A Framework for Formal Analysis of Privacy on SSO Protocols

Kailong Wang[1], Guangdong Bai[2(✉)], Naipeng Dong[1], and Jin Song Dong[1,3]

[1] National University of Singapore, Singapore, Singapore
{dcswaka,dcsdn,dcsdjs}@nus.edu.sg
[2] Singapore Institute of Technology, Singapore, Singapore
guangdong.bai@singaporetech.edu.sg
[3] Griffith University, Nathan, Australia

Abstract. Single Sign-on (SSO) protocols, which allow a website to authenticate its users via accounts registered with another website, are forming the basis of user identity management in contemporary websites. Given the critical role they are playing in safeguarding the privacy-sensitive web services and user data, SSO protocols deserve a rigorous formal verification. In this work, we provide a framework facilitating formal modeling of SSO protocols and analysis of their privacy property. Our framework incorporates a formal model of the web infrastructure (e.g., network and browsers), a set of attacker models (e.g., malicious IDP) and a formalization of the privacy property with respect to SSO protocols. Our analysis has identified a new type of attack that allows malicious participants to learn which websites the victim users have logged in to.

Keywords: Single Sign-on · Privacy · Formal verification framework

1 Introduction

Single Sign-on (SSO) protocols, which allow users to log in to a website, i.e., the relying party (RP), using the accounts registered with another website, i.e., the identity provider (IDP), are becoming the cornerstone of user identity management in contemporary websites. These protocols serve as the safeguard of various privacy-sensitive web services. Nonetheless, they have been continually found vulnerable and insecure by previous research [1–6].

Given the critical role that SSO protocols are playing, they deserve a rigorous security assessment, and formal verification ideally, before they are implemented and deployed for practical use. However, the challenge on formally verifying SSO protocols is at least twofold. First, formal verification requires an accurate formal model of the underlying web infrastructure which SSO protocols rely upon. The web infrastructure is complicated as it involves the server-side infrastructure (e.g., web servers and SSO SDKs), the client-side infrastructure (e.g., web browsers) and various communication channels. In addition, SSO protocols often rely on new techniques and features (e.g., HTML5's `postMessage`) to fulfill its

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 763–777, 2018.
https://doi.org/10.1007/978-3-319-78813-5_41

advanced functions (e.g., cross-domain communication on the client side). These features increase the complexity of the SSO protocols. For example, misusing `postMessage` or client-side storage may lead to credential leakage [1,7].

The second challenge is regarding the comprehensiveness of the attacker behaviors and the targeted properties. Since SSO protocols rely on web clients, web servers and various communication channels, they are naturally exposed to a large attack surface. As a result, the behaviors of malicious participants (e.g., malicious IDPs) have to be formalized when analyzing the SSO protocols. As for the properties, the *privacy* property – whether an attacker is able to track which RP a user has logged in to, is becoming a public concern [3]. However, existing studies have mainly focused on the authentication property [4,5,8].

In this work, we propose a framework for analyzing the privacy property of SSO protocols. Our framework consists of a formal model of the web infrastructure, three types of attacker models and a formal definition of the privacy property of SSO protocols. We abstract the whole infrastructure into three parts which are essential in SSO, including the *web browser*, the *network* and the *web server*. Our attack models contain three types of malicious IDP – the Honest-But-Curious IDP Server which infers the user's login information based on his own knowledge, the Malicious IDP Server which is capable of sending fake information to requesters and the Malicious IDP Client which is an IDP's client-side web page capable of invoking browser APIs (for example, to request the browser to open a new window). In our framework, we use the *applied pi calculus* [9] as our modeling language, given that it can be automatically verified using the state-of-the-art verifier ProVerif [10]. The privacy property is thus formalized as the observational equivalence [11].

We apply our framework to analyze a novel privacy-respecting protocol named SPRESSO [12]. This protocol is representative and is suitable to test our framework, because modeling it covers most of the web techniques, including end-to-end communication between web servers and the browser, HTML5's cross-domain communication, AJAX and so on. We have found that SPRESSO suffers from a privacy flaw which allows a malicious IDP to abuse two key pairs to learn which users have logged in to a particular RP.

2 A Verification Framework for SSO

In this section, we present our verification framework for formally analyzing SSO protocols. First, we introduce the used modeling language. Next, we explain our web infrastructure model, followed by the three attacker models. Finally, we present our formalization of the privacy property of SSO protocols.

2.1 The Modeling Language

We use a variant of the applied pi calculus [9] for modeling protocols, attackers and the privacy property. This calculus assumes an infinite set of names which are used for modeling communication channels and atomic data, an infinite set

$P, Q :=$	plain process	$A, B :=$	extended process
0	null process	P	plain process
$P \mid Q$	parallel composition	$A \mid B$	parallel
$!P$	replication	**new** x; A	variable restriction
new $\mathtt{n}$; P	name restriction	**new** $\mathtt{n}$; A	name restriction
in($\mathtt{u}$, x); P	message input	$\{M/x\}$	active substitution
out($\mathtt{u}$, M); P	message output		
if $M =_E N$ **then** P **else** Q	conditional		
let $x = M$ **in** P **else** Q	term evaluation		

Fig. 1. Applied Pi syntax

of variables, and a signature Σ consisting of finite number of symbols (with arity) which are used for modeling cryptographic primitives. Terms are defined as names, variables as well as function symbols applied to terms. A system is modeled as a plain process, whose syntax is defined in Fig. 1. The reasoning on the models in the applied pi calculus is with respect to the built-in Dolev-Yao attacker model [13] who can block, obtain, tamper and/or insert messages over public channels. A process is closed if all variables are either bound by restriction or input, or defined by an active substitution.

