP transaction, that one's state flow is exited. Upon a log_off command, the browser B exits all state diagrams of HTTP transactions and starts a log-off flow, which closes all channels and deletes the browser state. Figures 3 and 4 contain the main parts of the state diagram of a single HTTP transaction, i.e., an HTTP request-response pair. The start state typically corresponds to the inactive state of the browser window where the user views a page; the transaction is then triggered by the user selecting a link or directly inputting a new URL. The start state can also correspond to a user filling a form or to the middle of a redirect.

The browser begins with a *local negotiation* phase where it notifies its user U about the establishment of a new channel if the desired channel is of a different type than the previous one, e.g., insecure HTTP after secure HTTPS. If the user consents to the channel change, the browser procures a suitable channel. We allow the browser to reuse free channels, i.e., opened but not associated to an ongoing HTTP transaction, with the correct host and security level; this is the *channel reuse* phase. The *channel establishment* phase shown in Figure 3 contains the case that a new channel is needed. It first distinguishes whether the desired channel should be secure or insecure. Establishing an insecure channel is straightforward. Establishing a secure channel involves a certificate test and potential user interaction if the browser is in doubt about a certificate.

Figure 4 starts at State Channel_established from Figure 3 and handles an HTTP request and response. In the *request sending* phase, B issues an HTTP request as a GET or POST according to the $method$ of the initial user input and enters State Await_response, where it expects an HTTP response from the server. In the *response handling* phase, B handles different abstract response types: a normal answer Page, an Error, a Redirect, a scripted POST FormPOST, or an authentication request Authenticate. The latter leads to a user interaction in State Authentication_request and finally to the resending of the HTTP request with the login information from the user. The response types Redirect and FormPOST specify an address the browser will send an HTTP request to in the following HTTP transaction. This next HTTP request is treated by the next iteration of the entire state-transition diagram, but to trigger it the browser sends a message to its own port $\mathsf{self}_\mathsf{B}?$, with the format accepted in the start state.

4 Ideal User Browsing Behavior U

In browser-based protocols, the browser's user has an important role because the browser itself only provides rudimentary trust management. The user also controls most of the browser behavior and has the final say about the browser's actions. Thus we consider the user as an active protocol participant and model it by a machine U. In general, U is transparent; however, it enforces the general requirements for browser-based protocols. In particular, it stores data about trust relationships to other parties, performs user authentication including the crucial verification of the server's identity, tests certificates, observes the status of secure channels, and logs off from the browser in error cases.

As shown in Figure 2, the machine U works as a proxy between the browser and the protocol interface. Above the protocol interface, one has to imagine the remaining, protocol- or application-dependent actions of the human user, which we sometimes call the "real user". The machine U forwards normal browser communication from the real user to B and the browser's pages back. In contrast, it handles typical trust tasks solely in interaction with the browser, without involving the real user. With the message compromised, it notifies the real user that the browser behaved against U's expectations and that U aborted the interaction with it.

As in Section 3.3, we distinguish persistent and volatile variables of U. As these variables contain confidential data like passwords, they are also important in an information flow analysis.

The persistent variables of U model its trust relationships. The sequence T_U contains a tuple $(host, sid, login, sec)$ for each server that U has a special relationship with. The tuples have the type Server $= \mathsf{URLHost} \times \Sigma^* \times \Sigma^* \times \mathcal{P}(\mathsf{ChType})$. Here $host$ contains the server's hostname, sid its identity in a secure channel and $login$ the login information for user U. The set sec contains the channel types allowed for user authentication with

