Master's Degree
in Computer Science

Final Thesis

Run-time Prevention of Logic Flaws in Multi-Party Web Applications

Supervisor
Prof. Stefano Calzavara

Graduand
Lorenzo Veronese
Matricolation number 852058

Academic Year
2019/2020

ABSTRACT

Modern web applications often rely on third-party services to provide their functionality to users. The integration of these services is a non-trivial task and, as shown by the large number of attacks against Single-Sign-On and Cashier-as-a-Service protocols, often opens up possibilities for logic flaws in web security protocols. In this thesis we explore the design challenges of a run-time security monitor for web protocols, identifying the fundamental ingredients needed to mitigate logic flaws in multi-party web applications. We then present a black-box methodology to generate verified monitors from applied pi-calculus specifications of web protocols. These monitors are guaranteed to have the security properties defined in the specification phase and can be deployed on the browser-side (ServiceWorker) and the server-side (reverse proxy). We evaluate the effectiveness of the approach by testing it against a pool of vulnerable applications that use the OAuth 2.0 protocol and that integrate the PayPal payment system.

CONTENTS

1 INTRODUCTION

Modern web application often rely on third-party services to provide their functionality to users. These services often implement security protocols on top of HTTPS and provide APIs for their integration. For example, many applications authenticate users using the single-sign-on (SSO) protocols offered by Facebook, Google, Twitter, and many web stores use payment APIs provided by payment gateway services such as PayPal. This trend of integrating more and more of such services turned web applications into multi-party systems.

The integration, implementation and deployment of web-based security protocols is a non-trivial task and, as shown by the large number of vulnerabilities that have been publicly disclosed, careless design or engineering opens up possibilities for logic flaws, in both specifications and implementations [31, 30, 29, 18, 5, 25, 28].

Many efforts have been made to automatically test web applications for logical flaws [28, 25]. The main approach is to extract an informal model of the application from its code or network traces, then to infer logical properties and trying to break them by applying a set of attack patterns. This however does not prevent attacks to be executed on already deployed implementations, so many have been studying how to effectively mitigate these attacks by run-time monitoring [19, 13, 21, 35, 14, 22]. These works highlight how DFAs and data invariants are an effective way to generate security monitors. These monitors offer online protection to both the browser [13, 21] and the integrator side [35, 14, 22], but none of them motivate the positioning choice in a satisfactory way. Pellegrino and Balzarotti [25] focus on the client since the communication between multiple parties is usually channeled through the browser; Xinget al. [35], on the other hand, focus on the integrator since prior research show that the weakest link remains the integrator-side whose code seem way more error prone than that of the provider. Each approach shows limitations in its ability to prevent attacks in channels which it is not able to observe. Xing et al. [35] stress that their Integuard monitor does not offer protection on the provider-side, expecting further efforts to be made in that direction. To the best of our knowledge, no one has yet proposed a systematic study of protocol observability in multy-party web integration with the objective of evaluating the effectiveness of security monitors.

Most of the black-box approaches lack a formal guarantee of the effectiveness of their monitors, opening up opportunities for false negatives. Li and Xue [22] explicitly mention formal verification of likely invariants as a future research

direction; Guha et al. [19] cite soudness as one of the main limitations of their work. WPSE [13] is one of the few works that try to give some formal guarantees by proving that their monitor, coupled with a compromised client, offers the same security guarantees as an uncorrupted client. Our work build on this premise by automatically proving that the same security properties that hold in the protocol specification are maintained in case of a corrupted party in a monitored system.

There have been many efforts to formally verify both the specifications [18, 5] and implementations [4, 31, 38] of web protocols. The WebSpi library by Bansal et al. [5] has been successfully applied to find attacks against existing web protocols, and it offers a comprehensive set of components needed to model web applications.

In order to deal with legacy implementations of security protocols Pironti and Jürjens [26] present an approach to automatically generate security monitors from the specification of the protocol actors obtained by their observable behavior. Our work builds on this idea by applying it to the generation of security monitors for web protocols. We focus on the provider side by giving it a method to generate generic monitors that can be distributed together with web protocol SDKs. The purpose of the SDKs is, in fact, to enable developers to produce apps that use authentication/authorization in a way that provides the desired security properties. Multiple works [26, 28], however, show that these SDKs contain bugs or implicit assumptions that may lead to logical vulnerabilities.