Null process 0 does nothing. Process $P \mid Q$ models two processes P and Q running in parallel. Process $!P$ models infinite number of process P running in parallel, capturing unbounded number of sessions. Name restriction **new** $\mathtt{n}$; P binds the name $\mathtt{n}$ in process P, capturing both fresh random numbers and private names and channels. Message input **in**($\mathtt{u}$, x); P describes that the process reads a message from channel $\mathtt{u}$ and binds the received message to x in process P. Message output **out**($\mathtt{u}$, M); P describes that the process sends a message M on channel $\mathtt{u}$ and runs P afterwards. The conditional evaluation **if** $M =_E N$ **then** P **else** Q runs P when equation $M =_E N$ is true under equational theory E otherwise runs Q. If Q is null, this process can be reduced to **if** $M =_E N$ **then** P. The term evaluation **let** $x = M$ **in** P **else** Q bounds x to M and takes process branch P, otherwise, Q is taken. If Q is null, the term evaluation can be simplified to **let** $x = M$ **in** P. We denote **new** $\mathtt{n_1}; \cdots;$ **new** $\mathtt{n_m}$ by **new** $\tilde{\mathtt{n}}$. The extended process $\{M/x\}$ indicates the substitution of variable x with term M. An evaluation context is an extended process with a hole which is not in a scope of a replication, a conditional, an input, or an output.

2.2 Web Infrastructure Model

Figure 2 shows our abstraction of the web infrastructure. In the abstraction, the web infrastructure consists of three components: the *web browsers*, the *network* and the *web servers*. It represents a common scenario where users use the browsers to download documents and communicate with the web servers via the network.

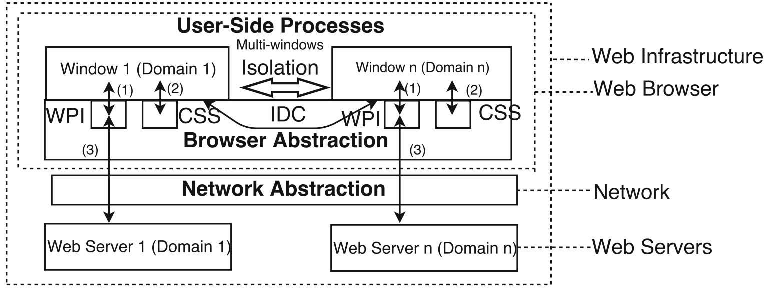

Fig. 2. Web infrastructure abstraction

Web Browser Model. The web browser model has a list of *windows/iframes* (denoted by *window* $1, \ldots,$ *window* n in Fig. 2) which are containers for the client-side documents of the websites. In addition, the model includes the *webpage parser/interpreter* (denoted by *WPI*), the *client-side storage* (denoted by *CSS*), the *inter-domain communication* (denoted by *IDC*) and the *isolation*. They are important features for analyzing the privacy property of SSO protocols.

WPI parses and interprets the programs downloaded from the web servers. The WPI includes complex functions which may not be relevant to the SSO protocols, such as page rendering. Therefore, our framework only models the part that processes the SSO-relevant commands, as shown in the following model. These commands include open windows `OW` (line k_1–k_2), get the parent window `parentOf` (line k_3), establish http(s) connections `http(s)Connect` (line k_4–k_5), send http(s) messages `http(s)Send` (line k_6–k_7) and receive http(s) messages `http(s)Receive` (line k_8–k_9). Note that terms w_1 and w_2 denote window names, terms $e_1, \cdots, e_m$ denote participants interacting with each other, such as windows and web servers, and terms $msg_1, \cdots, msg_l$ denote messages. Name `priv` denotes the private channel to call the browser commands, and name `priv'` denotes the private channel to send and receive the http(s) messages through the *network* which is defined later.

$$
\begin{array}{llr}
WPI := & (\mathbf{in}(\mathtt{priv}, (= \mathtt{OW}, w_1)); \mathbf{let}\ w_2 = \mathsf{Child}(w_1)\ \mathbf{in} & k_1 \\
& \ \mathbf{out}(\mathtt{priv}, (\mathtt{OW}, w_1, w_2)); & k_2 \\
& \ !\mathbf{in}(\mathtt{priv}, (= \mathtt{parentOf}, = w_2)); \mathbf{out}(\mathtt{priv}, (\mathtt{parentOf}, w_2, w_1)))| & k_3 \\
& (\mathbf{in}(\mathtt{priv}, (= \mathtt{http(s)Connect}, e_1, e_2)); & k_4 \\
& \ \mathbf{out}(\mathtt{priv}', (\mathtt{http(s)Connect}, e_1, e_2)))| & k_5 \\
& (\mathbf{in}(\mathtt{priv}, (= \mathtt{http(s)Send}, msg_1, e_3, e_4)); & k_6 \\
& \ \mathbf{out}(\mathtt{priv}', (\mathtt{http(s)Send}, msg_1, e_3, e_4)))| & k_7 \\
& (\mathbf{in}(\mathtt{priv}', (= \mathtt{http(s)Receive}, msg_2, e_5, e_6)); & k_8 \\
& \ \mathbf{out}(\mathtt{priv}, (\mathtt{http(s)Receive}, msg_2, e_5, e_6))). & k_9
\end{array}
$$

CSS includes both short-term storage (i.e., cookies and SessionStorage) and long-term (i.e., LocalStorage) storage that can only be accessed by the client-side web page of the same URL domain. In particular, we explicitly model LocalStorage because it may store data relevant to the privacy property. For example, compromising the LocalStorage may disclose the user's login status at a certain RP [3]. As shown in the following model, the LocalStorage is modeled as a process LS where `LSS` denotes the command of storing messages and `LSR` denotes that of retrieving messages. We do not explicitly model the short-term storage since it can be recorded in the local variables.

