UPPRESSO: An Unlinkable Privacy-PREserving Single Sign-On System

***Abstract*—As a widely adopted identity management and authentication mechanism in today's Internet, single sign-on (SSO) allows a user to maintain only the credential for the identity provider (IdP), instead of one credential for each relying party (RP), which shifts the burden of user authentication from RPs to the IdP. However, SSO introduces new privacy leakage threats, since (*a*) a curious IdP could track *all* the RPs a user has visited, and (*b*) collusive RPs could learn a user's online profile by linking her identifiers and activities across multiple RPs. Several privacy-preserving SSO solutions have been proposed to defend against either the curious IdP or collusive RPs, however, none of them can address both privacy leakage threats at the same time.**

In this paper, we propose a privacy-preserving SSO system, called *UPPRESSO*, to protect a user's login traces against both the curious IdP and collusive RPs. We first formally analyze the privacy dilemma between SSO security requirements and the new privacy requirements, and convert the SSO privacy problem into an identifier-transformation problem. Then, we design a novel *transformed RP designation* scheme to transform the identifier of the RP, to which the user requests to log in, into a privacy-preserving pseudo-identifier (PID_{RP}) through the cooperation between the user and the RP. Our *trapdoor user identification* scheme allows the RP to obtain a trapdoor from the transformation process and use it to derive a unique account of the user at that RP from her privacy-preserving pseudo-identifier (PID_U) generated by the IdP. The login process of UPPRESSO follows the service pattern of OpenID Connect (OIDC), a widely deployed SSO system, with minimum modifications. Our analysis shows UPPRESSO provides a comprehensive privacy protection while achieving the same security guarantees of OIDC. The experiment evaluation on our UPPRESSO prototype demonstrates a satisfying performance of 254 ms on average for each login.

***Index Terms*—Single sign-on, security, privacy.**

I. Web Model

Our formal analysis of UPPRESSO is based on the Dolev-Yao style web model [**?**], which has been widely used in formal analysis of SSO protocol, e.g., OAuth 2.0 [**?**] and OIDC [**?**]. To make the description cleaner, we foucus on our modification on OIDC, and assume DNS and HTTPS are secure, which has already been anaylzed in [**?**].

A. Communication Model

Here we give a brief presentation of generic Dolev-Yao-style communication model proposed by [**?**] on which our web model is based.

A $signature \Sigma$ consists of a finite set of function symbols, such as `encrypt`, `decrypt`, and `pair`, each with an arity. A function symbol with arity 0 (with no arguments) is a constant symbol. The set of *terms* is defined over a signature Σ, an infinite set of names, and an infinite set of variables.

Equational theory is defined as usual in Dolev-Yao models which introduces the symbol $\equiv$ representing the congruence relation on terms. For instance, $dec(enc(m, k), k) \equiv m$

Messages are defined as formal terms without variables (called ground terms). The signature Σ for the messages in the model is considered containing constants (such as ASCII strings and nonce), sequence symbols (such as n-ary sequences $\langle\rangle$, $\langle.\rangle$, $\langle., .\rangle$ etc.) and further function symbols (such as encryption/decryption and digital signatures). An HTTP request is a common message in the web model, containing a type `HTTPReq`, a nonce n, a method `GET` or `POST`, a domain , a path, URL parameters, request headers, and the body over the Σ in the sequence symbol formate. Here is an example for an HTTP GET request for the domain $\texttt{exa.com/path?para} = 1$ with the headers and body empty.

$$m := \langle \texttt{HTTPReq}, n, \texttt{GET}, \texttt{exa.com}, /path, \langle\langle para, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

Events are the basic communication elements in the model. An event is the term in the formate $\langle a, f, m\rangle$, where the a and f represent the address of sender and receiver, and m is the message transmitted.

Atomic Processes. An $atomic\ Dolev-Yao(DY)\ process$ is constructed as the tuple $p = (I^p, Z^p, R^p, s_0^p)$ representing the single node in the web model, such as the server and browser. I^p is the set of addresses a process listens to, and Z^p is the set of states (terms) which describes the process. R^p is the mapping between the pairs $\langle s, e\rangle$ and $\langle s', e'\rangle$ where $s, s' \in Z^p$ It's worth noting that for one process in a state only a finite set of events can be accepted by the process as the state and event are defined as the input of R^p.

Scripting Processes. The web model also contains the scripting process representing the client-side script loaded by browser such as JavaScript code. However, the $scripting\ process$ must rely on an $atom\ process$ such as browser and provide the relation R witch is called by this $atomic\ process$.

