WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

Stefano Calzavara
Università Ca' Foscari Venezia
calzavara@dais.unive.it

Riccardo Focardi
Università Ca' Foscari Venezia
focardi@unive.it

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Clara Schneidewind
TU Wien
clara.schneidewind@tuwien.ac.at

Marco Squarcina
Università Ca' Foscari Venezia
squarcina@unive.it

Mauro Tempesta
Università Ca' Foscari Venezia
tempesta@unive.it

Abstract

We present WPSE, a browser-side security monitor for web protocols designed to ensure compliance with the intended protocol flow, as well as confidentiality and integrity properties of messages. We formally prove that WPSE is expressive enough to protect web applications from a wide range of protocol implementation bugs and web attacks. We discuss concrete examples of attacks which can be prevented by WPSE on OAuth 2.0 and SAML 2.0, including a novel attack on the Google implementation of SAML 2.0 which we discovered by formalizing the protocol specification in WPSE. Moreover, we use WPSE to carry out an extensive experimental evaluation of OAuth 2.0 in the wild. Out of 90 tested websites, we identify security flaws in 55 websites (61.1%), including new critical vulnerabilities introduced by tracking libraries such as Facebook Pixel, all of which fixable by WPSE. Finally, we show that WPSE works flawlessly on 83 websites (92.2%), with the 7 compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability.

1 Introduction

Web protocols are security protocols deployed on top of HTTP and HTTPS, most notably to implement authentication and authorization at remote servers. Popular examples of web protocols include OAuth 2.0, OpenID Connect, SAML 2.0 and Shibboleth, which are routinely used by millions of users to access security-sensitive functionalities on their personal accounts.

Unfortunately, designing and implementing web protocols is a particular error-prone task even for security experts, as witnessed by the large number of vulnerabilities reported in the literature [43, 6, 5, 50, 28, 27, 48, 46]. The main reason for this is that web protocols involve communication with a web browser, which does not strictly follow the protocol specification, but reacts asynchronously to any input it receives, producing messages which may have an import on protocol security. Reactiveness is dangerous because the browser is agnostic to the web protocol semantics: it does not know when the protocol starts, nor when it ends, and is unaware of the order in which messages should be processed, as well as of the confidentiality and integrity guarantees desired for a protocol run. For example, in the context of OAuth 2.0, Bansal *et al.* [6] discussed *token redirection attacks* enabled by the presence of open redirectors, while Fett *et al.* [19] presented *state leak attacks* enabled by the communication of the Referer header; these attacks are not apparent from the protocol specification alone, but come from the subtleties of the browser behaviour.

Major service providers try to aid software developers to correctly integrate web protocols in their websites by means of JavaScript APIs; however, web developers are not forced to use them, can still use them incorrectly [47], and the APIs themselves do not necessarily implement the best security practices [43]. This unfortunate situation led to the proliferation of attacks against web protocols even at popular services.

In this paper, we propose a fundamental paradigm shift to strengthen the security guarantees of web protocols. The key idea we put forward is to extend browsers with a security monitor which is able to enforce the compliance of browser behaviours with respect to the web protocol specification. This approach brings two main benefits:

1. web applications are automatically protected against a large class of bugs and vulnerabilities on the browser-side, since the browser is aware of the intended protocol flow and any deviation from it is detected at runtime;

2. protocol specifications can be written and verified once, possibly as a community effort, and then uniformly enforced at a number of different websites by the browser.

Remarkably, though changing the behaviour of web browsers is always delicate for backward compatibility, the security monitor we propose is carefully designed to interact gracefully with existing websites, so that the website functionality is preserved unless it critically deviates from the intended protocol specification. Moreover, a large set of the monitor functionalities can be implemented as a browser extension, thereby offering immediate protection to Internet users and promising a significant practical impact.

1.1 Contributions

In this paper, we make the following contributions:

1. we identify three fundamental browser-side security properties for web protocols, that is, the *confidentiality* and *integrity* of message components, as well as the compliance with the intended *protocol flow*. We discuss concrete examples of their import for the popular authorization protocol OAuth 2.0;

2. we semantically characterize these properties and formally prove that their enforcement suffices to protect the web application from a wide range of protocol implementation bugs and attacks on the application code running in the browser;

3. we propose the Web Protocol Security Enforcer, or WPSE for short, a browser-side security monitor designed to enforce the aforementioned security properties, which we implement as a publicly available Google Chrome extension;

4. we experimentally assess the effectiveness of WPSE by testing it against 90 popular websites making use of OAuth 2.0 to implement single sign-on at major identity providers. In our analysis, we identified security flaws in 55 websites (61.1%), including new critical vulnerabilities caused by tracking libraries such as Facebook Pixel, all of which fixable by WPSE. We show that WPSE works flawlessly on 83 websites (92.2%), with the 7 compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability;

5. to show the generality of our approach, we also considered SAML 2.0, a popular web authorization protocol: while formalizing its specification, we found a new attack on the Google implementation of SAML 2.0 that has been awarded a bug bounty according to the Google Vulnerability Reward Program.[1]

[1] `https://www.google.com/about/appsecurity/reward-program/`

2 Security Challenges in Web Protocols

The design of web protocols comes with various security challenges which can often be attributed to the presence of the web browser that acts as a non-standard protocol participant. In the following, we discuss three crucial challenges, using the OAuth 2.0 authorization protocol as illustrative example.

2.1 Background on OAuth 2.0

OAuth 2.0 [25] is a web protocol that enables resource owners to grant controlled access to resources hosted at remote servers. Typically, OAuth 2.0 is also used for authenticating the resource owner to third parties by giving them access to the resource owner's identity stored at an identity provider. This functionality is known as Single Sign-On (SSO). Using standard terminology, we refer to the third-party application as *relying party* (*RP*) and to the website storing the resources, including the identity, as *identity provider* (*IdP*).[2]

The OAuth 2.0 specification defines four different protocol flows, also known as *grant types* or *modes*. We focus on the *authorization code* mode and the *implicit* mode since they are the most commonly used by websites.

The authorization code mode is intended for a *RP* whose main functionality is carried out at the server side. The high-level protocol flow is depicted in Figure 1. For the sake of readability, we introduce a simplified version of the protocol abstracting from some implementation details that are presented in Section 4.1. The protocol works as follows:

① the user U sends a request to *RP* for accessing a remote resource. The request specifies the *IdP* that holds the resource. In the case of SSO, this step determines which *IdP* should be used;

② *RP* redirects U to the login endpoint of *IdP*. This request contains the *RP*'s identity at *IdP*, the URI that *IdP* should redirect to after successful login and an optional state parameter for CSRF protection that should be bound to U's state;

③ *IdP* answers to the authorization request with a login form and the user provides her credentials;

④ *IdP* redirects U to the URI of *RP* specified at step ②, including the previously received state parameter and an authorization code;

[2] The OAuth 2.0 specification distinguishes between *resource servers* and *authorization servers* instead of considering one identity provider that stores the user's identity as well as its resources [25], but it is common practice to unify resource and authorization servers as one party [19, 43, 27].