$$LS := \mathbf{in}(\texttt{priv}, (= \texttt{LSS}, (index, msg))); !\mathbf{out}(\texttt{priv}, (\texttt{LSR}, (index, msg))).$$

IDC is mostly achieved by an API called `postMessage` in HTML5. It is extensively used in the SSO protocols since the involved participants (at least RP and IDP) which have to communicate with each other are typically from different domains. As shown in the following model, the `postMessage` is modeled as a process PM where `PMS` and `PMR` denote sending and receiving messages respectively. The sender and the receiver window identities (i.e. w_1 and w_2) are required to indicate the two endpoints of the `postMessage`.

$$PM := \mathbf{in}(\texttt{priv}, (= \texttt{PMS}, w_1, w_2, msg)); \mathbf{out}(\texttt{priv}, (\texttt{PMR}, w_2, msg)).$$

Isolation among domains is a security feature (the *same origin policy*) provided by the web browsers. This feature ensures the domains at the client side are isolated such that scripts from one domain cannot access data belonging to other domains. Since we model the windows as individual and parallel processes, the documents received by a window cannot be accessed by others. In addition, cross-domain messaging between different windows is via private channels, restraining messages only to the intended processes according to the protocol. Thus, isolation property is implicitly retained in our web browser model.

Network Model. The network model covers both http and https channels which are the basis for data transmission in the SSO protocols. The network model in our framework is shown below. The terms $e_1, \cdots, e_m$ denote communicating participants, while the terms $Msg_1, \cdots, Msg_l$ denote exchanged messages.

$$
\begin{array}{lll}
HTTPconnect & := \textbf{in}(\texttt{priv}', (= \texttt{httpConnect}, e_1, e_2)); \textbf{out}(\texttt{c}, (e_1, e_2)). & l_1 \\
HTTPsend & := \textbf{in}(\texttt{priv}', (= \texttt{httpSend}, Msg_1, e_3, e_4)); \textbf{out}(\texttt{c}, Msg_1, e_3, e_4). & l_2 \\
HTTPreceive & := \textbf{in}(\texttt{c}, (Msg_2, e_5, e_6)); \textbf{out}(\texttt{priv}', (\texttt{httpReceive}, Msg_2, e_5, e_6)). & l_3 \\
HTTPSconnect & := \textbf{in}(\texttt{priv}', (= \texttt{httpsConnect}, e_7, e_8)); & l_4 \\
 & \quad \textbf{let}\ k = \mathsf{httpskey}(e_7, e_8)\ \textbf{in}\ \textbf{out}(\texttt{c}, (e_7, e_8)). & l_5 \\
HTTPSsend & := \textbf{in}(\texttt{priv}', (= \texttt{httpsSend}, Msg_3, e_9, e_{10})); \textbf{new}\ \texttt{nonce}; & l_6 \\
 & \quad \textbf{let}\ key = \mathsf{httpskey}(e_9, e_{10})\ \textbf{in} & l_7 \\
 & \quad \textbf{out}(\texttt{c}, \mathsf{enc}((\texttt{nonce}, Msg_3), key), e_9, e_{10}). & l_8 \\
HTTPSreceive & := \textbf{in}(\texttt{c}, (EncMsg, e_{11}, e_{12})); \textbf{let}\ key = \mathsf{httpskey}(e_{11}, e_{12})\ \textbf{in} & l_9 \\
 & \quad \textbf{let}\ (Nonce, Msg_4) = \mathsf{dec}(EncMsg, key)\ \textbf{in} & l_{10} \\
 & \quad \textbf{out}(\texttt{priv}', (\texttt{httpsReceive}, Msg_4, e_{11}, e_{12})). & l_{11}
\end{array}
$$

The *http channels* are not encrypted. Hence, we simply model the http messages to be sent and received on the public channel c (line l_1–l_3). The *https channels* include two parts: *session key establishment* which sets up a session key between the two communicating participants using handshake protocols (line l_4–l_5), and *message exchange* which uses the established session key to protect the messages. In particular, the message is encrypted when it is sent out (line l_6–l_8) and decrypted when it is received (line l_9–l_{11}).

Web Servers. Web servers are the server-side SSO participants such as the RPs and IDPs. Their behaviors need to be manually modeled according to the protocol specifications.

Table 1. Interfaces: infrastructure inputs

Interfaces	Functionality
out(priv,(OW, w_1))	Open window request from window w_1
in(priv,(=OW, w_1, w_2))	Return created child window w_2 of w_1
out(priv,(parentOf, w_1))	Request parent window of window w_1
in(priv,(=parentOf, =w_1, w_2))	Return parent window w_2 of window w_1
out(priv,(PMS, w_1, w_2, *msg*)	Send postMessage from w_1 to w_2
in(priv,(=PMR, =w_1, *msg*)	Receive postMessage intended for w_1
out(priv,(LSS,(index, *msg*)))	Store message *msg* to LocalStorage
in(priv,(=LSR,(=index, *msg*)))	Retrieve message *msg* from LocalStorage
out(priv,(http(s)Connect, e_1, e_2))	Request http(s) connection over e_1 and e_2
in(c,(e_1, e_2)	Establish http(s) connection over e_1 and e_2
out(priv,(http(s)Send, *msg*, e_1, e_2)	Send http(s) message *msg* from e_1 to e_2
in(priv,(=http(s)Receive, *msg*, e_1, e_2))	Receive http(s) message *msg* from e_1 by e_2

In summary, we provide the interfaces listed in Table 1 to facilitate modeling of the SSO protocols using our web infrastructure. Each interface includes an

out message representing the command from a client-side process to the web infrastructure and an **in** message representing the response from the web infrastructure to the client-side process. Overall, the web infrastructure is defined as a process where all the above processes run in parallel, $WebInfra = WPI \mid LS \mid PM \mid HTTPconnect \mid HTTPsend \mid HTTPreceive \mid HTTPSconnect \mid HTTPSsend \mid HTTPSreceive$.