1.1 CONTRIBUTIONS

This thesis aims to identify the fundamental ingredients needed to mitigate logic flaws in multi-party web application by analyzing the observability of web protocols by the different parties. With this knowledge we show that different attacks needs specific entities to take action to be prevented and that security monitors for single parties might not be sufficient to cover the whole range of possible attacks. Moreover, we show that these attacks can be mitigated by using monitors deployed by the server.

We propose a black-box methodology to generate formally-verified monitors from applied-pi calculus specifications of web security protocols. These monitors are guaranteed to have the security properties defined in the specification phase and can be deployed on the browser-side (`ServiceWorker`) and the server-side (reverse proxy). To the best of our knowledge this is the first work that proposes to use the `ServiceWorker` browser API as a security mechanism.

We implemented a prototype of our approach based on the ProVerif[8] protocol verifier that is able to generate generic monitors as `JavaScript` files or python plugins for the mitmproxy[15] HTTP reverse proxy.

We extended the WebSpi[5] library to support the `ServiceWorker` API, the `referrer-policy` header and url fragments.

1.2 STRUCTURE OF THE THESIS

In chapter 2 we introduce the basic concepts and tools that will be used through the rest of the document. In particular, we introduce the OAuth 2.0 and PayPal web protocols, we give a tour of the features of the ProVerif protocol verifier and the WebSpi library. In section 3 we briefly survey the related work. In section 4 we introduce the different approaches to run-time monitoring of multi-party web applications by discussing their strengths and weaknesses. We then show how they can be used to mitigate the various attacks that can be found in the literature and discuss our idea. In section 5 we present our approach and the internals of our prototype, showing how ProVerif is used during the generation of formally-verified monitor implementations. In section 6 we discuss two case studies and show how security monitors for OAuth 2.0 and PayPal can be automatically generated. We then discuss the experimental evaluation of the generated monitors. We conclude in section 7.

2

BACKGROUND

2.1 MULTI-PARTY WEB APPLICATIONS

A *multi-party web application* (MPWA) [28] is a web application that integrates services from other trusted web APIs. These web API can provide *single-sign-on* (SSO) or *cashier-as-a-service* (CaaS) web security protocols. The entities that at minimum need to be involved in the execution of these protocols are three: The user, that through a browser (the User Agent or UA) interacts with a service provider or relaying party (SP), that integrates the API of a trusted third party (TTP) that might play the role of identity provider (IdP).

2.1.1 *OAuth 2.0*

OAuth 2.0 [20] is an authorization protocol that enables third parties to obtain limited access to a resources in behalf of the resource owner. Typically it is used as a SSO mechanism to authenticate the resource owner to third parties by giving them permission to access his identity information at the identity provider.

The OAuth 2.0 specification [20] defines four roles:

RESOURCE OWNER
: The entity capable of granting access to a resource. This refers to the end-user (U) which accesses the web through a user agent (UA).

RESOURCE SERVER
: The server hosting the protected resources. It can accept requests to the resources that use valid access tokens.

CLIENT
: The web application that makes request to the protected resources on behalf of the resource owner. This website will later be called service provider (SP).

AUTHORIZATION SERVER
: The server that issues access tokens after successfully authenticating the resource owner and obtaining authorization. The authorization server might be the same as the resource server or a separate entity. We will refer to this entity as trusted third party (TTP) or identity provider (IdP).

Depending on the intermediate credential used to represent the resource owner authorization, OAuth defines four *grant types*: 1. authentication code mode

2. implicit mode 3. resource owner credentials 4. client credentials. We will focus on the authorization code mode and implicit mode as they are the more widely used and do not require explicit user credentials.