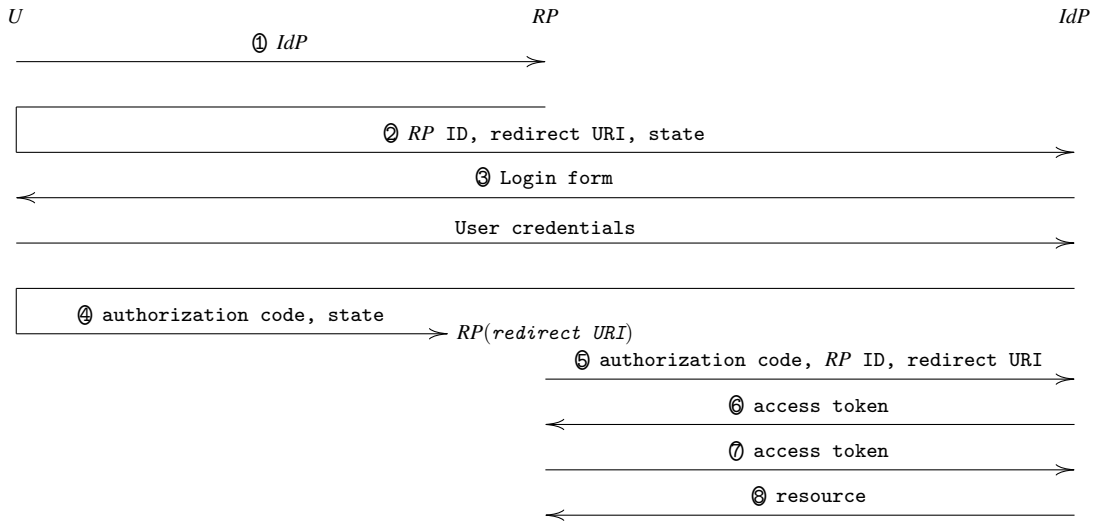

Figure 1: OAuth 2.0 (authorization code mode).

⑤ *RP* makes a request to *IdP* with the authorization code, including its identity, the redirect URI and optionally a shared secret with the *IdP*;

⑥ *IdP* answers with an access token to *RP*;

⑦ *RP* makes a request for the user's resource to *IdP*, including the access token;

⑧ *IdP* answers *RP* with the user's resource at *IdP*.

The implicit mode differs from the authorization code mode in steps ④-⑥. Instead of granting an authorization code to *RP*, the *IdP* provides an access token in the fragment identifier of the redirect URI. A piece of JavaScript code embedded in the page located at the redirect URI extracts the access token and communicates it to the *RP*.

2.2 Challenge #1: Protocol Flow

Protocols are specified in terms of a number of sequential message exchanges which honest participants are expected to follow, but the browser is not forced to comply with the intended protocol flow.

Example in OAuth 2.0. The use of the state parameter is recommended to prevent attacks leveraging this idiosyncrasy. When OAuth is used to implement SSO and *RP* does not provide the state parameter in its authorization request to *IdP* at step ②, it is possible to force the honest user's browser to authenticate as the attacker. This attack is known as *session swapping* [43].

We give a short overview on this attack against the authorization code mode. A web attacker *A* initiates SSO at *RP* with an identity provider *IdP*, performs steps ①-③ of the protocol and learns a valid authorization code for her session. Next, *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.

2.3 Challenge #2: Secrecy of Messages

The security of protocols typically relies on the confidentiality of cryptographic keys and credentials, but the browser is not aware of which data must be kept secret for protocol security.

Example in OAuth 2.0. The secrecy of the authorization credentials (namely authorization codes and access tokens) is crucial for meeting the protocol security requirements, since their knowledge allows an attacker to access the user's resources. The secrecy of the state parameter is also important to ensure session integrity.

An example of an unintended secrets leakage is the *state leak* attack described in [19]. If the page loaded at the redirect URI in step ④ loads a resource from a malicious server, the state parameter and the authorization code (that are part of the URL) are leaked in the Referer header of the outgoing request. The learned authorization code can potentially be used to obtain a valid access token for *U* at *IdP*, while the leaked state parameter enables the session swapping attack discussed previously.

2.4 Challenge #3: Integrity of Messages

Protocol participants are typically expected to perform a number of runtime checks to prove the integrity of the messages they receive and ensure the integrity of the messages they send, but the browser cannot perform

these checks unless they are explicitly carried out in a JavaScript implementation of the web protocol.

Example in OAuth 2.0. An attack that exploits this weakness is the *naïve RP session integrity* attack presented in [19]. Suppose that *RP* supports SSO with various identity providers and uses different redirect URIs to distinguish between them. In this case, an attacker controlling a malicious identity provider *AIdP* can confuse the *RP* about which provider is being used and force the user's browser to login as the attacker.

To this end, the attacker starts a SSO login at *RP* with an honest identity provider *HIdP* to obtain a valid authorization code for her account. If a honest user starts a login procedure at *RP* with *AIdP*, in step ④ *AIdP* is expected to redirect the user to *AIdP*'s redirect URI at *RP*. If *AIdP* redirects to the redirect URI of *HIdP* with the authorization code from the attacker session, then *RP* mistakenly assumes that the user intended to login with *HIdP*. Therefore, *RP* completes the login with *HIdP* using the attacker's account.

3 WPSE: Design and Implementation

The Web Protocol Security Enforcer (WPSE) is the first browser-side security monitor addressing the peculiar challenges of web protocols. The current prototype is implemented as an extension for Google Chrome, which we make available online.[3]

3.1 Key Ideas of WPSE

We illustrate WPSE on the authorization code mode of OAuth 2.0, where Google is used as identity provider and the state parameter is not used (since it is not mandatory at Google). For simplicity, here we show only the most common scenario where the user has an ongoing session with the identity provider and the authorization to access the user's resources on the provider has been previously granted to the relying party.

3.1.1 Protocol Flow

WPSE describes web protocols in terms of the HTTP(S) exchanges observed by the web browser, following the so-called *browser relayed messages* methodology first introduced by Wang *et al.* [46]. The specification of the protocol flow defines the syntactic structure and the expected (sequential) order of the HTTP(S) messages, supporting the choice of different execution branches when a particular protocol message is sent or received by the browser. The protocol specification is given in XML (*cf.* Appendix A), but for the sake of readability, we use in this paper an equivalent representation in terms of finite state automata, like the one depicted in Figure 2. Intuitively, each state of the automaton represents one stage of the protocol execution in the browser. By sending an HTTP(S) request or receiving an HTTP(S) response as dictated by the protocol, the automaton steps to the next state until it reaches a final state denoting the end of the protocol run. Afterwards, the automaton moves back to the initial state and a new protocol run can start.

The edges of the automaton are labeled with *message patterns*, describing the expected shape of the protocol messages at each state. We represent HTTP(S) requests as $e\langle a\rangle$, where e is the remote endpoint to which the message is sent and a is a list of parameters, while HTTP(S) responses are noted $e(h)$, where e is the remote endpoint from which the message is received and h is a list of headers.[4] The syntactic structure of e, a, h can be described using regular expressions. The message patterns should be considered as *guards* of the transition, which are only enabled for messages matching the pattern. For instance, the pattern ϕ_2 in Figure 2 matches a response from the endpoint G with a `Location` header that contains a URL with a parameter named `code`. If an HTTP(S) request or response does not satisfy any of the patterns of the outgoing transitions of the current state, it is blocked and the automaton is reset to the initial state, *i.e.*, the protocol run is aborted. In case of branches with more than one transition enabled at a given state, we solve the non-determinism by picking the first transition (with a matching pattern) according to the order defined in the XML specification. Patterns can be composed using standard logical connectives.

Each state of the automaton also allows for pausing the protocol execution in presence of requests and responses that are unrelated to the protocol. Messages are considered unrelated to the protocol if they are not of the shape of any valid message in the protocol specification. In the automaton, this is expressed by having a self-loop for each state, labeled with the negated disjunction of all patterns describing valid protocol messages. This is important for website functionality, because the input/output behavior of browsers on realistic websites is complex and hard to fully determine when writing a protocol specification. Also, the same protocol may be run on different websites, which need to fetch different resources as part of their protocol-unrelated functionalities, and we would like to ensure that the same protocol specification can be enforced uniformly on all these websites.

3.1.2 Security Policies

To incorporate secrecy and integrity policies in the automaton, we allow for binding parts of message patterns

[3] `https://sites.google.com/site/wpseproject/`

[4] We support HTTP headers also in requests. Here we omit them since they are not used in the protocols that we consider.