2.3 Attacker Models

In the SSO protocols, the privacy property is violated if the attacker learns which RPs the users have logged in to. Therefore, we mainly consider the malicious IDP since the privacy property would be trivially violated if either the user or the RP is malicious. According to the attacker's capabilities, we define three attacker models namely the *Honest-But-Curious IDP Server*, the *Malicious IDP Server* and the *Malicious IDP Client*.

Honest-But-Curious IDP Server tries to break the user's privacy based on its own knowledge. It records messages generated and received by itself and tries to derive the user's login information from those recorded messages. This attacker can be simulated in the applied pi calculus by sending the built-in Dolev-Yao attacker all the messages of base type (i.e., not of channel type) generated and received by the IDP, such that the existing reasoning techniques can be reused to check whether the privacy property is satisfied or not.

Malicious IDP Server can forge messages based on its knowledge and send them out upon requests in place of the authentic messages from the IDP server.

Malicious IDP Client mainly follows the behavior of an honest IDP's client-side web page but contains a malicious iframe. The malicious iframe has the capability of invoking the web infrastructure interfaces. Take the BrowserID, which is a well-known SSO protocol, as an example. It suffers from the attack that when a user logs in, a malicious window can be triggered to inform the attacker the RP the user has logged in to [2].

2.4 Formalization of SSO Privacy Property

We use the observational equivalence relation defined in the applied pi calculus [9] to formalize the privacy property. Intuitively, two processes are observationally equivalent if the Dolve-Yao attacker cannot distinguish the two processes. In order to further explain our formalization, we define the generalized evaluation context of SSO processes in the applied pi calculus as follows.

Definition 1 (Evaluation Context of General SSO Processes). *We define an evaluation context D as the following SSO process with a hole ([_]).*

$$D := \textbf{new}\ \tilde{\mathtt{n}};\ C(account, rp)\sigma_{11} \,|\, C(account, rp)\sigma_{12} \,|\, \cdots \,|\, [_] \,|\, \cdots \,|\, C(account, rp)\sigma_{nm} \,|\, !RP_1 \,|\, \cdots \,|\, !RP_m \,|\, !IDP \,|\, !WebInfra,$$

– $\tilde{\mathtt{n}}$ indicates private channel names and data in this process.
– Process $C(account, rp)$ models the client-side login process including the behaviors of the client-side RP, the client-side IDP, etc., together with the user's behaviors (e.g., input the password). The *account* and the *rp* are two free variables denoting the user account and the RP domain which are instantiated by $\mathtt{Account}_i$ and $\mathtt{RPname}_j$ respectively using the substitution σ_{ij} where $\sigma_{ij} = \{\mathtt{Account}_i/account,\ \mathtt{RPname}_j/rp\}$. There are totally n accounts and m RP domains, therefore σ is a set $\sigma = \{\sigma_{11}, \cdots, \sigma_{nm}\}$. The $RP_1, \cdots, RP_m$ and the IDP are honest RPs and IDP. Any sub-process can be null except $WebInfra$.
– The hole [_] can be filled with a process C(account, rp)σ_{ij} | C(account, rp)σ_{lk} with σ_{ij}, $\sigma_{lk} \in \sigma$.

With the evaluation context, we formally define privacy property as follows.

Definition 2 (SSO Protocol Privacy). *An SSO protocol preserves user's privacy if the following observational equivalence query is true*

$$D[C\{\mathtt{Account}_1/account, \mathtt{RPname}_1/rp\} | C\{\mathtt{Account}_2/account, \mathtt{RPname}_2/rp\}] \approx D[C\{\mathtt{Account}_1/account, \mathtt{RPname}_2/rp\} | C\{\mathtt{Account}_2/account, \mathtt{RPname}_1/rp\}]$$

for accounts $\mathtt{Account}_1$ *and* $\mathtt{Account}_2$ *and RPs* $\mathtt{RPname}_1$ *and* $\mathtt{RPname}_2$*.*

In this definition, $\mathtt{Account}_1$ and $\mathtt{Account}_2$ represent two user accounts, and $\mathtt{RPname}_1$ and $\mathtt{RPname}_2$ represent two RP domains. Intuitively, the definition indicates that an SSO protocol respects user's privacy when $\mathtt{Account}_1$ logs in to $\mathtt{RPname}_1$ and $\mathtt{Account}_2$ logs in to $\mathtt{RPname}_2$ cannot be differentiated from (i.e., observationally equivalent to) $\mathtt{Account}_1$ logs in to $\mathtt{RPname}_2$ and $\mathtt{Account}_2$ logs in to $\mathtt{RPname}_1$. Note that two account and two RPs are required in order to define privacy, given that if there is only one account or RP, the malicious IDP can trivially know who is logging in to the RP based on the RP-IDP or user-IDP communication.

3 Case Study

In this section, we use SPRESSO [12] as a case study to illustrate how to apply our framework to analyze an SSO protocol. The general process of the SPRESSO protocol is shown in Fig. 3. Following this process, the SPRESSO protocol is modeled, as shown in Figs. 4, 5 and 6.