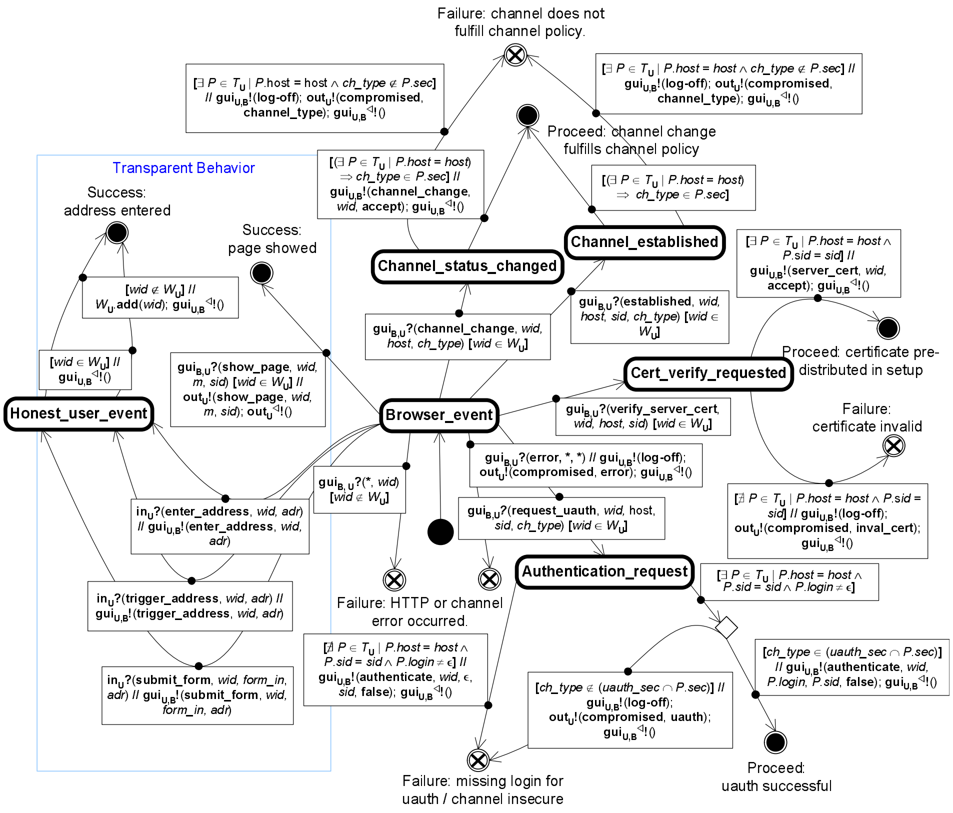

Failure: channel does not fulfill channel policy.
[$\exists P \in T_U$ | P.host = host $\wedge$ $ch\_type \notin P.sec$] // gui$_{U,B}$!(log-off); out$_U$!(compromised, channel_type); gui$_{U,B}{}^{\triangleleft}$!()
[$\exists P \in T_U$ | $P.host = host \wedge ch\_type \notin P.sec$] // gui$_{U,B}$!(log-off); out$_U$!(compromised, channel_type); gui$_{U,B}{}^{\triangleleft}$!()
Proceed: channel change fulfills channel policy
Transparent Behavior
Success: address entered
Success: page showed
[($\exists P \in T_U$ | $P.host = host$) $\Rightarrow ch\_type \in P.sec$] // gui$_{U,B}$!(channel_change, wid, accept); gui$_{U,B}{}^{\triangleleft}$!()
[($\exists P \in T_U$ | $P.host = host$) $\Rightarrow ch\_type \in P.sec$]
Channel_status_changed
Channel_established
[$\exists P \in T_U$ | $P.host = host \wedge P.sid = sid$] // gui$_{U,B}$!(server_cert, wid, accept); gui$_{U,B}{}^{\triangleleft}$!()
[$wid \notin W_U$] // W_U.add(wid); gui$_{U,B}{}^{\triangleleft}$!()
gui$_{B,U}$?(established, wid, $host$, sid, ch_type) [$wid \in W_U$]
gui$_{B,U}$?(channel_change, wid, $host$, ch_type) [$wid \in W_U$]
Proceed: certificate pre-distributed in setup
Cert_verify_requested
[$wid \in W_U$] // gui$_{U,B}{}^{\triangleleft}$!()
gui$_{B,U}$?(show_page, wid, m, sid) [$wid \in W_U$] // out$_U$!(show_page, wid, m, sid); out$_U{}^{\triangleleft}$!()
Honest_user_event
Browser_event
gui$_{B,U}$?(verify_server_cert, wid, $host$, sid) [$wid \in W_U$]
Failure: certificate invalid
gui$_{B,U}$?(error, *, *) // gui$_{U,B}$!(log-off); out$_U$!(compromised, error); gui$_{U,B}{}^{\triangleleft}$!()
gui$_{B,U}$?(*, wid) [$wid \notin W_U$]
in$_U$?(enter_address, wid, adr) // gui$_{U,B}$!(enter_address, wid, adr)
gui$_{B,U}$?(request_uauth, wid, host, sid, ch_type) [$wid \in W_U$]
[$\nexists P \in T_U$ | $P.host = host \wedge P.sid = sid$] // gui$_{U,B}$!(log-off); out$_U$!(compromised, inval_cert); gui$_{U,B}{}^{\triangleleft}$!()
Authentication_request
[$\exists P \in T_U$ | $P.host = host \wedge P.sid = sid \wedge P.login \neq \epsilon$]
in$_U$?(trigger_address, wid, adr) // gui$_{U,B}$!(trigger_address, wid, adr)
Failure: HTTP or channel error occurred.
[$\nexists P \in T_U$ | $P.host = host \wedge P.sid = sid \wedge P.login \neq \epsilon$] // gui$_{U,B}$!(authenticate, wid, ϵ, sid, false); gui$_{U,B}{}^{\triangleleft}$!()
[$ch\_type \in (uauth\_sec \cap P.sec)$] // gui$_{U,B}$!(authenticate, wid, $P.login$, $P.sid$, false); gui$_{U,B}{}^{\triangleleft}$!()
in$_U$?(submit_form, wid, $form\_in$, adr) // gui$_{U,B}$!(submit_form, wid, $form\_in$, adr)
[$ch\_type \notin (uauth\_sec \cap P.sec)$] // gui$_{U,B}$!(log-off); out$_U$!(compromised, uauth); gui$_{U,B}{}^{\triangleleft}$!()
Failure: missing login for uauth / channel insecure
Proceed: uauth successful