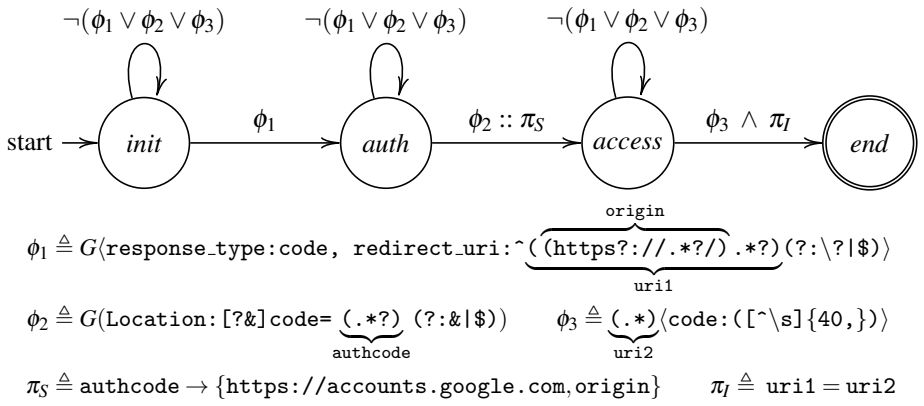

Figure 2: Automaton for OAuth 2.0 (authorization code mode) where G is the OAuth endpoint at Google.

to *identifiers*. For instance, in Figure 2 we bind the identifier `origin` to the content of the `redirect_uri` parameter, more precisely to the part matching the regular expression group `(https?://.*?/)`.[5] The scope of an identifier includes the state where it is first introduced and all its successor states, where the notion of successor is induced by the tree structure of the automaton. For instance, the scope of the identifier `origin` introduced in ϕ_1 includes the states *auth*, *access*, *end*.

The *secrecy policy* defines which parts of the HTTP(S) responses included in the protocol specification must be confidential among a set of web origins. We express secrecy policies π_S with the notation $\mathtt{x} \to S$ to denote that the value bound to the identifier `x` can be disclosed only to the origins specified in the set S. We call S the *secrecy set* of identifier `x` and represent such a policy on the message pattern where the identifier `x` is first introduced, using a double colon symbol :: as a separator. For instance, in Figure 2 we require that the value of the authorization code, which is bound to the identifier `authcode` introduced in ϕ_2, can be disclosed only to Google (at `https://accounts.google.com`) and the relying party (bound to the identifier `origin`). Confidential message components are stripped from HTTP(S) responses and substituted by random placeholders, so that they are isolated from browser accesses, *e.g.*, computations performed by JavaScript. When the automaton detects an HTTP(S) request including one of the generated placeholders, it replaces the latter with the corresponding original value, but only if the HTTP(S) request is directed to one of the origins which is entitled to learn it. A similar idea was explored by Stock and Johns to strengthen the security of password managers [42]. Since the substitution of confidential message components with placeholders changes the content of the messages, potentially introducing deviations with respect to the transition labels, the automaton processes HTTP(S) responses before stripping confidential values and HTTP(S) requests after replacing the placeholders with the original values. This way, the input/output behavior of the automaton matches the protocol specification.

The *integrity policy* defines runtime checks over the HTTP(S) messages. These checks allow for the comparison of incoming messages with the messages received in previous steps of the protocol execution. If any of the integrity checks fails, the corresponding message is not processed and the protocol run is aborted. To express integrity policies π_I in the automaton, we enrich the message patterns to include comparisons ranging over the identifiers introduced by preceding messages. In the case of OAuth 2.0, we would like to ensure that the browser is redirected by the *IdP* to the redirect URI specified in the first step of the protocol. Therefore, in Figure 2 the desired integrity policy is modeled by the condition $\mathtt{uri1} = \mathtt{uri2}$.

3.1.3 Enforcing Multiple Protocols

There are a couple of delicate points to address when multiple protocol specifications $P_1, \ldots, P_n$ must be enforced by WPSE:

1. if two different protocols P_i and P_j share messages with the same structure, there might be situations where WPSE does not know which of the two protocols is being run, yet a message may be allowed by P_i and disallowed by P_j or vice-versa;

2. if WPSE is enforcing a protocol P_i, it must block any message which may be part of another protocol P_j, otherwise it would be trivial to sidestep the security policy of P_i by first making the browser process the first message of P_j.

Both problems are solved by replacing the protocol specifications $P_1, \ldots, P_n$ with a single specification P with n

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp`

branches, one for each P_i. Using this construction, any ambiguity on which protocol specification should be enforced is solved by the determinism of the resulting finite state automaton. Moreover, the self loops of the automaton will only match the messages which are not part of any of the n protocol specifications, thereby preventing unintended protocol interleavings. Notice that the semantics of WPSE depends on the order of $P_1, \ldots, P_n$, due to the way we enforce determinism on the compiled automaton: if P_i starts with a request to u including two parameters a and b, while P_j starts with a request to u including just the parameter a, then P_i should occur before P_j to ensure it is actually taken into account.

3.2 Discussion

A number of points of the design and the implementation of WPSE are worth discussing more in detail.

3.2.1 Protocol Flow

WPSE provides a significant improvement in security over standard web browsers, as we show in the remainder of the paper, but the protection it offers is not for free, because it requires the specification of a protocol flow and a security policy. We think that it is possible to develop automated techniques to reconstruct the intended protocol flow from observable browser behaviours, while synthesizing the security policy looks more difficult. Manually finding the best security policy for a protocol may require significant expertise, but even simple policies can be useful to prevent a number of dangerous attacks, as we demonstrate in Section 4.

The specification style of the protocol flow supported by WPSE is simple, because it only allows sequential composition of messages and branching. As a result, our finite state automata are significantly simpler than the request graphs proposed by Guha *et al.* [24] to represent legitimate browser behaviors (from the server perspective). For instance, our finite state automata do not include loops and interleaving of messages, because it seems that these features are not extensively used in web protocols. Like standard security protocols, web protocols are typically specified in terms of a fixed number of sequential messages, which are appropriately supported by the specification language we chose.

3.2.2 Secrecy Enforcement

The implementation of the secrecy policies of WPSE is robust, but restrictive. Since WPSE substitutes confidential values with random placeholders, only the latter are exposed to browser-side scripts. Shielding secret values from script accesses is crucial to prevent confidentiality breaches via untrusted scripts or XSS, but it might also break the website functionality if a trusted script needs to compute over a secret value exchanged in the protocol. The current design of WPSE only supports a limited use of secrets by browser-side scripts, *i.e.*, scripts can only forward secrets unchanged to the web origins entitled to learn them. We empirically show that this is enough to support existing protocols like OAuth 2.0 and SAML, but other protocols may require more flexibility.

Dynamic information flow control deals with the problem of letting programs compute over secret values while avoiding confidentiality breaches and it has been applied in the context of web browsers [21, 26, 8, 36, 7]. We believe that dynamic information flow control can be fruitfully combined with WPSE to support more flexible secrecy policies. This integration can also be useful to provide confidentiality guarantees for values which are generated at the browser-side and sent in HTTP(S) requests, rather than received in HTTP(S) responses. We leave the study of the integration of dynamic information flow control into WPSE to future work.

3.2.3 Extension APIs

The current prototype of WPSE suffers from some limitations due to the Google Chrome extension APIs. In particular, the body of HTTP messages cannot be modified by extensions, hence the secrecy policy cannot be implemented when secret values are embedded in the page contents or the corresponding placeholders are sent as POST parameters. Currently, we protect secret values contained in the HTTP headers of a response (*e.g.*, cookies or parameters in the URL of a `Location` header) and we only substitute the corresponding placeholders when they are communicated via HTTP headers or as URL parameters. Clearly this is not a limitation of our general approach but rather one of the extension APIs, which can be solved by implementing the security monitor directly in the browser or as a separate proxy application. Despite these limitations, we were able to test the current prototype of WPSE on a number of real-world websites with very promising results, as reported in Section 5.