Figure 4 shows the overall model of the SPRESSO protocol. The client-side process is modeled as $C(\mathtt{Account}, \mathtt{RPname})$ where $\mathtt{Account}$ and $\mathtt{RPname}$ are

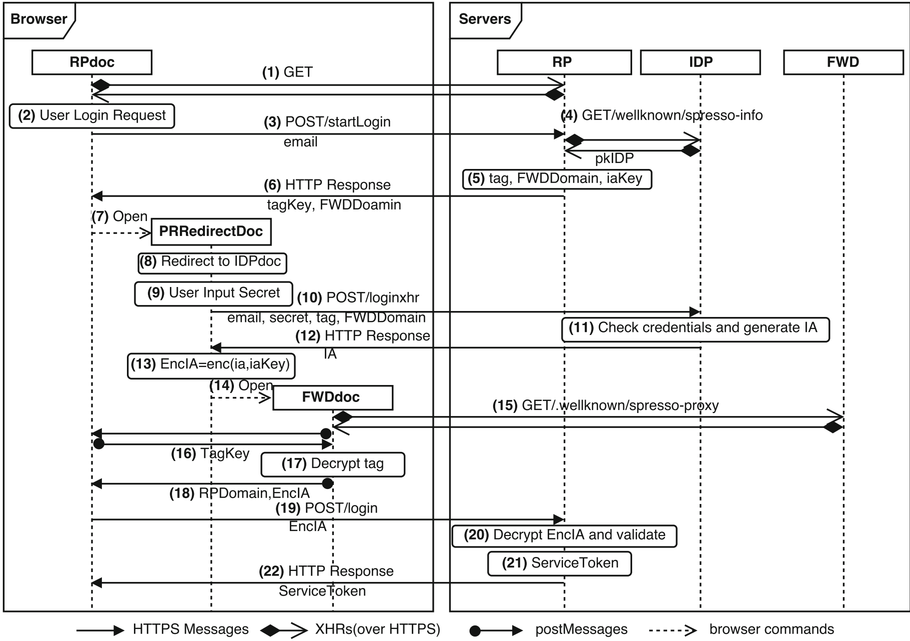

Fig. 3. SPRESSO protocol flow chart [12]

$SPRESSO_proc :=$

s_1 $\quad$ **new** $\mathtt{skidp}$; **let** $pkidp = \mathsf{pk}(\mathtt{skidp})$ **in**

s_2 $\quad$ **new** $\mathtt{IDPname}$; **new** $\mathtt{RPname}$; **new** $\mathtt{Account}$;

s_3 $\quad$ $(!\mathbf{out}(\mathtt{c}, \mathtt{IDPname}) \,|!\mathbf{out}(\mathtt{c}, \mathtt{RPname}) \mid C(\mathtt{Account}, \mathtt{RPname}) \mid$

s_4 $\quad$ $!IDP_proc(\mathtt{IDPname}) \,|!RP_proc(\mathtt{RPname}) \,|!\, WebInfra)$

Fig. 4. SPRESSO protocol

instantiations for the free variables *account* and *rp* in the process $C(account, rp)$. The process $C(\mathtt{Account}, \mathtt{RPname})$ is comprised of subprocesses namely the $RPdoc_proc$, the $IDPdoc_proc$ and the $FWDdoc_proc$, which represent the behaviors of the client-side web pages (i.e., the RPdoc, the IDPdoc and the FWDdoc) of the corresponding web servers. The IDP and the RP servers are modeled in the $IDP_proc(\mathtt{IDPname})$ and the $RP_proc(\mathtt{RPname})$. In the rest of this section, we detail the modeling of the client-side process and the server-side processes.

$\boldsymbol{C(account,\ rp)} :=$

q_1 $\mathbf{in}(\mathtt{c}, IDPname);$

q_2 $\mathbf{let}\ email = (account, IDPname)\ \mathbf{in}$

q_3 $\mathbf{let}\ RPname_1 = rp\ \mathbf{in}$

q_4 $\mathbf{new}\ \mathtt{root};\ \mathbf{out}(\mathtt{priv}, (\mathtt{OW}, \mathtt{root}));$

q_5 $\mathbf{in}(\mathtt{priv}, (= \mathtt{root}, rpdoc));$

q_6 $RPdoc_proc \mid IDPdoc_proc \mid FWDdoc_proc$

$\boldsymbol{RPdoc_proc} :=$

q_7 $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsConnect}, rpdoc, RPname_1));$

q_8 $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsSend}, email, rpdoc, RPname_1));$

q_9 $\mathbf{in}(\mathtt{priv}, (= \mathtt{httpsReceive}, (tagkey, fwdomain,$
$logsesstoken), = rpdoc, = RPname_1));$

q_{10} $\mathbf{out}(\mathtt{priv}, (\mathtt{OW}, rpdoc));\ \mathbf{in}(\mathtt{priv}, (= rpdoc, rddoc));$

q_{11} $\mathbf{out}(\mathtt{priv}_{\mathtt{rd}}, (logsesstoken, rddoc));$

q_{12} $\mathbf{in}(\mathtt{priv}, (= \mathtt{PMR}, fwdoc, = rpdoc, = \mathtt{ready}));$

q_{13} $\mathbf{out}(\mathtt{priv}, (\mathtt{PMS}, fwdoc, rpdoc, tagkey);$

q_{14} $\mathbf{in}(\mathtt{priv}, (= \mathtt{PMR}, = fwdoc, = rpdoc, (EncIA,$
$= \mathsf{getrpdomain}(RPname_1))));$

q_{15} $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsSend}, (EncIA, logsesstoken), rpdoc, RPname_1));$

q_{16} $\mathbf{in}(\mathtt{priv}, (= \mathtt{httpsReceive}, = \mathtt{success}, rpdoc, RPname_1)).$