Fig. 5. The ideal user browsing behavior represented by machine U

that server. The pairs $(host, sid)$ within this table must be unique. The set $uauth\_sec$ models the general policy of U for allowed channel types for user authentication and contains the channel types that are acceptable.

For the volatile variables, we use similar names as in the browser machine: the address adr and channel type ch_type refer to the address the browser established a channel to. For secure channels the server identity sid additionally contains the identity according to the server's certificate. The variable P is an instance of the type Server.

We define the state transition function δ_{U} by the state diagram in Figure 5. The Start state models the user machine being idle, waiting for an input from the real user with an address adr to retrieve or for a browser event. After having issued an address request to the browser, the machine observes the browser's behavior and reacts to events generated by B. The state machine models *transparent behavior* on the left side (around State Honest_user_event), where it only forwards messages between protocol interface and browser. This transparent part handles the messages enter_address, trigger_address, submit_form, and show_page. The user tracks channel status changes and channels established in the States Channel_status_changed and Channel_established, verifies certificates that B doubts in State Cert_verify_requested, and forwards errors and pages to the protocol interface. The user also handles the user authentication process in State $\mathsf{Authentication\_request}$.

5 Channel Machine Secchan

Our browser model comes with a channel abstraction $\mathsf{secchan}$ for secure and insecure browser channels. For space reasons, we describe this machine only partially here.

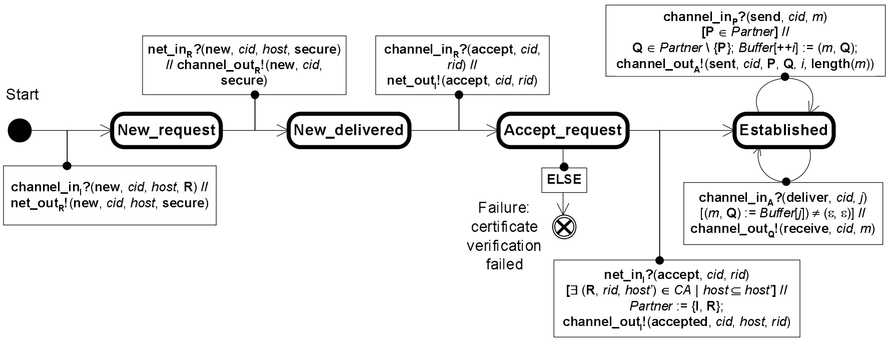
As shown in Figure 2, each machine M with network access has two ports $\mathsf{channel\_in_M}!$ and $\mathsf{channel\_out_M}?$ to connect to $\mathsf{secchan}$. The channel machine connects to the adversary by two means. The ports $\mathsf{net\_out_M}!$ and $\mathsf{net\_in_M}?$ are for insecure channels and not needed here. The ports $\mathsf{channel\_out_A}!$ and $\mathsf{channel\_in_A}?$ modeling the imperfections of secure channels. The adversary also controls the network scheduling and decides which messages are delivered.