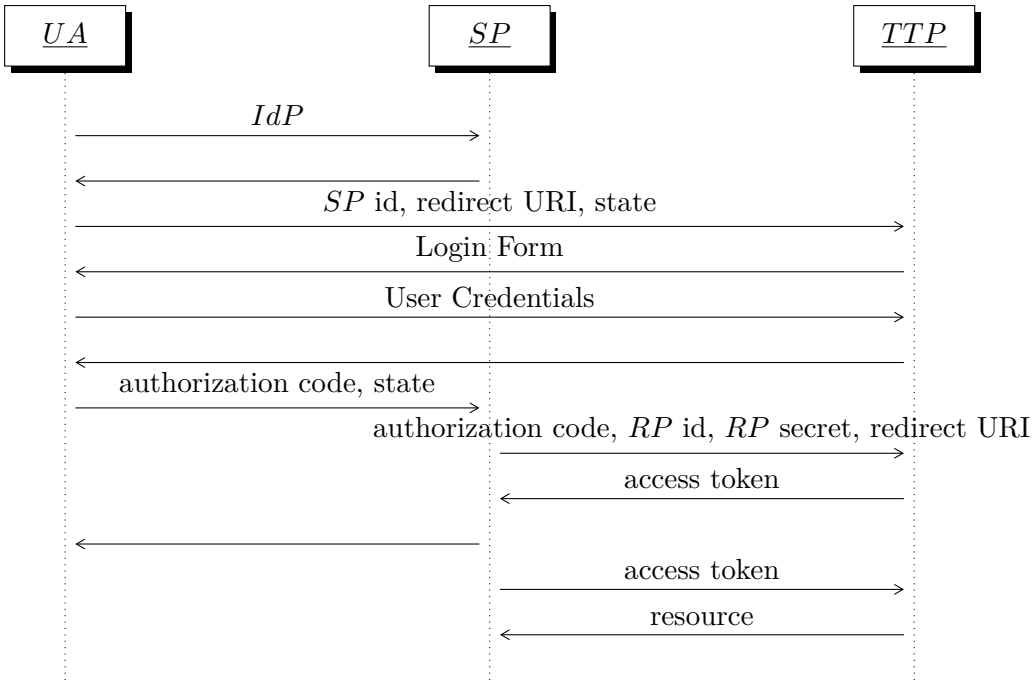


Figure 1: OAuth 2.0 Authorization Code Mode

Figure 1 shows an high-level view of the interaction between the different entities, using the authorization code mode:

1. The user starts the protocol by visiting the service provider website and specifying that she wants to log in using IdP as identity provider.
2. The SP redirects the user back to TTP specifying its identifier at IdP, the URL to which the user is redirected when the access is granted and an optional **state** parameter for CSRF protection.
3. The user authenticates with TTP and gives SP permissions to access her resources.
4. TTP redirects the user back at SP's redirection URI that was specified in the second message.The redirection URI includes an **authorization code** and the state parameter provided earlier by the service provider.
5. The authorization code is exchanged with an access **token** at the token endpoint of the TTP. Here the SP authenticates with TTP by providing its client secret and the redirect URI that was used to redirect the user.
6. The authentication token can then be used by the SP to request user resources.

The implicit mode differs from the authorization code mode in steps 4 and 5, where the access token is issued to the client directly instead of issuing an authorization code. The access token is provided in the URI fragment of the redirection URI, to be read by the client application. The implicit mode is optimized for clients implemented in a browser using JavaScript, it improves the responsiveness and efficiency of some clients since it reduces the number of round trips requires, but at the expense of the security of the access token. When issuing the token the authorization server does not authenticate the *SP* (but it might use the redirect URI for this purpose) and the access token can be exposed to the resource owner or other applications with access to the user agent.

2.1.2 *PayPal*

PayPal is a cashier-as-a-service (CaaS) API that enables merchants to use a third party service as a payment method. This enables users to make purchases on the merchant website by only trusting the third party (PayPal) with her data. This service is often called payment gateway.

