WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

Stefano Calzavara\textsuperscript{2}, Riccardo Focardi\textsuperscript{2}, Matteo Maffei\textsuperscript{1}, Clara Schneidewind\textsuperscript{1}, Marco Squarcina\textsuperscript{2}, Mauro Tempesta\textsuperscript{2}

TU Wien\textsuperscript{1}, Ca' Foscari University of Venice\textsuperscript{2}
Security of Web Protocols

Modern web applications rely on sophisticated multi-party protocols (OAuth 2.0, OpenID Connect, SAML 2.0, Shibboleth, …)

**Challenges**

- **Security model**: network attacker plus compromised JS libraries, compromised web domains, compromised servers…

- **Semantic model**: client-side (browser, JS, …), server-side (PHP, JS, …), web (DNS, …)

- Compared to standard crypto protocols, **larger attack surface and harder to verify!**
Client-Side Security (Selected State-of-the-Art)

Browser Mechanisms (SOP, CSP...)

Practical enforcement
Heuristics...no security guarantees
Client-Side Security (Selected State-of-the-Art)

Browser Mechanisms (SOP, CSP…)

Formal Methods

- In-Depth JS Information Flow Analysis (e.g., JSFlow)

Practical enforcement
Heuristics…no security guarantees

Formal local security guarantees
No end-to-end security, not yet practical
Client-Side Security (Selected State-of-the-Art)

Browser Mechanisms (SOP, CSP…)

Formal Methods

• In-Depth JS Information Flow Analysis (e.g., JSFlow)

• Web Protocol Analysis (e.g., WebSpi)

Practical enforcement
Heuristics…no security guarantees

Formal local security guarantees
No end-to-end security, impractical

Formal end-to-end security guarantees
No security enforcement in case of bugs or malicious components (e.g., XSS)
Get the best of the two worlds: **practical enforcement** and **formal end-to-end security guarantees**

via **client-side interface-based monitoring**
WPSE: Web Protocol Security Enforcer

Overall Goal

If web protocol is correct (e.g., as proven in ProVerif) and WPSE is in place, security properties carry over to a compromised page (e.g., buggy libraries, malicious scripts, XSS, etc.)

Browser extension (Chrome) that enforces

- the **intended message flow** at run-time (by dropping or modifying non-compliant requests)

- the **secrecy and integrity of sensitive data** against malicious scripts (e.g., replace cookies with random strings before storing them in the cookie jar)

Tested on various web protocols

- **New attacks on OAuth 2.0 and SAML 2.0** (500 EUR bug bounty from Google, fix required a new authentication window!)

Coming May 7th, 2018: A more secure sign-in flow on Chrome
April 25, 2018

If your organization uses SAML to sign users into G Suite services, those users will soon see an additional step in the process when using Chrome as their web browser. Starting on May 7th, 2018, after signing in on a SAML provider's website, they'll be brought to a new screen on accounts.google.com to confirm their identity. This screen will provide an additional layer of security and help prevent users from unknowingly signing in to an account created and controlled by an attacker.
OAuth 2.0
(Authorization Code Mode)

1. Identity Provider (IdP) initiates the OAuth flow.
2. Relying Party (RP) redirects the user to the Identity Provider (IdP) with a redirect URI and state parameter.
3. User credentials are submitted via a login form.
4. RP receives an authorization code and state from IdP.
5. RP exchanges the authorization code for an access token and RP id.
6. RP uses the access token to obtain a resource.

Parameter bound to U's state for CSRF protection.
Challenge 1: Message Flow

- If state parameter is not used, **session swapping attack**: honest user's browser authenticates as the attacker

  - A web attacker A initiates SSO at RP with an identity provider IdP, performing steps 1 - 3 of the protocol and learns a valid authorization code for her session.

  - A creates a page on her website that, when visited, automatically triggers a request to the redirect URI of RP and includes the authorization code.