$\boldsymbol{IDPdoc_proc} :=$

q_{17} $\mathbf{in}(\mathtt{priv}_{\mathtt{rd}}, (logsesstoken_1, rddoc_1));$

q_{18} $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsConnect}, rddoc_1, RPname_1));$

q_{19} $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsSend}, logsesstoken_1, rddoc_1, RPname_1));$

q_{20} $\mathbf{in}(\mathtt{priv}, (= \mathtt{httpsReceive}, (= rddoc_1, tag, fwdomain1,$
$= email, iakey), = rddoc_1, = RPname_1));$

q_{21} $\mathbf{new}\ \mathtt{idpdoc};\ \ \mathbf{out}(\mathtt{priv}, (\mathtt{httpsConnect}, \mathtt{idpdoc}, IDPname));$

q_{22} $\mathbf{let}\ password = \mathsf{getpss}(email)\ \mathbf{in}$

q_{23} $\mathbf{out}(\mathtt{priv}, (\mathtt{httpsSend}, (email, password, fwdomain_1, tag),$
$\mathtt{idpdoc}, IDPname));$

q_{24} $\mathbf{in}(\mathtt{priv}, (= \mathtt{httpsReceive}, ia, = \mathtt{idpdoc}, = IDPname));$

q_{25} $\mathbf{let}\ EncIA = enc(ia, iakey)\ \mathbf{in}$

q_{26} $\mathbf{out}(\mathtt{priv}, (\mathtt{OW}, rddoc_1));$

q_{27} $\mathbf{in}(\mathtt{priv}, (= rddoc_1, fwdoc_1));$

q_{28} $(\mathbf{out}(\mathtt{priv}_{\mathtt{fw}}, (EncIA, tag, fwdoc_1))).$

$\boldsymbol{FWDdoc_proc} :=$

q_{29} $\mathbf{in}(\mathtt{priv}_{\mathtt{fw}}, (EncIA_1, tag_1, fwdoc_2));$

q_{30} $\mathbf{out}(\mathtt{priv}, (\mathtt{parentOf}, fwdoc_2));$

q_{31} $\mathbf{in}(\mathtt{priv}, (= \mathtt{parentOf}, rddoc_2, = fwdoc_2);$

q_{32} $\mathbf{out}(\mathtt{priv}, (\mathtt{parentOf}, rddoc_2));$

q_{33} $\mathbf{in}(\mathtt{priv}, (= \mathtt{parentOf}, rpdoc_1, = rddoc_2);$

q_{34} $\mathbf{out}(\mathtt{priv}, (\mathtt{PMS}, fwdoc_2, rpdoc_1, \mathtt{ready}));$

q_{35} $\mathbf{in}(\mathtt{priv}, (= \mathtt{PMR}, = fwdoc_2, = rpdoc_1, tagkey_1));$

q_{36} $\mathbf{let}\ (RPdomain_1, nonce_4) = \mathsf{dec}(tag_1, tagkey_1)\ \mathbf{in}$

q_{37} $\mathbf{out}(\mathtt{priv}, (\mathtt{PMS}, fwdoc_2, rpdoc_1, (EncIA_1, RPdomain_1))).$

Fig. 5. The client-side process

3.1 Client-Side Process

The model of the client-side process of the SPRESSO protocol is shown in Fig. 5.

RPdoc_proc models the RP login page, i.e., the RPdoc in Fig. 3. We assume the user has an account (*account*) from the IDP (*IDPname*) (line q_1–q_2 in Fig. 5)[1]. If the user wants to log in to the RP ($RPname_1$) (line q_3), he opens the RP's login page, i.e. the RPdoc, by sending the `OW` command to our framework (line q_4–q_5). The RPdoc sends a login request and establishes an https connection with the RP server by sending the `httpsConnect` command to our framework in ((1,2), line q_7). Then the RPdoc sends the email address to the RP server by sending the `httpsSend` command to our framework ((2,3), line q_8) and receives the response by waiting for the message marked by the `httpsReceive` command from our framework ((6), line q_9). Next, the RPdoc opens the window RPRedirectDoc (line q_{10}) and passes the *loginsesstoken* and the RPRedirectDoc identity *rddoc* via a private channel $\texttt{priv}_{\texttt{rd}}$ ((7), line q_{11}). Then the RPdoc receives the `ready` from its grandchild window FWDdoc via `postMessage` by waiting for the message marked by the `PMR` command from our framework (line q_{12}), and replies the received *tagKey* back via `postMessage` by sending the `PMS` command ((16), line q_{13}). Then, the RPdoc delivers the encrypted identity assertion ($EncIA$) from the FWDdoc ((18), line q_{14}) to the RP server by https ((19), line q_{15}). Then the RPdoc waits for the successful login notification ((22), line q_{16}).

IDPdoc_proc models the IDP login page, i.e., the IDPdoc in Fig. 3. The previously created window RPRedirectDoc redirects itself to the IDPdoc ((8), line q_{17}–q_{20}). This step is to avoid the identity leak of the RP to the IDP due to the referrer header set by the browser. Since our browser model does not include the referrer header, we can simply continue the IDPdoc process right after the RPRedirectdoc process (line q_{21}–q_{28}). The IDPdoc extracts the IDP domain from the received email address and establishes an https connection with the IDP server ((8), line q_{21}). The user sends his credentials (i.e., email address and password) to IDP server ((9, 10), line q_{22}–q_{23}). Next, the IDPdoc receives the identity assertion *ia* ((12), line q_{24}) and generates an encrypted identity assertion ($EncIA$) with *iakey* ((13), line q_{25}). Finally, the IDPdoc opens a new window FWDdoc (line q_{26}–q_{27}), and passes the $EncIA$, the *tag* and the FWDdoc identity $fwdoc_1$ to the FWDdoc ((14), line q_{28}).