The machine $\mathsf{secchan}$ chooses channel identifiers uniquely and keeps track of channels. To model insecure DNS, it queries the adversary for ports corresponding to hostnames. It has a table CA of tuples $binding = (port, sid, host) \in \Sigma^+ \times \Sigma^* \times \mathsf{URLHost}$ linking a certified identity sid and a base hostname $host$ to the port index of a communication partner of $\mathsf{secchan}$. The setup of this table enforces that the identities sid are nonambigously bound to one unique machine M, i.e., if an honest party M controls an identity sid no other machine may act under this identity. We define security domains as tree of URIs covered by a server identity sid, for which no other party M^* can control a server identity sid^*. More formally we define such domains as follows:

Definition 1 (Channel Security Domain). *For a binding* $(\mathsf{M}, sid_\mathsf{M}, host_\mathsf{M})$*, we call a* $URIM$ *with* $\mathsf{host}(URIM) \subseteq host_\mathsf{M} \subseteq sid_\mathsf{M}$ *a* channel security domain *of machine* M *if* CA *does not contain another entry* $(\mathsf{M}^*, sid^*, host^*)$ *with* $\mathsf{M}^* \neq \mathsf{M}$ *and* $host^* \supseteq host$ *for any* $host \subseteq \mathsf{host}(URIM)$. $\diamond$

For handling a concrete channel instance, $\mathsf{secchan}$ dispatches the communication to a sub-machine. Such an instance contains the channel identifier cid, the port indices of the $initiator$ and $responder$, the server's actual address $host$, the server's identity rid and the security level, here $\mathsf{server\_auth\_secure}$. We depict the most important steps of a secure channel instance in Figure 6 and discuss the establishment of a secure channel in the following.

Clients initiate secure channels to an address $host$ by the command new with the parameter $ch\_type = \mathsf{server\_auth\_secure}$. Then $\mathsf{secchan}$ queries the adversary for the recipient port index R corresponding to $host$, chooses a unique channel identifier cid, and dispatches to a sub-machine for a secure channel (Figure 6) with R, $host$, and cid as parameters. The sub-machine contains the channel type $ch\_type = \mathsf{server\_auth\_secure}$ as constant variable. This sub-machine handles further communication. First it notifies the server with the channel identifier cid. The server may accept the channel and identify itself under an identity $rid \in \mathsf{URLHost}$. The secure channel instance verifies the server identity in State $\mathsf{accept\_request}$: It tests whether it has a tuple $(\mathsf{S}, rid, host') \in CA$ such that the current address $host$ lies under the base address $host'$. If yes, it notifies the client that the channel was accepted. From now on, the port indices of client and server are non-ambiguously bound to the channel identifier cid. Thus client and server are the fixed *channel partners* of this channel. Both partners may send messages referring to cid.



Start
New_request
New_delivered
Accept_request
Established
channel_in$_I$?(new, cid, host, R) // net_out$_R$!(new, cid, host, secure)
net_in$_R$?(new, cid, host, secure) // channel_out$_R$!(new, cid, secure)
channel_in$_R$?(accept, cid, rid) // net_out$_I$!(accept, cid, rid)
ELSE
Failure: certificate verification failed
net_in$_I$?(accept, cid, rid) [∃ (R, rid, host') ∈ CA | host ⊆ host'] // Partner := {I, R}; channel_out$_I$!(accepted, cid, host, rid)
channel_in$_P$?(send, cid, m) [P ∈ Partner] // Q ∈ Partner \ {P}; Buffer[++i] := (m, Q); channel_out$_A$!(sent, cid, P, Q, i, length(m))
channel_in$_A$?(deliver, cid, j) [(m, Q) := Buffer[j]) ≠ (ε, ε)] // channel_out$_Q$!(receive, cid, m)

Fig. 6. State diagram of a single instance of a secure channel

6 Security of User Authentication

In this section, we present the first protocol proof based on a detailed browser model: We show the security of typical password-based user authentication by one server. Such user authentication is an important building block for most other security protocols based on browsers, e.g., in federated identity management.

6.1 Authentication Server

The overall system is a special case of the architecture shown in Figure 2. We consider the definition of one server S; of course there can be several such servers and also servers of different types interacting with the same browsers and users. We only rename the free ports of this server from $\mathsf{in_S}$? and $\mathsf{out_S}$! into $\mathsf{uauth\_in_S}$? and $\mathsf{uauth\_out_S}$! to indicate that it offers a user authentication service. Further, we specialize the architecture by allowing the adversary full access to the browser's cache and history, i.e., we show that user authentication (in contrast to some other protocols) is not vulnerable to such attacks. This means that the adversary connects to all free ports in Figure 2 that are not defined to belong to the protocol interface.