B. Web System

The web system contains a set of processes (including atomic processes and scripting processes) and represents the web infrastructure. A web system is defined as a tuple ($\mathcal{W}$, $\mathcal{S}$, `script`, E^0). $\mathcal{W}$ is the set of atomic processes containing honest processes and malicious processes, $\mathcal{S}$ is the set of scripting processes including honest scripts and malicious scripts, `script` is the set of concrete script code related with specific scripting process in $\mathcal{S}$, and E^0 is the set of events which could be accepted by the processes in $\mathcal{W}$.

Configuration. We firstly define the set of states S of a system, consists of all the current states of processes in $\mathcal{W}$. And the set of events E, for each event $e \in E$, there is always a state $s \in S$, e and s can be accepted by one of the processes as the input. A $configuration$ of the system is defined as the tuple (S, E, N) where N is the mentioned sequence of unused nonces.

Processing Steps. A processing step is the system migrating from the configuration (S, E, N) to (S', E', N') by processing an event $e \in E$.

C. Model Of UPPRESSO

The UPPRESSO model is a web system which is defined as

$$\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \texttt{script}, E^0),$$

$\mathcal{W}$ is the finite set of atom processes in UPPRESSO system including a single IdP server process, multiple honest RP server processes , the browsers representing the users, and the attacker processes. We assume that all the honest RPs are implemented following the same rule so that the process are considered consistent besides of the addresses they listen to. The browsers controlled by user are considered honest. That is, the browser controlled by attackers can behave as an independent atomic process. $\mathcal{S}$ is the finite set of scripting processes consists of $script\_rp$, $script\_idp$ and $script\_attacker$. The $script\_rp$ and $script\_idp$ are downloaded from honest RP and IdP processes and the $script\_attacker$ is downloaded from attacker process considered existing in all browser processes.

We now give a brief description about UPPRESSO model.

- The browser is responsible to send HTTP request, receive HTTP response, handle user behaviour and transmit message between scripting process. As the browser is honest, we only focus on the scripting process running on the browser. The detailed model of browser is shown in [**?**]
- IdP server process (defined as p^i) only accepts the events whose messages are HTTP request and the $path \in /scriptPath, /registrationPath, /loginPath, /loginStatePath, /authorizePath$. The function of each path is shown in Section **??**. All the events are accepted by p^i in any state but the output may be different. The detailed R^i is shown in *.
- RP server process (defined as p^r) only accepts the events whose messages are HTTP request and the $path \in \{ /scriptPath, /loginPath, /startNegotiationPath, /registrationResultPath, /uploadTokenPath \}$. The function of each path is shown in Section **??**. The event with $path \in \{ /scriptPath, /loginPath, /startNegotiationPath \}$ are accepted in any state. However, the event with $path \equiv /registrationResultPath$ is accepted only when the state s is the output for event $path \equiv /startNegotiationPath$. In the same way the following accepted events must be arranged as $path$ in the sequence $/registrationResultPath$, $/uploadTokenPath$.
- IdP and RP scripting process accepts the events in the formate as HTTP response and postMessage. The details about accepted events are shown in **??**.

D. Security Of UPPRESSO

As we assume that the HTTP requests and responses are well protected by TLS, and the postMeassage are securely implemented in browser, therefore, web attackers are not considered. In this section, we are going to prove the following theorem,

Theorem 1. *Let $\mathcal{UWS}$ be a UPPRESSO web system, then $\mathcal{UWS}$ is secure.*

Firstly, an SSO system is considered secure **iff** only the legitimate user can always log into an honest RP under her unique account. Based on the model of UPPRESSO, we found that an attacker can visit an honest RP as the honest user only when the attacker own the cookie which is bound to the honest user by RP. Therefore, the definition of a secure UPPRESSO system is,

Definition 1. Let $\mathcal{UWS}$ be a UPPRESSO web system, $\mathcal{UWS}$ is secure **iff** for any honest RP $r \in \mathcal{W}$ and the authenticated cookie c for honest u, c is unknown to the attacker a.

Therefore, to prove theorem 1, we are going to prove that an authenticate cookie c is unknown to attacker a. The proof can be separated as two parts, initially a does not know any authenticated cookie, and the following requirements must be met.

- If c is the authenticated cookie owned by u, c cannot be obtained by a;.
- If c is an unauthenticated cookie owned by a, c cannot be authenticated by r for u.

Proof Outline. Here we introduce the lemmas briefly to prove that $\mathcal{UWS}$ follows the requirements by Definition 1 so that $\mathcal{UWS}$ is secure. And the detailed proofs to these lemmas are in **??**.

Lemma 1. The cookie owned by honest user will not be leaked to any attacker.