4 Fortifying Web Protocols with WPSE

To better appreciate the security guarantees offered by WPSE, we consider two popular web protocols: OAuth 2.0 and SAML. The security of both protocols has already been studied in depth, so they are an excellent benchmark to assess the effectiveness of WPSE: we refer to [6, 19, 43] for security analyses of OAuth 2.0 and to [3, 4] for research studies on SAML. Remarkably, by writing down a precise security policy for SAML, we were able to expose a new critical attack against the Google implementation of the protocol.

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

Table 1: Overview of the attacks against OAuth 2.0.

4.1 Attacks Against OAuth 2.0

We review in this section several attacks on OAuth 2.0 from the literature, analysing whether they are prevented by our extension. We focus in particular on those presented in [6, 19, 43], since they apply to the OAuth 2.0 flows presented in this work. In Table 1 we provide an overview of the attacks that WPSE is able to prevent, grouped according to the type of violation of the security properties that they expose.

4.1.1 Protocol Flow Deviations

This category covers attacks that force the user's browser to skip messages or to accept them in a wrong order. For instance, some attacks, *e.g.*, some variants of CSRF and session swapping, rely on completing a social login in the user's browser that was not initiated before. This is a clear deviation from the intended protocol flow and, as a consequence, WPSE blocks these attacks.

We exemplify on the session swapping attack discussed in Section 2.2. Here the attacker tricks the user into sending a request containing the attacker's authorization credential (*e.g.*, the authorization code) to *RP* (step ④ of the protocol flow). Since the state parameter is not used, the *RP* cannot verify whether this request was preceded by a social login request by the user. Our security monitor blocks the (out-of-order) request since it matches the pattern ϕ_3, which is allowed by the automaton in Figure 2 only in state *access*. Thus, the attack is successfully prevented.

4.1.2 Secrecy Violations

This category covers attacks where sensitive information is unintentionally leaked, *e.g.*, via the Referer header or because of the presence of open redirectors at *RP*. Sensitive data can either be leaked to untrusted third parties that should not be involved in the protocol flow (as in the state leak attack) or protocol parties that are not trusted for a specific secret (as in the 307 redirect attack). WPSE can prevent this class of attacks since the secrecy policy allows one to specify the origins that are entitled to receive a secret.

We illustrate how the monitor prevents these attacks in case of the state leak attack discussed in Section 2.3, focusing on the authorization code. In the attack, the authorization code is leaked via the Referer header of the request fetching a resource from the attacker website which is embedded in the page located at the redirect URI of *RP* (step ④ of the protocol). When the authorization code (`authcode`) is received (step ②), the monitor extracts it from the Location header and replaces it with a random placeholder before the request is processed by the browser. After step ④, the request to the attacker's website is sent, but the monitor does not replace the placeholder with the actual value of the authorization code since the secrecy set associated to `authcode` in π_S does not include the domain of the attacker.

4.1.3 Integrity Violations

This category contains attacks that maintain the general protocol flow, but the contents of the exchanged messages do not satisfy some integrity constraints required by the protocol. WPSE can prevent these attacks by enforcing browser-side integrity checks.

Consider the naïve RP session integrity attack presented in Section 2.4. In this attack, the malicious identity provider *AIdP* redirects the user's browser to the redirect URI of the honest identity provider *HIdP* at *RP* during step ④ of the protocol. At step ②, the redirect URI is provided to *AIdP* as parameter. This request corresponds to the pattern ϕ_1 of the automation and the redirect URI associated to *AIdP* is bound to the identifier `uri1`. At step ④, *AIdP* redirects the browser to a different redirect URI, which is bound to the identifier `uri2`. Although the shape of the request satisfies pattern ϕ_3, the monitor cannot move from state *access* to state *end* since the constraint `uri1 = uri2` in the integrity policy π_I is violated. Thus, no transition is enabled for the state *access* and the request is blocked by WPSE, therefore preventing the attack.

4.2 Attacks Against SAML

The *Security Assertion Markup Language* (SAML) 2.0 [34] is an open standard for sharing authentication and authorization across a multitude of domains. SAML is based on XML messages called *assertions* and defines different *profiles* to account for a variety of use cases and

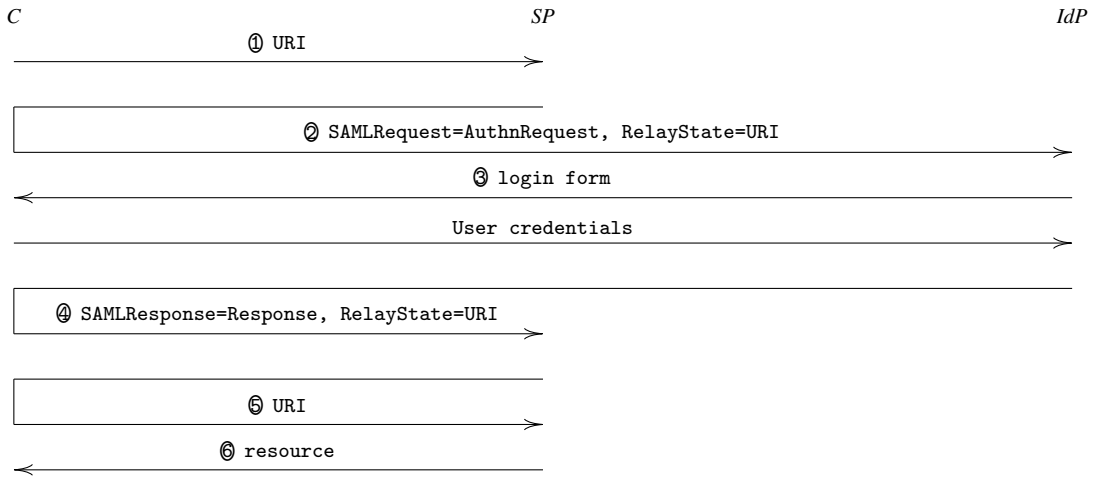

Figure 3: SAML 2.0 SP-Initiated SSO with Redirect/POST Bindings.

deployment scenarios. SSO functionality is enabled by the SAML 2.0 web browser SSO profile, whose typical use case is the *SP*-Initiated SSO with Redirect/POST Bindings [33, 4]. Similarly to OAuth 2.0, there are three entities involved: a user controlling a web browser (*C*), an identity provider (*IdP*) and a service provider (*SP*). The protocol prescribes how *C* can access a resource provided by an *SP* after authenticating with an *IdP*.

The relevant steps of the protocol are depicted in Figure 3. In step ①, *C* requests from *SP* the resource located at URI; in ② the *SP* redirects the browser to the *IdP* sending an *AuthnRequest* XML message in deflated, base64-encoded form and a *RelayState* parameter; *C* provides his credentials to the *IdP* in step ③ where they are verified; in step ④ the *IdP* causes the browser to issue a POST request to the Assertion Consumer Service at the *SP* containing the base64-encoded *SamlResponse* and the *RelayState* parameters; in ⑤ the *SP* processes the response, creates a security context at the service provider and redirects *C* to the target resource at URI; given that a security context is in place, the *SP* provider returns the resource to *C*.

The RelayState is a mechanism for preserving some state information at the *SP*, such as the resource URI requested by the user [20]. If the RelayState parameter is used within a request message, then subsequent responses must maintain the exact value received with the request [35]. A violation of this constraint enables attacks such as [3], in which *C* requests a resource URI_i at a malicious SP_i. SP_i pretends to be *C* at the honest *SP* and requests a different resource at *SP* located at URI_{SP} which is returned to SP_i. The malicious service provider replies to *C* by providing a redirection address containing a different resource URI, thus causing the browser to send URI_i instead of instead of URI as the value of RelayState at steps ②,④. The result is that *C* forcibly accesses a resource at *SP*, while he originally asked for a resource from SP_i.