FWDdoc_proc models the FWDdoc in Fig. 3. The FWDdoc is a proxy within the browser to transfer information between windows, hiding the identity of the RP from the IDP. The FWDdoc first receives the encrypted identity assertion ($EncIA_1$), the tag (tag_1) and the FWDdoc identity ($fwdoc_2$) from its parent IDPdoc (line q_{29}). Then he identifies its grandfather window $rpdoc_1$ by sending the `parentOf` command to our framework (line q_{30}–q_{33}). Next the FWDdoc sends the `ready` to its grandfather window $rpdoc_1$ and receives the $tagkey_1$ via

[1] For simple reference to the same information in different figures, we use the following format ((k), line x_j) to represent the step k in Fig. 3, line x_j in Fig. 5 (when x_j is a q_j) or Fig. 6 (when x_j is a p_j).

postMessage (line q_{34}–q_{35}). Finally, the FWDdoc decrypts the tag with the $tagkey_1$ to extract the $RPDomain_1$ (line q_{36}) and sends the $EncIA$ back to the RPdoc specified by the $RPDomain_1$ ((17,18), line q_{37}).

3.2 Server-Side Processes

RP_proc(*RPname*) models the RP server in Fig. 6. The RP establishes an https connection with the RPdoc upon the request (line p_1) and receives an email address ((3), line p_2). The RP extracts the IDP domain name from the received email address and requests the public key from the corresponding IDP ((4), line p_3–p_4). Next, the RP generates the following session sensitive values: a nonce ($\texttt{nonce}_3$), a symmetric key to encrypt the identity assertion ($\texttt{iaKey}_1$), a key to

```
RP_  proc(RPname2) :=
p1   in(c, (rpdoc2, RPname2));
p2   in(priv, (= httpsReceive, (account2, IDPname2), = rpdoc2, = RPname2));
p3   out(priv, (httpsConnect, RPname2, IDPname2));
p4   in(priv, (= httpsReceive, pkidp, = RPname2, = IDPname2));
p5   new nonce3;  new iakey1;  new tagkey2;  new logsesstoken2;
p6   new fwdomain2;
p7   let RPdomain2 = getrpdomain(RPname2) in
p8   let tag2 = enc((RPdomain2, nonce3), tagkey2) in
p9   out(priv, (httpsSend, (tagkey2, fwdomain2, logsesstoken2),
     rpdoc2, RPname2));
p10  in(c, (rddoc2,= RPname2));
p11  in(priv, (= httpsReceive, = logsesstoken2, = rddoc2,= RPname2));
p12  out(priv, (httpsSend, (rddoc2, tag2, fwdomain2,
     (account2,IDPname2), iakey1), rddoc2, RPname2));
p13  in(priv, (= httpsReceive, (EncIA2, = logsesstoken2),
     = rpdoc2,= RPname2));
p14  let ia2 = dec(EncIA2, iakey1) in
p15  let (= tag2,= (account2,IDPname2), = fwdomain2) = getmsg(ia2, pkidp) in
p16  (out(priv, (httpsSend, success, rpdoc2, RPname2)))
p17  else(out(priv, (httpsSend, retry, rpdoc2, RPname2))).

IDP_ proc(IDPname3) :=
p18  in(c, (RPname3, = IDPname3));
p19  out(priv, (httpsSend, pkidp, RPnanme3, IDPname3));
p20  in(c, (idpdoc1,= IDPname3));
p21  in(priv, (= httpsReceive, (email2, password1,
     fwdomain3, tag3), = idpdoc1, = IDPname3));
p22  if password1 = getpss(email2) then
p23  let ia3 = sign((tag3, email2, fwdomain3), skidp) in
p24  out(priv, (httpsSend, ia3, idpdoc1, IDPname3)).
```

Fig. 6. The server-side processes

encrypt the tag ($\mathtt{tagKey}_2$) and a login session token ($\mathtt{logsesstoken}_2$) (line p_5) and chooses a forward domain ($\mathtt{fwdomain}_2$) (line p_6). The RP generates the tag (tag_2) by encrypting the $\mathtt{nonce}_3$ and its domain name $RPdomain_2$ using the $\mathtt{tagKey}_2$ ((5), (line p_7–p_8)). The $\mathtt{tagKey}_2$, the $\mathtt{fwdomain}_2$ and the $\mathtt{logsesstoken}_2$ are sent to the RPdoc ((6), line p_9). Then the RP receives the $\mathtt{logsesstoken}_2$ from the RPRedirectDoc (line p_{11}). Finally, the RP receives the encrypted indentity assertion ($EncIA_2$) together with the $\mathtt{logsesstoken}_2$ from the RPdoc (line p_{13}), after which it extracts the identity assertion (ia_2) (line p_{14}) and checks the signature of the IDP as well as the signed messages ((20), line p_{15}). Upon successful checks, the RP sends the `success` to the RPdoc. Otherwise, the `retry` is sent ((21), line p_{16}–p_{17}).

IDP_proc(`IDPname` ***)*** models the IDP server in Fig. 6. The IDP establishes an https connection with the RP upon the request and passes its public key ((4), line p_{18}-p_{19}). Next, the IDP establishes an https connection with the IDPdoc upon the request and receives the email address ($email_2$), the password ($password_1$), the forward domain ($fwdomain_3$) and the tag (tag_3) from the IDPdoc ((10), line p_{20}-p_{21}). The IDP checks the validity of the password associated with the email address (line p_{22}). Once succeeds, the IDP generates an identity assertion (ia_3) by signing the tag_3, the $email_2$ and the $fwdomain_3$ with its private key (line p_{23}), and sends the identity assertion to the IDPdoc ((13,14), line p_{24}).