Proof. That is, due to the Same-Origin policy, the honest browser will not leak the cookies to any attacker. And based on the UPPRESSO model, it is to prove that RP server and RP script will not send any cookies to other processes either. Therefore, the attackers cannot obtain the u's authenticated cookie. □

Based on the model of UPPRESSO about RP server process, the procedure of the cookie being authenticated is described as follows.

Definition 2. In $\mathcal{UWS}$, the cookie c is to be set authenticated for user u only when RP r receives a valid u's identity proof from the owner of c.

Then we are going to prove that $\mathcal{UWS}$ follows the requirements that the cookie of the attacker cannot be set authenticated.

Here we propose the lemmas

Lemma 2. Attackers cannot obtain the user u's password in $\mathcal{UWS}$.

Lemma 3. Attackers cannot forge the IdP issued proofs in $\mathcal{UWS}$.

Proof. Lemma 2 can be easily proved because the password is only sent by honest IdP scripting process to IdP server. Lemma 3 is proved as the IdP issued proofs are well signed and verified. Therefore, the following lemma can be proved base on Lemma 2 and Lemma 3. $\square$

Lemma 4. Attackers cannot obtain the u's valid identity proof in $\mathcal{UWS}$.

Proof. We now give a brief proof about Lemma 4. As the attacker attempts to obtain a valid identity proof, he must receive the proof from one of following processes, IdP server process, RP server process, IdP scripting process and RP scripting process. That is, according to the model we find the honest RP scripting process only send identity proof to honest RP server and RP server will not send the proof to any process. It can be proved that only the process who holds u's password can obtain the u's identity proof from IdP server. As the attacker does not know u's password so that he cannot obtain the identity proof from IdP server. To prove that attacker cannot obtain the identity from IdP scripting process is a little complicated so that we here only give a straightforward conclusion. That is when the honest user u sends the identity proof from the IdP scripting process, the receiver is restricted by the RP Certification $cert_r$. And the identity proof is valid in honest RP r only if the $cert_r$ belongs to r (the full proof is in **??**). $\square$

Therefore, $\mathcal{UWS}$ satisfies the requirements in Definition **??**, such that Theorem 1 is proved.

E. Privacy Of UPPRESSO

Firstly we introduce the definition in [] about static equivalence.

Definition 3. Two messages t_1 and t_2 are statically equivalent, written $t_1 \approx t_2$, if and only if, for all terms such as $M(x)$ and $N(x)$ which only contain one variable x without nonces, it is true that $M(t_1) \equiv N(t_1)$ **iff** $M(t_2) \equiv N(t_2)$. For instance, there are the messages m and m', symmetric key k, such that $enc(m,k) \approx enc(m',k)$ is always true to the attackers without the k.

Here we give the new definitions

Definition 4. For a large prime p (2048-bit length) and $p-1$'s prime factor q (256-bit length), there are two constants g_1, g_2 as the generators of p and the constants n_1, n_2 (n_1, n_2 $< q$). We define the function symbol $modpow(a,b,p) = a^b \bmod p$, there are $modpow(g_1,n_1,p) \approx modpow(g_2,n_2,p)$ and $modpow(g_1,n_1,p) \approx modpow(g_1,n_2,p)$ always true due to the discrete logarithm problem as the n_1 and n_2 are unknown.

Definition 5. Equivalence of HTTP requests. There are messages m_1 and m_2, we say that $m_1 \approx m_2$ **iff** the following conditions are met,

- If m_1 and m_2 are HTTPs requests, they are equivalent to the observers besides of the receiver.
- If m_1 and m_2 are HTTPs requests, they are equivalent for the receiver **iff** the value of the Host,Path,Origin and Referer headers in both requests are same, as well as the value of the Parameters and Body are statically equivalent.
- If m_1 and m_2 are HTTP requests, they are equivalent to all the observers as the equivalent HTTPS requests to receivers.

Definition 6. Equivalence of events. There are events $e_1 := \langle a_1, f_1, m_1 \rangle$ and $e_2 := \langle a_2, f_2, m_2 \rangle$, we say that $e_1 \approx e_2$ **iff**

- $a_1 \equiv a_2$ or a_1 and a_2 belong to random addresses.
- $f_1 \equiv f_2$ or f_1 and f_2 belong to random addresses.
- m_1 and m_2 are equivalent.

Then we are going to prove the following theorem

Theorem 2. *Let $\mathcal{UWS}$ be a UPPRESSO web system, then $\mathcal{UWS}$ is IdP-Privacy and RP-Privacy.*

The definitions about IdP-Privacy and RP-Privacy are designed as follows.