Interestingly, by using WPSE it is possible to instruct the browser with knowledge of the protocol in such a way that the client can verify whether the requests at steps ②,④ are related to the initial request. We distilled a simple policy for the SAML 2.0 web browser SSO profile that enforces an integrity constraint on the value of the RelayState parameter, thus blocking requests to undesired resources due to a violation of the policy.

Furthermore, SAML 2.0 does not specify any way to maintain a contextual binding between the request at step ② and the request at step ④. It follows that only the *SAMLResponse* and *RelayState* parameters are enough to allow *C* to access the resource at URI. We discovered that this shortcoming in the protocol has a critical impact on real *SP*s using the SAML-based SSO profile described in this section. Indeed, we managed to mount an attack against Google that allows a web attacker to authenticate any user on Google's suite applications under the attacker's account, with effects similar to a Login CSRF attack. Since Google can act as a Service Provider (*SP*) with a third party *IdP*, an attacker registered to a given *IdP* can simulate a login attempt with his legitimate credentials to obtain a valid POST request to the Google assertion consumer service (step ④). Once accessed, a malicious web page can then cause a victim's browser to issue the attacker's request to the Google assertion consumer service, thus forcing the victim inside the attacker's controlled authenticated session.

The vulnerability can be exploited by any web attacker with a valid account on a third party *IdP* that uses Google as *SP*. In particular, our university uses SAML 2.0 with Google as a Service provider to offer email and storage

facilities to students and employees. We have implemented the attack by constructing a malicious webpage that silently performs a login on Google's suite applications using one of our personal accounts. The vulnerability allows the attacker to access private information of the victim that has been saved in the account, such as activity history, notes and documents. We have responsibly reported this vulnerability to Google who rewarded us according to their bug bounty program. As soon as they are available, we will provide on our website the details of the fixes that Google is implementing to resolve the issue [14].

From the browser standpoint, this attack is clearly caused by a violation of the protocol flow given that steps ①-③ are carried out by the attacker and step ④ and subsequent ones involve the victim. WPSE identifies the outgoing request to the *IdP* as a protocol flow deviation, thereby preventing the attack.

4.3 Out-of-Scope Attacks

We have shown that WPSE is able to block a wide range of attacks on existing web protocols. However, some classes of attacks cannot be prevented by browser-side security monitoring. Specifically, WPSE cannot prevent:

1. attacks which do not deviate from the expected protocol flow. An example of such an attack against OAuth 2.0 is the *automatic login CSRF* attack presented in [6], which exploits the lack of CSRF protection on the login form of the relying party to force an authentication to the identity provider. This class of attacks can be prevented by implementing appropriate defenses against known web attacks;

2. attacks which cause deviations from the expected protocol flow that are not observable by the browser. In particular, this class of attacks includes *network attacks*, where the attacker corrupts the traffic exchanged between the protocol participants. For instance, a network attacker can run the *IdP mix-up* attack from [19] when the first step of OAuth 2.0 is performed over HTTP. This class of attacks can be prevented by making use of HTTPS, preferably backed up by HSTS;

3. attacks which do not involve the user's browser at all. An example is the *impersonation* attack on OAuth 2.0 discussed in [43], where public information is used for authentication. Another example is the *DuoSec* vulnerability found on several SAML implementations [30] that exploits a bug in the XML libraries used by SPs to parse SAML messages. This class of attacks must be necessarily solved at the server side.

5 Experimental Evaluation

Having discussed how WPSE can prevent several real-world attacks presented in the literature, we finally move to on-field experiments. The goal of the present section is assessing the practical security benefits offered by WPSE on existing websites in the wild, as well as to test the compatibility of its browser-side security monitoring with current web technologies and programming practices. To this end, we experimentally assessed the effectiveness of WPSE by testing it against websites using OAuth 2.0 to implement SSO at high-profile *IdP*s.

5.1 Experimental Setup

We developed a crawler to automatically identify existing OAuth 2.0 implementations in the wild. Our analysis is not meant to provide a comprehensive coverage of the deployment of OAuth 2.0 on the web, but just to identify a few popular identity providers and their relying parties to carry out a first experimental evaluation of WPSE.

We started from a comprehensive list of OAuth 2.0 identity providers[6] and we collected for each of them the list of the HTTP(S) endpoints used in their implementation of the protocol. Inspired by [45], our crawler looks for login pages on websites to find syntactic occurrences of these endpoints: after accessing a homepage, the crawler extracts a list of (at most) 10 links which may likely point to a login page, using a simple heuristic. It also retrieves, using the Bing search engine, the 5 most popular pages of the website. For all these pages, the crawler checks for the presence of the OAuth 2.0 endpoints in the HTML code and in the 5 topmost scripts included by them. By running our crawler on the Alexa 100k top websites, we found that Facebook (1,666 websites), Google (1,071 websites) and VK (403 websites) are the most popular identity providers in the wild.

We then developed a faithful XML representation of the OAuth 2.0 implementations available at the selected identity providers. There is obviously a large overlap between these specifications, though slight differences are present in practice, *e.g.*, the use of the `response_type` parameter is mandatory at Google, but can be omitted at Facebook and VK to default to the authorization code mode. For the sake of simplicity, we decided to model the most common use case of OAuth 2.0, *i.e.*, we assume that the user has an ongoing session with the identity provider and that authorization to access the user's resources on the provider has been previously granted to the relying party. For each identity provider we devised a specification that supports the OAuth 2.0 authorization code and implicit modes, with and without the optional

[6] `https://en.wikipedia.org/wiki/List_of_OAuth_providers`

state parameter, leading to 4 possible execution paths. Finally, we created a dataset of 90 websites by sampling 30 relying parties for each identity provider, covering both the authorization code mode and the implicit mode of OAuth 2.0. We have manually visited these websites with a browser running WPSE both to verify if the protocol run was completed successfully and to assess whether all the functionalities of the sites were working properly. In the following we report on the results of testing our extension against these websites from both a security and a compatibility point of view.

5.2 Security Analysis

We devised an automated technique to check whether WPSE can stop dangerous real-world attacks. Since we did not want to attack the websites, we focused on two classes of vulnerabilities which are easy to detect just by navigating the websites when using WPSE. The first class of vulnerabilities enables confidentiality violations: it is found when one of the placeholders generated by WPSE to enforce its secrecy policies is sent to an unintended web origin. The second class of vulnerabilities, instead, is related to the use of the state parameter: if the state parameter is unused or set to a predictable static value, then session swapping becomes possible (see Section 2.2). We can detect these cases by checking which protocol specification is enforced by WPSE and by making the state parameter secret, so that all the values bound to it are collected by WPSE when they are substituted by the placeholders used to enforce the secrecy policy.

We observed that our extension prevented the leakage of sensitive data on 4 different relying parties. Interestingly, we found that the security violation exposed by the tool are in all cases due to the presence of tracking or advertisements libraries such as Facebook Pixel,[7] Google AdSense,[8] Heap[9] and others. For example, this has been observed on `ticktick.com`, a website offering collaborative task management tools. The leakage is enabled by two conditions:

1. the website allows its users to perform a login via Google using the implicit mode;
2. the Facebook tracking library is embedded in the page which serves as redirect URI.