  - When a honest user visits this page, the login procedure is completed at RP and an attacker session is established in the user's browser.
Challenge 2: Message Secrecy

- If page loaded in step 4 includes resources from malicious server, **state leak attack**
  - The authorisation code and the state parameter are leaked in the Referer Header of the outgoing request
  - The former can be used to access the honest user’s resource, the latter enables the state swapping attack
Challenge 3: Message Integrity

- If RP supports various IdPs with different redirect URIs, **naïve RP session integrity attack**: RP doesn’t know which provider is being used and user’s browser logs in as the attacker
  - Attacker logs in at *RP* with an honest identity provider *HIdP* and obtains valid authorization code for her account
  - If a honest user starts a login procedure at *RP* with *AIdP*, in step 4 *AIdP* redirects to the redirect URI of *HIdP* with the authorization code from the attacker session
  - *RP* completes the login with *HIdP* using the attacker’s account
WPSE at Work

Request to a redirect URI uri1

\[\phi_1 \triangleq G(\text{response_type:code, redirect_uri:}^\text{origin}{(https?://.*?):(?:|\?|$)})\]

\[\phi_2 \triangleq G(\text{Location:[?&]code=}(.*?)(?:|$))\]

\[\phi_3 \triangleq (\text{code:}([^\s]{40},))\]

\[\pi_S \triangleq \text{authcode} \rightarrow \{\text{https://accounts.google.com,origin}\}\]

\[\pi_I \triangleq \text{uri1} = \text{uri2}\]

Automaton for OAuth 2.0 (authorization code mode) where \(G\) is the OAuth endpoint at Google.

**Secrecy policy**: authorization code disclosed only to selected origins (enforced by replacing it with random placeholder)

**Integrity policy**: the browser is redirected to the URI specified in the first protocol step

Response with a location header that contains a URL with a parameter named code
Attacks in Scope

<table>
<thead>
<tr>
<th>Detected Violation</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol flow deviation</td>
<td>Session swapping [43]</td>
</tr>
<tr>
<td></td>
<td>Social login CSRF on stateless clients [6]</td>
</tr>
<tr>
<td></td>
<td>IdP mix-up attack (web attacker) [19]</td>
</tr>
<tr>
<td>Secrecy violation</td>
<td>Unauthorized login by authentication code redirection [6]</td>
</tr>
<tr>
<td></td>
<td>Resource theft by access token redirection [6]</td>
</tr>
<tr>
<td></td>
<td>307 redirect attack [19]</td>
</tr>
<tr>
<td></td>
<td>State leak attack [19]</td>
</tr>
<tr>
<td>Integrity violation</td>
<td>Cross social-network request forgery [6]</td>
</tr>
<tr>
<td></td>
<td>Naïve RP session integrity attack [19]</td>
</tr>
</tbody>
</table>

- All prevented by WPSE
Attacks out of Scope

- Attacks **without deviations from the expected protocol flow** (e.g., automatic login CSRF)
  - Can be prevented by introducing CSRF protection

- Attacks whose **protocol deviations cannot be detected on the browser** (e.g., network attacks like the IdP mix-up attack)
  - Can be prevented by using HTTPS, preferably with HSTS

- Attacks which **do not involve the browser** at all (e.g., impersonation attack where public information is used for authentication or server-side attacks)
Experimental Evaluation

- **Setup**: 3 top IdPs (Facebook, Google, VK), 30 RPs each

- **Confidentiality**: sensitive data leaked on 4 RPs, due to the presence of tracking libraries

- **Integrity**: 55 out of 90 affected by lack or misuse of state parameter (41 do not support it, 14 use a predictable or even constant string)

- **Compatibility**: 83 flawless navigation, 7 break due to security critical deviations from protocol standard (e.g., sending authorization code via HTTP)
New Attacks

• OAuth 2.0: access token leaked by tracking libraries, allowing for arbitrary access to user data
  • Not seen as critical vulnerability, as "it is a responsibility of web developers to include the tracking library only in pages without sensitive contents"

• SAML 2.0: lack of contextual binding between messages, allowing a web attacker to authenticate any user on Google’s suite applications under the attacker’s account, similar to a Login CSRF
  • Bug bounty and significant patch by Google
Formal Security Result
(formalized in the applied pi-calculus)

If

(H1) the protocol fulfils safety property $P$ with a benign webpage (original protocol security)

(H2) the monitor $M$ allows the browser to perform only a subset of the input/output sequences possibly performed by the browser itself in a honest protocol run (message flow monitoring)

(H3) secrets (like keys and cookies) are not leaked and securely stored by the browser (confidentiality and integrity monitoring)

Then

the protocols fulfils safety property $P$ with a compromised browser monitored by our extension

E.g., check that with ProVerif

Standard program verification task

We design clever (backward compatible) techniques to do that
Lightweight policies on the client-side suffice to enforce provable security guarantees in security critical protocols.

Challenges ahead:

- **Automatic synthesis of policies** from protocol specs
- **Embed policy checking into the browser**: find sweet spot between expressiveness and performance
  - **How?** Trusted types, capability-based enforcement, …
We are hiring PhDs and Postdocs!