3.3 Verification Results

We transform the IDP_proc into the honest-but-curious attacker and query the privacy property in ProVerif. Next we add the malicious IDP client to the $SPRESSO_proc$ and query the privacy property. The verification results show that SPRESSO preserves the privacy property against the above two attacker models. Finally, we transform the IDP_proc into the malicious IDP server and query privacy. The verification result shows that SPRESSO does not preserve privacy property against the malicious IDP server. By analyzing the trace generated by ProVerif, we summary the following attack.

A Logic Flaw in SPRESSO. When a victim user uses his/her account `Account` which is registered from a malicious IDP to log in to an RP, the RP server requests a public key from the malicious IDP server. At this step, for a particular RP $\mathtt{RP}_\mathtt{i}$, if the malicious IDP wants to learn its login users, the IDP can issue a fake public key $\mathtt{pkIDP}_\mathtt{i}$ to it ((4) in Fig. 3); for other RPs, the IDP issues the normal public key `pkIDP`. Later in identity assertion (`IA`) generation, the IDP always uses the private key corresponding to `pkIDP` ((11) in Fig. 3). As a result, a failure is caused when $\mathtt{RP}_\mathtt{i}$ verifies the `IA` using the public key $\mathtt{pkIDP}_\mathtt{i}$ it fetched previously ((20) in Fig. 3). This implies that the user is successfully logged in to the IDP, but actually fails to log in to the RP. We assume that the user will log in again upon receiving a login failure notification, which is common in reality. Upon receiving the second log in request ((10) in Fig. 3), the malicious

IDP knows the identity of the user who wants to log in to $\mathtt{RP_i}$. This sabotages the declared privacy property of SPRESSO.

4 Related Work

SSO Privacy Property has drawn little attention until recently. Not much work has been done on the SSO privacy checking and verification. BrowserID developed by Mozilla is claimed to preserve the SSO privacy that prevents IDPs from learning which RP a user is trying to log in to. Fett et al. [2,3] have analyzed the privacy property of BrowserID manually by trace indistinguishability with a comprehensive protocol model and have found an attack. In our work, we have discovered a new privacy attack which is not considered in their analysis.

Web Infrastructure Modeling is also a relatively new research area with few models incorporating crucial web mechanisms. Previous work associated with SSO web security analysis [5,8,14,15] only considers a very limited web model. TrustFound [16,17] has proposed a model for network attacker. Akhawe et al. [18] have built a general model of the web and have verified the model using an automatic verification tool Alloy. Bansal et al. [19,20] have proposed a more comprehensive web infrastructure model WebSpi in the applied pi calculus and have analyzed the authentication property of OAuth2.0 using WebSpi. Fett et al. [2,3,12] have built and applied a complex and complete web infrastructure model that closely follows the published standards and specifications for the web. Compared to this work, our web infrastructure model is compact and specific to SSO protocols, which can successfully run on ProVerif.

5 Conclusion

In this paper, we present a formal framework consisting of a web infrastructure formal model, three attacker models, and the formalization of the privacy property. We have analyzed SPRESSO using our framework and have detected a previously-unknown flaw which allows a malicious IDP to use an incorrect public key to differentiate the users which log in to a particular RP.

Acknowledgment. This research is supported by the National Research Foundation, Singapore (No. NRF2015NCR-NCR003-003).

References

1. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE S&P (2012)
2. Fett, D., Küsters, R., Schmitz, G.: An expressive model for the web infrastructure: definition and application to the BrowserID SSO system. In: IEEE S&P (2014)

3. Fett, D., Küsters, R., Schmitz, G.: Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In: ESORICS, pp. 43–65 (2015)
4. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: AuthScan: automatic extraction of web authentication protocols from implementations. In: NDSS (2013)
5. Sun, S.-T., Hawkey, K., Beznosov, K.: Systematically breaking and fixing openid security: formal analysis, semi-automated empirical evaluation, and practical countermeasures. Comput. Secur. **31**, 465–483 (2012)
6. Ye, Q., Bai, G., Wang, K., Dong, J.S.: Formal analysis of a single sign-on protocol implementation for android. In: ICECCS, pp. 90–99 (2015)
7. Hanna, S., Shinz, E.C.R., Akhawe, D., Boehmz, A., Saxena, P., Song, D.: The emperor's new API: on the (in)secure usage of new client side primitives. In: W2SP (2010)
8. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Tobarra, L.: Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In: Workshop on Formal Methods in Security Engineering (2008)
9. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL, pp. 104–115 (2001)
10. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: CSFW, pp. 82–96 (2001)
11. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols. J. Comput. Secur. **17**, 435–487 (2009)
12. Fett, D., Küsters, R., Schmitz, G.: SPRESSO: a secure, privacy-respecting single sign-on system for the web. In: CCS, pp. 1358–1369 (2015)
13. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**, 198–207 (1983)
14. Jackson, D.: In: Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS, p. 20 (2002)
15. Kerschbaum, F.: Simple cross-site attack prevention. In: Workshop on Security and Privacy in Communications Networks, pp. 464–472 (2007)
16. Bai, G., Hao, J., Wu, J., Liu, Y., Liang, Z., Martin, A.: Trustfound: towards a formal foundation for model checking trusted computing platforms. In: FM, pp. 110–126 (2014)
17. Hao, J., Liu, Y., Cai, W., Bai, G., Sun, J.: vTRUST: a formal modeling and verification framework for virtualization systems. In: ICFEM, pp. 329–346 (2013)
18. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: CSF, pp. 290–304 (2010)
19. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffei, S.: Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In: POST, pp. 126–146 (2013)
20. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: CSF, pp. 247–262 (2012)