The inputs at the ports that S does not share with a prior machine and its persistent variables are shown in Tables 1 and 2. We refer to the two parts of an entry e in the user metadata table $MetaU_{\mathsf{S}}$ as $e.id$ and $e.login$. We require that both id and $login$ are unique within the table $MetaU_{\mathsf{S}}$ of a correct server S at all times.

Table 1. Protocol in- and outputs of the authentication server S

Port	Type	Parameters	Description
$\mathsf{uauth\_in_S}$?			Input to authentication server S
	start	$cid : \Sigma^*$	Start authentication of channel cid
$\mathsf{uauth\_out_S}$!			Output of authentication server S
	done	$cid : \Sigma^*$, $idu : \Sigma^*$	Authentication for channel cid finished with identity idu, where ϵ means failure.

Table 2. Persistent variables of the authentication server S

Name	Domain	Description	Init.
$host_S$	URLHost	Hostname of this server	See setup
sid_S	Σ^*	Identity of this server for secure channels	See setup
$MetaU_S$	$\mathcal{P}(\Sigma^+ \times \Sigma^*)$	Pairs of known user identities and login information.	$\emptyset$

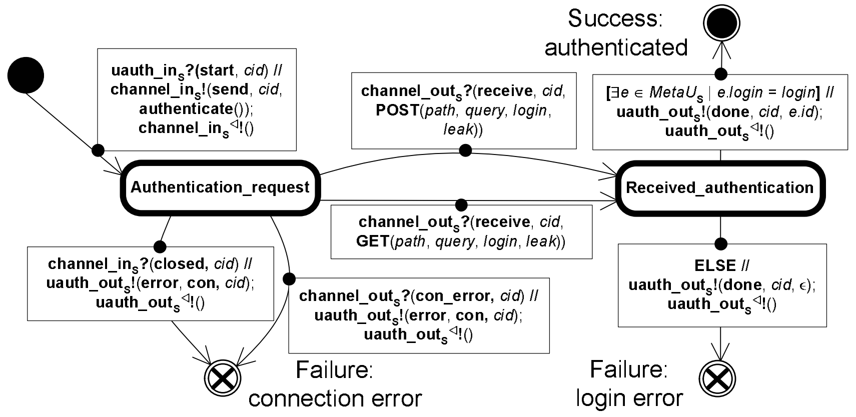

Success: authenticated
uauth_in_S?(start, cid) // channel_in_S!(send, cid, authenticate()); channel_in_S◁!()
channel_out_S?(receive, cid, POST(path, query, login, leak))
[∃e ∈ MetaU_S | e.login = login] // uauth_out_S!(done, cid, e.id); uauth_out_S◁!()
Authentication_request
Received_authentication
channel_out_S?(receive, cid, GET(path, query, login, leak))
channel_in_S?(closed, cid) // uauth_out_S!(error, con, cid); uauth_out_S◁!()
channel_out_S?(con_error, cid) // uauth_out_S!(error, con, cid); uauth_out_S◁!()
ELSE // uauth_out_S!(done, cid, ϵ); uauth_out_S◁!()
Failure: connection error
Failure: login error

Fig. 7. State machine of the user authentication server S

The state machine for one authentication protocol run of server S is shown in Figure 7. The server user (typically a higher protocol) starts authentication for some channel with identifier cid. The server sends an authentication request over the channel cid. Upon receipt of an authentication message, it looks up whether the included login information is present in its user metadata. If yes, it outputs the corresponding identity as the main part of the authentication result, else ϵ.