Definition 7. IdP-Privacy Let $\mathcal{UWS}$ be a UPPRESSO web system, there are honest RPs $r_1, r_2 \in \mathcal{W}$, IdP $i \in \mathcal{W}$ and the honest user u, then $\mathcal{UWS}$ is IdP-Privacy **iff** for every event e_1 received by i during the u logging in to r_1, there is always an event e_2 for the u logging in to r_2, and e_1 and e_2 are equivalent.

Proof. Here we only give a brief proof that $\mathcal{UWS}$ meets the conditions defined in Definition 7. Firstly, it is assumed that the HTTPs transmissions well implemented such that all the events to IdP are regarded as equivalent to web attackers. As we consider IdP server is honest but curious, i can only hold the events to IdP server process and does not attempt to steal parameters from other processes or set any illegal parameters in the system.

Here we only focus on the same user's multiple requests to the IdP. IdP server only accepts the events whose messages are HTTP request and the $path \in /scriptPath$, $/registrationPath$, $/loginPath$, $/loginStatePath$, $/authorizePath$. All the path will be visited in each login procedure. It can be easily found that the visits to $/scriptPath$ and $/loginStatePath$ carrying no parameters and bodies so that the events must be equivalent. The visits to $/loginPath$ only carry u's username and password so that the events are equivalent. Moreover ,the visits to $/registrationPath$ and

$/authorizePath$ carry the PID_{RP}s and $endpoint$s where PID_{RP}s are statically equivalent because of Definition 4 and $endpoint$s are unrelated random constants . Therefore, $\mathcal{UWS}$ meets the conditions defined in Definition 1, so that theorem 1 is proved. □

Definition 8. RP-Privacy Let $\mathcal{UWS}$ be a UPPRESSO web system, there are honest RPs $r_1, r_2 \in \mathcal{W}$ and the honest users u_1 and u_2, then $\mathcal{UWS}$ is RP-Privacy **iff** event through r_1 and r_2 share their states

- for every event e_1 received by r_2 during the u_1 logging in to r_2, there is always an event e_2 for the u_2 logging in to r_2, and e_1 and e_2 are equivalent to r_1.
- for every events received by r_2, the event cannot be straightforward linked to the existing user's attributes at r_1.

RP server process only accepts the events whose messages are HTTP request and the $path \in \{$ $/scriptPath$, $/loginPath$, $/startNegotiationPath$, $/registrationResultPath$, $/uploadTokenPath$ $\}$. As the RPs may behave malicious so that the events received by RP scripting process should also be considered. However, all of the messages received by RP scripting process are transmitted to RP server. Therefore, we only need to focus on the events received by RP server.

Firstly, we assume that all the parameters are set legally. We give the brief proof. The events visiting to $/scriptPath$ and $/loginPath$ carry no parameters and bodies so that the events must be equivalent. The visits to $/startNegotiationPath$ only carry the nonce so that the events are equivalent. The visits to $/registrationResultPath$ carry the IdP signed registration result, however, the contents in the result contains the PID_{RP}, N_U and $endpint$. The contents are all random constants (PID_{RP} is regarded as same as N_U) so that the events are equivalent. The visits to $/uploadTokenPath$ includes the identity proof containing the PID_{RP}, PID_U. According to Definition 4, the PID_Us are statically equivalent to r_1. Moreover, with the r_2 shared state, $Account_{r_1}$s are known to r_1. However, r_1 is unable to transform $Account_{r_1}$s into the users' account $Account_{r_2}$ at r_2 so that the events cannot be linked to the existing user. Therefore, the requirements of Definition 8 are met.

However, as the RPs are considered maybe malicious, such that they will attempt to steal the data from other process or set the malicious parameters during the login procedure. That is, according to Definition 4, the PID_U the $Accounts$ must be equivalent to the attacker as long as the attacker does not know the ID_U. Therefore the attacker may attempt to steal the ID_U from UPPRESSO system. But it is easy to be found that IdP will not send the plain ID_U to any process so that RPs cannot obtain the ID_U. Another way is that RPs may attempt to treat the $Account$ or PID_U to be generated insecurely, but we are going to prove it is impossible.