Under these settings, right after step ④ of the protocol, the tracking library sends a request to `https://www.facebook.com/tr/` with the full URL of the current page, which includes the access token issued by Google. We argue that this is a critical vulnerability, given that leaking the access token to an unauthorized party allows unintended access to sensitive data owned by the users of the affected website. We promptly reported the issue to the major tracking library vendors and the vulnerable websites. Library vendors informed us that they are not providing any fix since it is a responsibility of web developers to include the tracking library only in pages without sensitive contents.[10]

For what concerns the second class of vulnerabilities, 55 out of 90 websites have been found affected by the lack or misuse of the state parameter. More in detail, we identified 41 websites that do not support it, while the remaining 14 websites miss the security benefit of the state parameter by using a predictable or constant string as a value. We claim that such disheartening situation is mainly caused by the identity providers not setting this important parameter as mandatory. In fact, the state parameter is listed as recommended by Google and optional by VK. On the other hand, Facebook marks the state parameter as mandatory in its documentation, but our experiments showed that it fails to fulfill the requirement in practice. Additionally, it would be advisable to clearly point out in the OAuth 2.0 documentation of each provider the security implications of the parameter. For instance, according to the Google documentation,[11] the state parameter can be used "for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery": we believe that this description is too vague and opens the door to misunderstandings.

5.3 Compatibility Analysis

To detect whether WPSE negatively affects the web browser functionality, we performed a basic navigation session on the websites in our dataset. This interaction includes an access to their homepage, the identification of the SSO page, the execution of the OAuth 2.0 protocol, and a brief navigation of the private area of the website. In our experiments, the usage of WPSE did not impact in a perceivable way the browser performance or the time required to load webpages. We were able to navigate 81 websites flawlessly, but we also found 9 websites where we did not manage to successfully complete the protocol run.

In all the cases, the reason for the compatibility issues was the same, *i.e.*, the presence of an HTTP(S) request with a parameter called `code` after the execution of the protocol run. This message has the same syntactic

[7] `https://www.facebook.com/business/a/facebook-pixel`
[8] `https://www.google.com/adsense`
[9] `https://heapanalytics.com/`
[10] See, for instance, Google AdSense program policy available at `https://support.google.com/adsense/topic/6162392`
[11] `https://developers.google.com/identity/protocols/OAuth2WebServer`

structure as the last request sent as part of the authorization code mode of OAuth 2.0 and is detected as an attack when our security monitor moves back to its initial state at the end of the protocol run, because the message is indistinguishable from a session swapping attempt (see Section 2.2). We manually investigated all these cases: 2 of them were related to the use of the Gigya social login provider, which offers a unified access interface to many identity providers including Facebook and Google; the other 7, instead, were due to a second exchange of the authorization code at the end of the protocol run. We were able to solve the first issue by writing an XML specification for Gigya (limited to Facebook and Google), while the other cases openly deviate from the OAuth 2.0 specification, where the authorization code is only supposed to be sent to the redirect URI and delivered to the relying party from there. These custom practices are hard to explain and to support and, unsurprisingly, may introduce security flaws. In fact, one of the websites deviating from the OAuth 2.0 specification suffers from a serious security issue, because the authorization code is first communicated to the website over HTTP before being sent over HTTPS, thus becoming exposed to network attackers. We responsibly disclosed this security issue to the website owners.

In the end, all the compatibility issues we found boil down to the fact that a web protocol message has a relatively weak syntactic structure, which may end up matching a custom message used by websites as part of their functionality. We think that most of these issues can be robustly solved by using more explicit message formats for standardized web protocols like OAuth 2.0: explicitness is indeed a widely recognized prudent engineering practice for traditional security protocols [1]. Having structured message formats could be extremely helpful for a precise browser-side fortification of web protocols which minimizes compatibility issues.

6 Formal Guarantees

Now we formally characterize the security guarantees offered by our monitoring technique. Here we provide an intuitive description of the formal result, referring the interested reader to [15] for a complete account.

The formal result states that given a web protocol that is proven secure for a set of network participants and an uncorrupted client, by our monitoring approach we can achieve the same security guarantees given a corrupted client (*e.g.*, due to XSS attacks). More precisely this means that all attacks that will not occur in the presence of an ideally behaving client can be fixed by our monitor. Of course, these security guarantees only span the run of the protocol that is proven secure and its protocol-specific secrets. So the monitor can *e.g.*, ensure that the

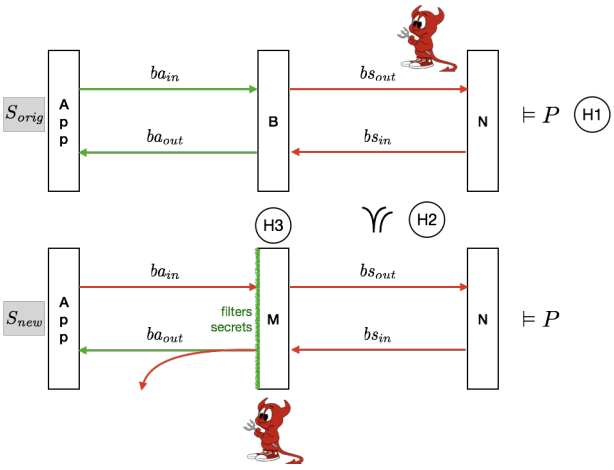

Figure 4: Visual description of Theorem 1

OAuth 2.0 protocol is securely executed in the presence of compromised scripts which might result in successful authentication and the setting of a session cookie. However, the monitor cannot prevent that this session cookie is leaked by a malicious script after the protocol run is over. So other security techniques (*e.g.*, the `HttpOnly` attribute for cookies) have to be in place or the protocol specification can in principle be extended to include the subsequent application steps (*e.g.*, we can protect session cookies like we do for access tokens).

Our theory is elaborated within the applied pi calculus [37], a popular process calculus for the formal analysis of cryptographic protocols, which is supported by various automated cryptographic protocol verifiers, such as ProVerif [10]. Bansal *et al.* [6] have recently presented a technique to leverage ProVerif for the analysis of web protocol specifications, including OAuth.

We give an overview on the theorem in Figure 4. We assume that the protocol specification has already been proven secure in a setting where the browser-side application is well-behaved and, in particular, follows the protocol specification (S_{orig}). Intuitively, our theorem says that security carries over to a setting (S_{new}) where the browser-side application is totally under the control of the attacker (*e.g.*, because of XSS attacks or a simple bug in the code) but the communication between the browser and the other protocol parties is mediated by our monitor.

Specifically, S_{orig} includes a browser B and an uncompromised application App, which exchange messages via private (green) communication channels ba_{in}, ba_{out}. The communication between the browser B and the network N is performed via the public (red) channels bs_{in}, bs_{out} that can be observed and infiltrated by the network attacker. S_{new} shows the setting in which the application is compromised: channel ba_{in} for requests from the application to the browser is made public, modeling that

arbitrary requests can be performed on it by the attacker. In addition, we assume the channel ba_{out} modeling the responses from the browser to the app to leak all messages and consequently modeling that the compromised application might leak these secrets. Indeed, the compromised application can communicate with the network attacker, which can in turn use the learned information to attack the protocol.

We state a simplified version of the correctness theorem as follows:

Theorem 1 (Monitor Correctness)**.** *Let processes App, N, B and M as defined in S_{orig} and P be a property on execution traces against a network attacker. Assume that the following conditions hold:*

(H1) $S_{orig} \vDash P$ *('S_{orig} satisfies P')*

(H2) $M \downarrow bs_{in}, bs_{out} \preccurlyeq S_{orig} \downarrow bs_{in}, bs_{out}$ *('the set of requests/responses on bs_{in},bs_{out} allowed by M are a subset of those produced by S_{orig}')*

*(H3) M does not leak any secrets (*i.e.*, messages initially unknown to the attacker) on ba_{out}*

Then it also holds that:

(C) $S_{new} \vDash P$ *('S_{new} satisfies P').*

Assumption (H1) states that the process as shown in S_{orig} satisfies a certain trace property. In the applied pi calculus, this is modeled by requiring that each partial execution trace of S_{orig} in parallel with an arbitrary network attacker satisfies the trace predicate P. Assumption (H2) states that the requests/responses allowed by the monitor M on the channels bs_{in}, bs_{out}, which model the communication between the browser and the network, are a subset of those possibly performed by the process S_{orig}. Intuitively, this means that the monitor allows for the intended protocol flow, filtering out messages deviating from it. Formally this is captured by projecting the execution traces of the corresponding processes to those components that model the input and output behavior on bs_{in} and bs_{out} and by requiring that for every such execution trace of M there is a corresponding one for S_{orig}. Finally, assumption (H3) states that the monitor M should not leak any secrets with its outputs on channel ba_{out}. In applied pi calculus this is captured by requiring that the outputs of M on channel ba_{out} do not to contain any information that increases the attacker knowledge.