6.2 Setup Assumptions

As set-up for a particular user machine U and authentication server S, they exchange login information $login_{U,S} \neq \epsilon$ such that U and S are the only parties that obtain information about it. Further, U must know a valid certificate identity of S so that it can verify later that it has a secure channel to S. Formally, the result of the set-up is this:

Definition 2 (UAuth Setup Assumptions). *For a user* U *and an authentication server* S *we assume:*

a. The set T_U *of* U*'s trusted servers contains an entry* $(hostS, sidS, login_{U,S}, \{\text{secure}\})$ *where* $hostS = host_S$ *and* $sidS = sid_S$ *for the hostname and identity variables of* S.

b. The authentication server's user table $MetaU_S$ *contains an entry* $(id_U, login_{U,S})$*, where the user's identity* id_U *is unique in* $MetaU_S$.

c. No other variables contain information about $login_{U,S}$.

d. The binding table CA of the secure channel abstraction secchan *contains a triple* $(S, sid_S, host_S)$ *where* $\text{host}(URIS) \subseteq host_S \subseteq sid_S$ *and URIS defines a channel security domain of* S. ◇

6.3 Security of User Authentication

We now show that user authentication as defined by the general user machine U and the specific authentication server S is secure. Essentially, security means that when S has performed a successful protocol run of the user authentication protocol, indicated by the output $(\mathsf{done}, cid, id_{\mathsf{U}})$, then it is indeed connected with the user with identity id_{U}. More precisely, we show in Lemma 1 that such an output implies that S holds a secure channel with channel identifier cid where the communication partner is the browser B of the user U who has this identity id_{U}. We make a relatively strong statement: We have not required that S only makes its requests on secure channels, nor that the user correctly logs out of browser sessions or otherwise protects caches and histories. Extended protocols, e.g., the continued secure use of the channel for which the authentication is made, may need additional assumptions.

Lemma 1 (User Authentication). *Let a correct user machine* U *and authentication server* S *be given that have performed setup according to Definition 2 at some time with the user identity* id_{U}*, and let the user's browser* B *be correct. Then the following statement holds unless an adversary can guess* $login_{\mathsf{U},\mathsf{S}}$ *based on a priori knowledge of its distribution, its length, and the results of previous guessing attempts, which each exclude one potential value: If* S *outputs* $(\mathsf{done}, cid, id_{\mathsf{U}})$ *at* $\mathsf{uauth\_out_S!}$ *then there exists a secure channel instance* $\mathsf{SCh_{bs}}$ *in* $\mathsf{secchan}$ *with*

$$\mathsf{SCh_{bs}}.cid = cid \;\wedge\; \mathsf{SCh_{bs}}.state = \mathsf{established}$$
$$\wedge\; \mathsf{SCh_{bs}}.ch\_type = \mathsf{server\_auth\_secure} \;\wedge\; \mathsf{SCh_{bs}}.Partner = \{\mathsf{B}, \mathsf{S}\}.$$

We present the full proof of Lemma 1 in the long version of this paper [12]. □

7 Conclusion

In prior art, browser-based protocols only came with vulnerability analyses and informal security considerations. However, those methods do not guarantee the protocols' security and do not meet the requirements of industry embracing browser-based protocols in complex scenarios. We designed the first model for the rigorous security analysis of browser-based protocols. Our model encompasses generic machines for browsers, user browsing behavior and channel abstraction that allow precise protocol proofs. We have also proven the security of the initial password-based user authentication, a very common protocol on its own and a key ingredient of browser-based protocols. In future work, we will use this model to analyze and prove the security of POST- and artifact-based protocols in the prominent area of identity federation.

References

1. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1994.
2. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels (extended abstract). In *Advances in Cryptology: EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002. Extended version in IACR Cryptology ePrint Archive 2002/059, `http://eprint.iacr.org/`.