- RP may lead the login using the forged ID_{RP} or PID_{RP} so that PID_Us and $Accounts$ are no more equivalent. However, ID_{RP} are provided by the $Cert$, protected by the IdP's signature and verified by IdP script. PID_{RP} is generated by the ID_{RP} and the honest user generated nonce. Therefore, it is impossible to lead the honest user to use the illegal ID_{RP} and PID_{RP}.
- RP may also lead the same user to upload the identity proof with same PID_U or $Account$ so that the system is not RP-Privacy according to Definition 8. However, the PID_U is generated with the user's nonce N_U so that it is not controlled by the RP. $Account$ is generated as the form $ID_{RP}^{ID_U} \mod p$, while RPs may lead the user to use the same ID_{RP} to generate identity proof. However, the ID_{RP} is bound with $Cert$ which is verified by the user and it is easy for user to find out the login RP does not coincide the RP name shown on her browser.

Therefore, we consider that $\mathcal{W}$ meet all the requests defined in Definition 8 so that theorem 2 is proved.

APPENDIX A
WEB MODEL

A. Data Formate

Here we provide the details of formate of some messages we use to construct the UPPRESSO.

HTTP Messages. An HTTP request message is the term of the form $\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle$, and an HTTP response message is the term of the form $\langle \texttt{HTTPResp}, nonce, status, headers, body\rangle$. The details are dined as follows:

- HTTPReq and HTTPResp are the type of messages.
- $nonce$ is the constant nonce mapping the response with the specific request.
- $method$ is the HTTP method, such as GET and POST.
- $host$ is the constant string domain of visited server.
- $path$ is the constant string representing the concrete resource of the server.
- $parameters$ contains the parameters carried by the url as the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, , for example the $parameters$ HTTP request sent to the url $http://www.example.com?type = confirm$ is $\langle\langle type, confirm\rangle\rangle$.
- $headers$ is the header content of each HTTP messages as the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, such as $\langle\langle Referer, http://www.example.com\rangle, \langle Cookies, c\rangle\rangle$.
- $body$ is the body content carried by HTTP POST request or HTTP response in the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$.
- $status$ is the HTTP status code defined by HTTP standard.

URL. A URL is a term $\langle \texttt{URL}, protocol, host, path, parameters\rangle$, where URL is the type, $protocol$ is chosen in S, P as S stands for HTTPS and P stands for HTTP. The $host, path, andparameters$ are same as in HTTP message.

Origin. An Origin is a term $\langle host, protocol\rangle$ that stands for the specific domain used by the HTTP CORS policy, where $host$ and $protocol$ are defined as same as in URL.

POSTMESSAGE. PostMessage is used in the browser for transmitting messages between scripts from different origins. We define the postMessage as the form $\langle \texttt{POSTMESSAGE}, target, Content, Origin\rangle$, where POSTMESSAGE is the type, $target$ is the constant nonce which stands the for the receiver, $Content$ is the message transmitted and Origin is restricts the receiver's origin.

XMLHTTPREQUEST. XMLHTTPRequest is the HTTP message transmitted by scripts in the browser. That is the XMLHTTPRequest is converted with the HTTP message by the browser. The XMLHTTPRequest in the form $\langle \texttt{XMLHTTPREQUEST}, URL, methods, Body, nonce\rangle$ can be converted into HTTP request message by the browser,and $\langle \texttt{XMLHTTPREQUEST}, Body, nonce\rangle$ is converted from HTTP response message.

Data Operation. The data used in UPPRESSO are defined in the following forms:

- **Standardized Data** is the data in the fixed format, for instance the HTTP request is the standardized data in the form $\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle$. We assume there is an HTTP request $r := \langle \texttt{HTTPReq}, n, \texttt{GET}, example.com, /path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$, here we define the operation on the r. That is the elements in r can be accessed in the form $r.name$, such that $r.method \equiv \texttt{GET}$, $r.path \equiv /path$ and $r.body \equiv \langle\rangle$.
- **Dictionary Data** is the data in the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, for instance the $body$ in HTTP request is dictionary data. We assume there is a $body := \langle\langle username, alice\rangle, \langle password, 123\rangle\rangle$, here we define the operation on the $body$. That is we can access the elements in $body$ in the form $body[name]$, such that $body[username] \equiv alice$ and $body[password] \equiv 123$. We can also add the new attributes to the dictionary, for example after we set $body[age] := 18$, the $body$ are changed into$\langle\langle username, alice\rangle, \langle password, 123\rangle, \langle age, 18\rangle\rangle$.

B. Model Of UPPRESSO

In this section, we will introduce the model of processes in UPPRESSO system, containing IdP server process, RP server process, IdP scripting process and RP scripting process.