Together these assumptions ensure that the monitored browser behaves as the ideal protocol participant in S_{orig} towards the network and additionally assure that an attacker cannot gain any additional knowledge via a compromised application that could enable her to perform attacks against the protocol over the network. Formally, this is captured in conclusion (C) that requires the partial execution traces of S_{new} to satisfy the trace predicate P.

6.1 Discussion

Our formal result is interesting for various reasons. First, it allows us to establish formal security guarantees in a stronger attacker model by checking certain semantic conditions on the monitor, without having to prove from scratch the security of the protocol with the monitor in place on the browser-side. Second, the theorem demonstrates that enforcing the three security properties identified in Section 2 does indeed suffice to protect web protocols from a large class of bugs and vulnerabilities on the browser side: (H2) captures the compliance with the intended protocol flow as well as data integrity, while (H3) characterizes the secrecy of messages.

Finally, the three hypotheses of the theorem are usually extremely easy to check. For instance, let us consider the OAuth protocol. As previously mentioned, this has been formally analyzed in [6], so (H1) holds true. In particular, the intended protocol flow is directly derivable from the applied pi calculus specification. The automaton in Figure 2 only allows for the intended protocol flow, which is clearly contained in the execution traces analyzed in [6]. Hence (H2) holds true as well. Finally, the only secrets in the protocol specification are those subject to the confidentiality policy in the automaton in Figure 2: as previously mentioned, these are replaced by placeholders, which are then passed to the web application. Hence no secret can ever leak, which validates (H3).

7 Related Work

7.1 Analysis of Web Protocols

The first paper to highlight the differences between web protocols and traditional cryptographic protocols is due to Gross *et al.* [22]. The paper presented a model of web browsers, based on a formalism reminiscent of input/output automata, and applied it to the analysis of password-based authentication, a key ingredient of most browser-based protocols. The model was later used to formally assess the security of the WSFPI protocol [23].

Traditional protocol verification tools have been successfully applied to find attacks in protocol specifications. For instance, Armando *et al.* analyzed both the SAML protocol and a variant of the protocol implemented by Google using the SATMC model-checker [4]. Their analysis exposed an attack against the authentication goals of the Google implementation. Follow-up work by the same group used a more accurate model to find an authentication flaw also in the original SAML

specification [3]. Akhawe *et al.* used the Alloy framework to develop a core model of the web infrastructure, geared towards attack finding [2]. The paper studied the security of the WebAuth authentication protocol among other case studies, finding a login CSRF attack against it. The WebSpi library for ProVerif by Bansal *et al.* has been successfully applied to find attacks against existing web protocols, including OAuth 2.0 [6] and cloud storage protocols [5]. Fett *et al.* developed the most comprehensive model of the web infrastructure available to date and fruitfully applied it to the analysis of a number of web protocols, including BrowserID [17], SPRESSO [18] and OAuth 2.0 [19].

Protocol analysis techniques are useful to verify the security of protocols, but they assume websites are correctly implemented and do not depart from the specification, hence many security researchers performed empirical security assessments of existing web protocol implementations, finding dangerous attacks in the wild. Protocols which deserved attention by the research community include SAML [41], OAuth 2.0 [43, 27] and OpenID Connect [28]. Automated tools for finding vulnerabilities in web protocol implementations have also been proposed by security researchers [46, 50, 48, 31]. None of these works, however, presented a technique to protect users accessing vulnerable websites in their browsers.

7.2 Security Automata

The use of finite state automata for security enforcement is certainly not new. The pioneering work in the area is due to Schneider [40], which first introduced a formalization of security automata and studied their expressive power in terms of a class of enforceable policies. Security automata can only stop a program execution when a policy violation is detected; later work by Ligatti *et al.* extended the class of security automata to also include edit automata, which can suppress and insert individual program actions [29]. Edit automata have been applied to the web security setting by Yu *et al.*, who used them to express security policies for JavaScript code [49]. The focus of their paper, however, is not on web protocols and is only limited to JavaScript, because input/output operations which are not JavaScript-initiated are not exposed to their security monitor.

Guha *et al.* also used finite state automata to encode web security policies [24]. Their approach is based on three steps: first, they apply a static analysis for JavaScript to construct the control flow graph of an Ajax application to protect and then they use it to synthesize a request graph, which summarizes the expected input/output behavior of the application. Finally, they use the request graph to instruct a server-side proxy, which performs a dynamic monitoring of browser requests to prevent observable violations to the expected control flow. The security enforcement can thus be seen as the computation of a finite state automaton built from the request graph. Their technique, however, is only limited to Ajax applications and operates at the server side, rather than at the browser side.

7.3 Browser-Side Defenses

The present paper positions itself in the popular research line of extending web browsers with stronger security policies. To the best of our knowledge, this is the first work which explicitly focuses on web protocols, but a number of other proposals on browser-side security are worth mentioning. Enforcing information flow policies in web browsers is a hot topic nowadays and a few fairly sophisticated proposals have been published as of now [21, 26, 8, 36, 7]. Information flow control can be used to provide confidentiality and integrity guarantees for browser-controlled data, but it cannot be directly used to detect deviations from expected web protocol executions, which instead are naturally captured by security automata. Combining our approach with browser-based information flow control can improve its practicality, because a more precise information flow tracking would certainly help a more permissive security enforcement.

A number of browser changes and extensions have been proposed to improve web session security, both from the industry and the academia. Widely deployed industrial proposals include Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS). Notable proposals from the academia include Allowed Referrer Lists [16], SessionShield [32], Zan [44], CSFire [38], Serene [39], CookiExt [11], SessInt [12] and Michrome [13]. Moreover, JavaScript security policies are a very popular research line in their own right: we refer to the survey by Bielova [9] for a good overview of existing techniques. None of these works, however, tackles web protocols.

8 Conclusion

We presented WPSE, the first browser-side security monitor designed to address the security challenges of web protocols, and we showed that the security policies enforceable by WPSE suffice to prevent a large number of real-world attacks. Our work encompasses a thorough review of well-known attacks reported in the literature and an extensive experimental analysis performed in the wild, which exposed several undocumented security vulnerabilities fixable by WPSE in existing OAuth 2.0 implementations. We also discovered a new attack on the Google implementation of SAML 2.0 by formalizing its specification in WPSE. In terms of compatibility, we

showed that WPSE works flawlessly on many existing websites, with the few compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability. In the end, we conclude that the browser-side security monitoring of web protocols is both useful for security and feasible in practice.

As to future work, we observe that our current assessment of WPSE in the wild only covers two specific classes of vulnerabilities, which can be discovered just by navigating the tested websites: extending the analysis to cover active attacks (in an ethical manner) is an interesting direction to get a better picture of the current state of the OAuth 2.0 deployment. We would also like to improve the usability of WPSE by implementing a more graceful error handling procedure: *e.g.*, when an error occurs, we could give users the possibility to proceed just as it routinely happens with invalid HTTPS certificates. Using more descriptive warning messages may also be useful for web developers that are visiting their websites with WPSE so that they can understand the issue and provide the appropriate fixes to the server side code. Finally, we plan to identify automated techniques to synthesize protocol specifications for WPSE starting from observable browser behaviours in order to make it easier to adopt our security monitor in an industrial setting.