3. Scott Cantor and Marlena Erdos. Shibboleth-architecture draft v05, May 2002. http://shibboleth.internet2.edu/.
4. Zhe Dang and Richard Kemmerer. Using the ASTRAL model checker for cryptographic protocol analysis. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. http://dimacs.rutgers.edu/Workshops/Security/.
5. Tim Dierks and Christopher Allen. RFC 2246: The TLS protocol, January 1999. Status: Standards Track.
6. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
7. Bruno Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Proc. International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1997.
8. Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999. Status: Standards Track.
9. Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001. USENIX. An extended version is available as MIT-LCS-TR-818.
10. Thomas Groß. Security analysis of the SAML Single Sign-on Browser/Artifact profile. In *Proc. 19th Annual Computer Security Applications Conference*. IEEE, December 2003.
11. Thomas Groß and Birgit Pfitzmann. Proving a WS-Federation Passive Requestor profile. In *2004 ACM Workshop on Secure Web Services (SWS)*, Washington, DC, USA, October 2004. ACM Press.
12. Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser model for security analysis of browser-based protocols. IACR Cryptology ePrint Archive 2005/127, May 2005. http://eprint.iacr.org/.
13. Chris Kaler and Anthony Nadalin (ed.). Web Services Federation Language (WS-Federation), Version 1.0, July 2003. BEA and IBM and Microsoft and RSA Security and VeriSign, http://www-106.ibm.com/developerworks/webservices/library/ws-fed/.
14. Chris Kaler and Anthony Nadalin (ed.). WS-Federation: Passive Requestor Profile, Version 1.0, July 2003. BEA and IBM and Microsoft and RSA Security and VeriSign, http://www-106.ibm.com/developerworks/library/ws-fedpass/.
15. Richard A. Kemmerer. Using formal verification techniques to analyze encryption protocols. In *Proc. 1987 IEEE Symp. on Security and Privacy*, pages 134–138, Oakland, California, April 1987. IEEE.
16. David P. Kormann and Aviel D. Rubin. Risks of the Passport single signon protocol. *Computer Networks*, 33(1–6):51–58, June 2000.
17. Hugo Krawczyk. SKEME: A versatile secure key exchange mechanism for the Internet. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '96)*, pages 114–127, San Diego, California, February 1996. Internet Society.
18. Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). In *CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2001.
19. Liberty Alliance Project. Liberty Phase 2 final specifications, November 2003. http://www.projectliberty.org/.
20. Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–135, 1995.

21. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.
22. Nancy Lynch. I/O automaton models and proofs for shared-key communication systems. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 14–29, 1999.
23. Catherine Meadows. Using narrowing in the analysis of key management protocols. In *Proc. 10th IEEE Symposium on Security & Privacy*, pages 138–147, 1989.
24. Microsoft Corporation. .NET Passport documentation, in particular Technical Overview, and SDK 2.1 Documentation (started 1999), September 2001.
25. Jonathan K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proc. 5th IEEE Symposium on Security & Privacy*, pages 134–141, 1984.
26. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. 18th IEEE Symposium on Security & Privacy*, pages 141–151, 1997.
27. John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0 and related protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997. `http://dimacs.rutgers.edu/Workshops/Security/`.
28. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.
29. OASIS Standard. Security assertion markup language (SAML) V1.1, November 2002. `http://www.oasis-open.org/committees/security/docs/`.
30. Object Management Group. Unified modeling language (UML), March 2003. `http://www.omg.org/technology/documents/formal/uml.htm`.
31. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Cryptology*, 6(1):85–128, 1998.
32. Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
33. Birgit Pfitzmann. Privacy in enterprise identity federation - policies for Liberty single signon. In *3rd International Workshop on Privacy Enhancing Technologies (PET 2003)*, volume 2760 of *Lecture Notes in Computer Science*, pages 189–204, Berlin, March 2003. Springer-Verlag, Berlin Germany.
34. Birgit Pfitzmann. Privacy in enterprise identity federation - policies for Liberty 2 single signon. *Elsevier Information Security Technical Report (ISTR)*, 9(1):45–58, 2004. `http://www.sciencedirect.com/science/journal/13634127`.
35. Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. 22nd IEEE Symposium on Security & Privacy*, pages 184–200, 2001. Extended version of the model (with Michael Backes) IACR Cryptology ePrint Archive 2004/082, `http://eprint.iacr.org/`.
36. Birgit Pfitzmann and Michael Waidner. Privacy in browser-based attribute exchange. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 52–62, Washington, USA, November 2002.
37. Birgit Pfitzmann and Michael Waidner. Analysis of Liberty single-signon with enabled clients. *IEEE Internet Computing*, 7(6):38–44, 2003.
38. Birgit Pfitzmann and Michael Waidner. Federated identity-management protocols — where user authentication protocols may go. In *Security Protocols—11th International Workshop*, Lecture Notes in Computer Science, Cambridge, UK, April 2003. Springer-Verlag, Berlin Germany.
39. Eric Rescorla. Internet RFC 2818: HTTP over TLS, May 2000.