C. IdP server process

The state

D. Analysis

Algorithm 1 R^i

Input: $\langle a, f, m\rangle, s$

1: **let** $s := s'$

let $n, method, path, parameters, headers, body$ **such that**
  $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$
  if $possible$; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, s'.IdPScript\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /login$ **then**
  let $cookie := headers[Cookie][cookie]$
  let $session := s'.sessions[cookie]$
  let $username := body[username]$
  let $password := body[password]$
  if $password \not\equiv \texttt{SecretOfID}(username)$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure}\rangle$
    stop $\langle f, a, m'\rangle, s'$
  end if
  let $session[uid] := \texttt{UIDOfUser}(username)$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess}\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /loginInfo$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := session[username]$
  if $username \not\equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Logged}\rangle$
    stop $\langle f, a, m'\rangle, s'$
  end if
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unlogged}\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /dynamicRegistration$ **then**
  let $PID_RP := body[PID_{RP}]$
  let $Endpoint := body[Endpoint]$
  let $Nonce := body[Nonce]$
  if $PID_RP \in \texttt{ListOfPID}()$ **then**
    let $Content := \langle Fail, PID_{RP}, Nonce\rangle$
    let $Sig := \texttt{Sig}(Content, s'.SignKey)$
    let $RegistrationResult := \langle Content, Sig\rangle$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult\rangle$
    stop $\langle f, a, m'\rangle, s'$
  end if
  let $Validity := \texttt{CurrentTime}() + s'.Validity$
  let $s'.RPs := s'.RPs +^{\langle\rangle} \langle PID_{RP}, Endpoint, Validity\rangle$
  let $Content := \langle OK, PID_{RP}, Nonce, Validity\rangle$
  let $Sig := \texttt{Sig}(Content, s'.SignKey)$
  let $RegistrationResult := \langle Content, Sig\rangle$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /authorize$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := session[username]$
  if $username \equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    stop $\langle f, a, m'\rangle, s'$
  end if
  let $PID_{RP} := parameters[PID_{RP}]$
  let $Endpoint := parameters[Endpoint]$

if $PID_{RP} \notin \texttt{ListOfPID}() \vee Endpoint \notin \texttt{EndpointsOFRP}(PID)$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
let $UID := session[uid]$
let $PID_U := \texttt{ModPow}(PID_{RP}, UID, s'.p)$
let $Validity := \texttt{CurrentTime}() + s'.Validity$
let $Content := \langle PID_{RP}, PID_U, s'.ID, Validity\rangle$
let $Sig := \texttt{Sig}(Content, s'.SignKey)$
let $Token := \langle Content, Sig\rangle$
let $s'.Tokens := s'.Tokens +^{\langle\rangle} Token$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Token, Token\rangle\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
stop $\langle\rangle, s'$

Algorithm 2 R^r

Input: $\langle a, f, m\rangle, s$
let $s := s'$
let $n, method, path, parameters, headers, body$ **such that** $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$ **if** $possible$; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, s'.RPScript\rangle$
stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /login$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 302, \langle\langle Location, s'.IdP.ScriptUrl\rangle\rangle, \langle\rangle\rangle$
stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /startNegotiation$ **then**
let $cookie := headers[Cookie]$
let $session := s'.sessions[cookie]$
let $N_U := parameters[N_U]$
let $PID_{RP} := \texttt{ModPow}(s'.ID_{RP}, N_U, s'.IdP.p)$
let $t := \texttt{ExEU}(N_U, s'.IdP.q)$
let $session[N_U] := N_U$
let $session[PID_{RP}] := PID_{RP}$
let $session[t] := t$
let $session[state] := expectRegistration$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Cert, s'.Cert\rangle\rangle$
stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /registrationResult$ **then**
let $cookie := headers[Cookie]$
let $session := s'.sessions[cookie]$
if $session[state] \not\equiv expectRegistration$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
let $RegistrationResult := body[RegistrationResult]$
let $Content := RegistrationResult.Content$
if $\texttt{checksig}(Content, RegistrationResult.Sig, s'.IdP.PubKey) \equiv \texttt{FALSE}$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
let $session := \texttt{null}$
stop $\langle f, a, m'\rangle, s'$
end if
if $Content.Result \not\equiv OK$ **then**