Acknowledgments. This work has been partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC), by Netidee through the project EtherTrust (grant agreement 2158), by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694) and COMET K1 SBA. The paper also acknowledges support from the MIUR project ADAPT and by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks funded by CISCO Systems Inc. and Leonardo SpA.

References

[1] M. Abadi and R. M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 290–304, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An Authentication Flaw in Browser-Based Single Sign-On protocols: Impact and Remediations. *Computers & Security*, 33:41–58, 2013.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-Based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, pages 1–10, 2008.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Proceedings of the 2nd International Conference on Principles of Security and Trust (POST 2013)*, pages 126–146, 2013.

[6] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[7] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[8] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST 2014)*, pages 159–178, 2014.

[9] N. Bielova. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming*, 82(8):243–262, 2013.

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

[11] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the Browser against Session Hijacking Attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[12] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably Sound Browser-Based

Enforcement of Web Session Integrity. In *Proceedings of the IEEE 27th Computer Security Foundations Symposium (CSF 2014)*, pages 366–380, 2014.

[13] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei. Micro-policies for web session security. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016)*, pages 179–193, 2016.

[14] S. Calzavara, R. Focardi, M. Maffei, C. Schneidewind, M. Squarcina, and M. Tempesta. Login-CSRF on Google due to SAML2.0 flaws. `https://secgroup.dais.unive.it/login-csrf-google-saml2-flaws/`.

[15] S. Calzavara, R. Focardi, M. Maffei, C. Schneidewind, M. Squarcina, and M. Tempesta. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring - Technical report. `https://sites.google.com/site/wpseproject/`.

[16] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight Server Support for Browser-Based CSRF Protection. In *Proceedings of the 22nd International World Wide Web Conference (WWW 2013)*, pages 273–284, 2013.

[17] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P 2014)*, pages 673–688, 2014.

[18] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, pages 1358–1369, 2015.

[19] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*, pages 1204–1215, 2016.

[20] Google. GSuite Administrator Help, Set up SSO via a third party Identity provider. `https://support.google.com/a/answer/6262987`, 2018.

[21] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*, pages 748–759, 2012.

[22] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser Model for Security Analysis of Browser-Based Protocols. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, pages 489–508, 2005.

[23] T. Groß, B. Pfitzmann, and A. Sadeghi. Proving a WS-Federation Passive Requestor Profile with a Browser Model. In *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005, Fairfax, VA, USA, November 11, 2005*, pages 54–64, 2005.

[24] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International Conference on World Wide Web (WWW 2009)*, pages 561–570, 2009.

[25] D. Hardt. The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`, 2012.

[26] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow Security for JavaScript and its APIs. *Journal of Computer Security*, 24(2):181–234, 2016.

[27] W. Li and C. J. Mitchell. Security Issues in OAuth 2.0 SSO Implementations. In *Proceedings of the 17th International Conference in Information Security (ISC 2014)*, pages 529–541, 2014.

[28] W. Li and C. J. Mitchell. Analysing the Security of Google's Implementation of OpenID Connect. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2016)*, pages 357–376, 2016.

[29] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[30] K. Ludwig. Duo Finds SAML Vulnerabilities Affecting Multiple Implementations. `https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations`, 2018.

[31] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single Sign-On Security–An Evaluation of OpenID Connect. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 251–266, 2017.

[32] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS 2011)*, pages 87–100, 2011.

[33] OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`, 2005.

[34] OASIS. Security Assertion Markup Language (SAML) v2.0. `https://www.oasis-open.org/standards#samlv2.0`, 2005.

[35] OASIS. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://www.oasis-open.org/committees/download.php/56779/sstc-saml-bindings-errata-2.0-wd-06.pdf`, 2015.

[36] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF 2015)*, pages 366–379, 2015.

[37] M. D. Ryan and B. Smyth. Applied Pi Calculus. In *Formal Models and Techniques for Analyzing Security Protocols*, chapter 6. IOS Press, 2011.

[38] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and Precise Client-Side Protection against CSRF Attacks. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS 2011)*, pages 100–116, 2011.

[39] P. D. Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: Self-Reliant Client-Side Protection against Session Fixation. In *Proceedings of the 2012 Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012*, pages 59–72, 2012.

[40] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[41] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *Proceedings of the 21th USENIX Security Symposium*, pages 397–412, 2012.

[42] B. Stock and M. Johns. Protecting users against XSS-based password manager abuse. In *Proceedings of the 9th ACM Asia Conference on Information, Computer and Communications Security (AsiaCCS 2014)*, pages 183–194, 2014.

[43] S. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, (CCS'12)*, pages 378–390, 2012.

[44] S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 615–626, 2011.

[45] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Measuring Login Webpage Security. In *Proceedings of 32nd ACM Symposium on Applied Computing (SAC 2017)*, pages 1753–1760, 2017.

[46] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P 2012)*, pages 365–379, 2012.

[47] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium*, pages 399–314, 2013.

[48] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016)*, pages 651–662, 2016.

[49] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL 2007)*, pages 237–249, 2007.

[50] Y. Zhou and D. Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium*, pages 495–510, 2014.

A Sample XML Specification

Figure 5 shows the XML specification of the OAuth 2.0 automaton in Figure 2. The protocol is enclosed within `<Protocol>` tags and describes the flow as a sequence of requests and responses. For every message we detail its pattern, possibly specifying the endpoint and a list of parameters for requests or a list of headers for responses.

Identifiers can be introduced in the protocol flow specification by adding the `id` attribute to the tag of the message component of interest. Additional identifiers can be defined within `<Definition>` tags, where the value that is associated to the new identifier is the part of the `<Source>` matching the regular expression `<Regexp>`. If the regular expression contains a capturing group, denoted by parenthesis, only the string matching the group is selected. The syntax `${id}` can be used to refer to the value bound to the identifier `id`.

Security policies are defined within `<Secrecy>` and `<Integrity>` tags. The secrecy policy specifies that the value in `<Target>` must be sent only to the enumerated origins. The integrity policy specifies that the value in `<Target>` must match the content of `<Matches>`, which can possibly be a regular expression.

```
<Specification name="google-explicit-nostate">
    <Protocol>
        <Request method="GET" desc="req_init">
            <Endpoint>
                <Regexp> https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth </Regexp>
            </Endpoint>
            <Parameter name="response_type"> code </Parameter>
            <Parameter name="redirect_uri" id="req_init_redirect_uri" />
        </Request>
        <Response desc="resp_init">
            <Endpoint>
                <Regexp> https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth </Regexp>
            </Endpoint>
            <Header name="Location" id="resp_init_location" />
        </Response>
        <Request method="GET" desc="req_code">
            <Endpoint id="uri2"/>
            <Parameter name="code">
                <Regexp> [^\s]{40,} </Regexp>
            </Parameter>
        </Request>
    </Protocol>
    <Identifiers>
        <Definition id="uri1">
            <Source> ${req_init_redirect_uri} </Source>
            <Regexp> ^(https?://.*?)(?:\?|$) </Regexp>
        </Definition>
        <Definition id="origin">
            <Source> ${req_init_redirect_uri} </Source>
            <Regexp> ^(https?://.*?/).* </Regexp>
        </Definition>
        <Definition id="authcode">
            <Source> ${resp_init_location} </Source>
            <Regexp> [?&amp;]code=(.*?)(?:&amp;|$) </Regexp>
        </Definition>
    </Identifiers>
    <Policy>
        <Secrecy> <!-- the auth code contained in the Location header must be kept secret -->
            <Target> ${authcode} </Target>
            <Origin> ${origin} </Origin>
            <Origin> https://accounts.google.com/ </Origin>
        </Secrecy>
        <Integrity> <!-- the last message must be sent to the redirect URI initially specified -->
            <Target> ${uri2} </Target>
            <Matches> ${uri1} </Matches>
        </Integrity>
    </Policy>
</Specification>
```

Figure 5: XML specification for the automaton in Figure 2.