40. Victor Shoup. On formal models for secure key exchange. Research Report RZ 3120 (#93166), IBM Research, April 1999. Version 4, November 1999, available from `http://www.shoup.net/papers/`.
41. David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

A Details of Browser B

Table 3 contains the interface of B, i.e., its ports and the messages sent or expected there. We now describe the additional functions and predicates used by the state-transition function: The function ctype(adr) with ctype : URLHostPath $\longrightarrow$ ChType determines the channel type corresponding to the argument adr. If the address is HTTPS the channel type is server_auth_secure, in other cases insecure. The functions path(adr) : URLHostPath $\longrightarrow$ URLHost, path(adr) : URLHostPath $\longrightarrow$ URLPath, and query(adr) : URLHostPath $\longrightarrow$ Σ^* return parts of an URL argument adr. We use several predicated for handling of HTML forms: fparse(Σ^*) : $\Sigma^* \longrightarrow \Sigma^*$ extracts a from from a HTML document. The predicate fmatch$(form, form_in)$ with fmatch : $\Sigma^* \times \Sigma^* \longrightarrow$ Bool checks whether the parameter names of $form$ and the user inputs $form\_in$ match. The function fmerge$(form, form_in)$: $\Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$ merges a given $form$ with the user inputs $form\_in$ to a new form. The predicate parsable(m) : $\Sigma^* \longrightarrow$ Bool checks whether message m is parsable according to the HTTP specification for HTTP responses.

Table 3. Input and output types of browser machine B

Port	**Type**	**Parameters**	**Description**
$\mathsf{in_B}?$	do_leak		Leak command from OS
$\mathsf{out_B}!$	leak	$info : \Sigma^*$	Info leakage of B to OS
$\mathsf{gui_{U,B}}?$			*Inputs from user* U
	enter_address	$wid : \Sigma^*$, adr : $\mathsf{URLHostPath}$	Input in address line
	trigger_address	$wid : \Sigma^*$, adr : $\mathsf{URLHostPath}$	Clicking of a link
	submit_form	$wid : \Sigma^*$, $m : \Sigma^*$, adr : $\mathsf{URLHostPath}$	Submission of a form
	channel_change	$wid : \Sigma^*$, d : $\{\mathsf{accept}, \mathsf{reject}\}$	Consent to sec level
	server_cert	$wid : \Sigma^*$, d : $\{\mathsf{accept}, \mathsf{reject}\}$, $sid : \Sigma^*$	Result of cert verify
	authenticate	$wid : \Sigma^*$, $login : \Sigma^*$, $sid : \Sigma^*$, $store$: Bool	User authentication
	log_off		User logs off from B
$\mathsf{gui_{B,U}}!$			*Outputs to user* U
	error	$wid : \Sigma^*$, $type$: $\{\mathsf{con}, \mathsf{res}\}$	Error notification
	established	$wid : \Sigma^*$, $host$: $\mathsf{URLHost}$, $sid : \Sigma^*$, ch_type : ChType	A channel was established
	channel_change	$wid : \Sigma^*$, $host$: $\mathsf{URLHost}$, ch_type : ChType	Channel sec level changed
	verify_server_cert	$wid : \Sigma^*$, $host$: $\mathsf{URLHost}$, $sid : \Sigma^*$	Request to verify cert
	request_uauth	$wid : \Sigma^*$, $host$: $\mathsf{URLHost}$, $sid : \Sigma^*$, ch_type : ChType	Request for user auth
	show_page	$wid : \Sigma^*$, $m : \Sigma^*$, $sid : \Sigma^*$	Rendering a payload page
$\mathsf{self_B}!$, $\mathsf{self_B}?$			*Selfdelegation of* B
	trigger_address	adr : $\mathsf{URLHostPath}$	Triggers a redirect
	submit_form	$m : \Sigma^*$, adr : $\mathsf{URLHostPath}$	Scripted form submission
$\mathsf{channel\_out_B}?$			*Inputs from* secchan
	accepted	$cid : \Sigma^*$, $host$: $\mathsf{URLHost}$, $sid : \Sigma^*$	Server accepted channel
	receive	$cid : \Sigma^*$, $m : \Sigma^*$	Received a message
	closed	$cid : \Sigma^*$	Server closed channel
	con_error	$cid : \Sigma^*$	Connection error notify
	ca_untrusted	$sid : \Sigma^*$	Browser does not trust CA
	inval_cert	$sid : \Sigma^*$	Cert was fully invalid
$\mathsf{channel\_in_B}!$			*Outputs to* secchan
	new	$host$: $\mathsf{URLHost}$, $type$: ChType	Establish a new channel
	send	$cid : \Sigma^*$, $m : \Sigma^*$	Send message to channel
	close	$cid : \Sigma^*$	Close channel
	accept_server_cert	$sid : \Sigma^*$	User accepted server cert