let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle$
let $session := \mathtt{null}$
stop $\langle f, a, m'\rangle, s'$
end if
let $PID_{RP} := session[PID_{RP}]$
let $N_U := session[N_U]$
let $Nonce := \mathtt{Hash}(N_U)$
let $Time := \mathtt{CurrentTime}()$
if $PID_{RP} \not\equiv Content.PID_{RP} \vee Nonce \not\equiv Content.Nonce \vee Time > Content.Validity$ **then**
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle$
let $session := \mathtt{null}$
stop $\langle f, a, m'\rangle, s'$
end if
let $session[PIDValidity] := Content.Validity$
let $Endpoint \in s'.Endpoints$
let $session[state] := expectToken$
let $Nonce' := \mathtt{Random}()$
let $session[Nonce] := Nonce'$
let $Body := \langle PID_{RP}, Endpoint, Nonce'\rangle$
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, Body\rangle$
stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /uploadToken$ **then**
let $cookie := headers[Cookie]$
let $session := s'.sessions[cookie]$
if $session[state] \not\equiv expectToken$ **then**
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
let $Token := body[Token]$
if $\mathtt{checksig}(Token.Content, Token.Sig, s'.IdP.PubKey) \equiv \mathtt{FALSE}$ **then**
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
let $PID_{RP} := session[PID_{RP}]$
let $Time := \mathtt{CurrentTime}()$
let $PIDValidity := session[PIDValidity]$
let $Content := Token.Content$
if $PID_{RP} \not\equiv Content.PID_{RP} \vee Time > Content.Validity \vee Time > PIDValidity$ **then**
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
let $PID_U := Content.PID_U$
let $t := session[t]$
let $Account := \mathtt{ModPow}(PID_U, t, s'.IdP.p)$
if $Account \in \mathtt{ListOfUser}()$ **then**
let $\mathtt{RegisterUser}(Account)$
end if
let $session[user] := Account$
let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{LoginSuccess}\rangle$
stop $\langle f, a, m'\rangle, s'$
end if
stop $\langle\rangle, s'$

Algorithm 3 $script\_idp$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret\rangle$

let $s' := scriptstate$
let $command := \langle\rangle$
let $target := \mathtt{PARENTWINDOW}(tree, docnonce)$
let $IdPDomain := s'.Parameters[IdPDomain]$
switch $s'.q$ **do**
  case $start$**:**
    let $N_U := \mathtt{Random}()$
    let $command := \langle \mathtt{POSTMESSAGE}, target, \langle\langle N_U, N_U\rangle\rangle, \mathtt{null}\rangle$
    let $s'.Parameters[N_U] := N_U$
    let $s'.q := expectCert$
  case $expectCert$**:**
    let $pattern := \langle \mathtt{POSTMESSAGE}, *, Content, *\rangle$
    let $input := \mathtt{CHOOSEINPUT}(scriptinputs, pattern)$
    if $input \not\equiv \mathtt{null}$ **then**
      let $Cert := input.Content[Cert]$
      let $s'.Parameters[Cert] := Cert$
      if $\mathtt{checksig}(Cert.Content, Cert.Sig, s'.PubKey) \equiv \mathtt{null}$ **then**
        let stop $\langle\rangle$
      end if
      let $N_U := s'.Parameters[N_U]$
      let $PID_{RP} := \mathtt{ModPow}(Cert.Content.ID_{RP}, N_U, s'.p)$
      let $s'.Parameters[PID_{RP}] := PID_{RP}$
      let $Endpoint := \mathtt{RandomUrl}()$
      let $s'.Parameters[Endpoint] := Endpoint$
      let $Nonce := \mathtt{Hash} N_U$
      let $Url := \langle \mathtt{URL}, \mathtt{S}, IdPDomain, /dynamicRegistration, \langle\rangle\rangle$
      let $s'.refXHR := \mathtt{Random}()$
      let $command := \langle \mathtt{XMLHTTPREQUEST}, Url, \mathtt{POST},$
        $\langle\langle PID_{RP}, PID_{RP}\rangle, \langle Nonce, Nonce\rangle, \langle Endpoint, Endpoint\rangle\rangle, s'refXHR\rangle$
      let $s'.q := expectRegistrationResult$
    end if
  case expectRegistrationResult**:**
    let $pattern := \langle \mathtt{XMLHTTPREQUEST}, Body, s'.refXHR\rangle$
    let $input := \mathtt{CHOOSEINPUT}(scriptinputs, pattern)$
    if $input \not\equiv \mathtt{null}$ **then**
      let $RegistrationResult := input.Body[RegistrationResult]$
      if $RegistrationResult.Content.Result \not\equiv OK$ **then**
        let $s'.q := stop$
        let stop $\langle\rangle$
      end if
      let $command := \langle \mathtt{POSTMESSAGE}, target, \langle\langle RegistrationResult, RegistrationResult\rangle\rangle, \mathtt{null}\rangle$
      let $s'.q := expectProofRquest$
    end if
  case expectProofRquest**:**
    let $pattern := \langle \mathtt{POSTMESSAGE}, *, Content, *\rangle$
    let $input := \mathtt{CHOOSEINPUT}(scriptinputs, pattern)$
    if $input \not\equiv \mathtt{null}$ **then**
      let $PID_{RP} := input.Content[PID_{RP}]$
      let $Endpoint_{RP} := input.Content[Endpoint]$
      let $s'.Parameters[Nonce] := input.Content[Nonce]$
      let $Cert := s'.Parameters[Cert]$
      let $s'.Parameters[Endpoint_{RP}] := Endpoint_{RP}$
      if $Endpoint_{RP} \notin Cert.Content.Endpoints \vee PID_{RP} \not\equiv s'.Parameters[PID_{RP}]$ **then**
        let $s'.q := stop$
        let stop $\langle\rangle$
      end if

```
      let Url := ⟨URL, S, IdPDomain, /loginInfo, ⟨⟩⟩
      let s'.refXHR := Random()
      let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'refXHR⟩
      let s'.q := expectLoginState
    end if
  case expectLoginState:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      if input.Body ≡ Logged then
        let username ∈ ids
        let Url := ⟨URL, S, IdPDomain, /login, ⟨⟩⟩   mystates'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨username, username⟩, ⟨password, secret⟩⟩, s'refXHR⟩
        let s'.q := expectLoginResult
      else if input.Body ≡ Unlogged then
        let PID_RP := s'.Parameters[PID_RP]
        let Endpoint := s'.Parameters[Endpoint]
        let Nonce := s'.Parameters[Nonce]
        let Url := ⟨URL, S, IdPDomain, /authorize,
            ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩
        let s'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'refXHR⟩
        let s'.q := expectToken
      end if
    end if
  case expectLoginResult:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      if input.Body ≢ LoginSuccess then
        let stop ⟨⟩
      end if
      let PID_RP := s'.Parameters[PID_RP]
      let Endpoint := s'.Parameters[Endpoint]
      let Nonce := s'.Parameters[Nonce]
      let Url := ⟨URL, S, IdPDomain, /authorize,
          ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩
      let s'.refXHR := Random()
      let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'refXHR⟩
      let s'.q := expectToken
    end if
  case expectToken:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      let Token := input.Body[Token]
       let RPOringin := ⟨s'.Parameters[Endpoint_RP], S⟩
       let command := ⟨POSTMESSAGE, target, ⟨Token, Token⟩, RPOrigin⟩
       let s.q := stop
     end if
end switch
let stop ⟨s', cookies, localStorage, sessionStorage, command⟩
```

Algorithm 4 *script_rp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret\rangle$

```
let s' := scriptstate
let command := ⟨⟩
let IdPWindow := SUBWINDOWS(tree, AUXWINDOW(tree, docnonce)).1.nonce
let RPDomain := s'.Parameters[RPDomain]
let IdPOringin := ⟨s'.Parameters[IdPDomian], S⟩
switch s'.q do
    case start:
        let Url := ⟨URL, S, RPDomain, /login, ⟨⟩⟩
        let command := ⟨IFRAME, Url,_S ELF⟩
        let s'.q := expectN_U
    case expectN_U:
        let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let N_U := input.Content[N_U]
            let Url := ⟨URL, S, RPDomain, /startNegotiation, ⟨⟩⟩
            let s'.refXHR := Random()
            let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨N_U, N_U⟩⟩, s'refXHR⟩
            let s'.q := expectCert
        end if
    case expectCert:
        let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let Cert := input.Content[Cert]
            let command := ⟨POSTMESSAGE, IdPWindow, ⟨⟨Cert, Cert⟩⟩, IdPOringin⟩
            let s'.q := expectRegistrationResult
        end if
    case expectRegistrationResult:
        let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let RegistrationResult := input.Content[RegistrationResult]
            let Url := ⟨URL, S, RPDomain, /registrationResult, ⟨⟩⟩
            let s'.refXHR := Random()
            let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, s'refXHR⟩
            let s'.q := expectTokenRequest
        end if
    case expectTokenRequest:
        let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let PID_RP := input.Content.Body[PID_RP]
            let Endpoint := input.Content.Body[Endpoint]
            let Nonce := input.Content.Body[Nonce]
            let command := ⟨POSTMESSAGE, IdPWindow,
                ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩, IdPOringin⟩
            let s'.q := expectToken
        end if
    case expectToken:
        let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let Token := input.Content[Token]
            let Url := ⟨URL, S, RPDomain, /uploadToken, ⟨⟩⟩
            let s'.refXHR := Random()
```

```
            let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨Token, Token⟩⟩, s'refXHR⟩
            let s'.q := expectLoginResult
        end if
    case expectLoginResult:
        let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            if input.Body ≡ LoginSuccess then
                let LoadHomepage()
            end if
        end if
end switch
```