PayPal offers different APIs for accessing its services. The main ones are the PayPal Standard Checkout and the PayPal Express checkout. Over the years these services changed name multiple times, but their main difference resides in how the integrator website is notified about the status of the transaction. The PayPal Standard checkout process make use of the *Instant Payment Notification* (IPN) message service. After the checkout is completed on the PayPal website, a message with the status of the transaction is sent to the IPN endpoint of the integrator website. This message might not be sent immediately and the integrator website needs to explicitly acknowledge it for it not to be re-sent. The PayPal Express Checkout avoids using IPN messages and instead uses a token, that is issued by the PayPal service after the payment confirmation by the user to complete the transaction. This behavior is related to the optional *Payment Data Transfer* (PDT) notification service for PayPal Standard. Using PDT a merchant can receive immediate notification of a transaction when the user is redirected back to the merchant website after completing the checkout at PayPal. Note that with PayPal Standard it is possible to enable both the PDT and IPN message services, so that is possible to receive order confirmation notifications immediately and still receive other types of notifications with IPN. Indeed, PDT notification major weaknesses are that the only notification that is capable of receiving is order confirmations (this do not include, for example chargeback notifications) and that notifications are sent only once and do not need to be explicitly acknowledged.

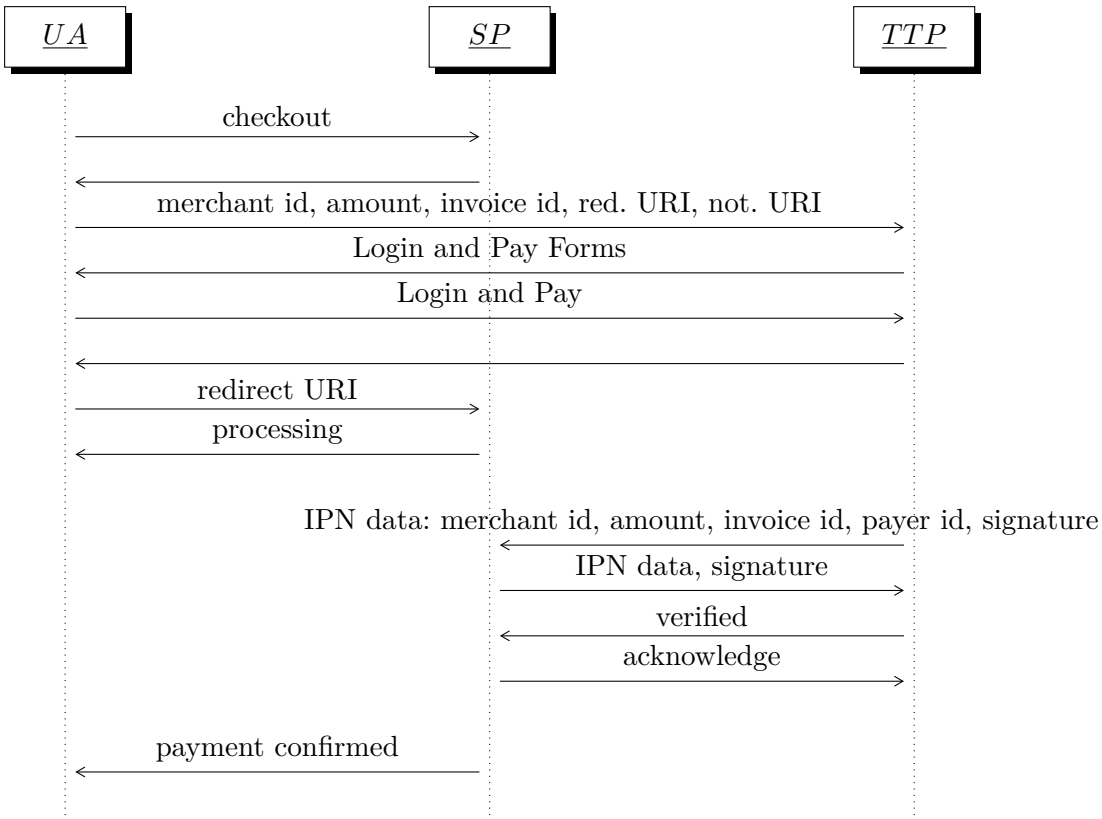


Figure 2: PayPal Standard IPN Flow

Figure 2 shows an high-level view of the protocol flow of PayPal Standard with IPN notifications:

1. The user starts the checkout process and is redirected to the PayPal website to complete the transaction. Inside the redirection data, the merchant website sends the TTP its identifier, the amount that needs to be payed, its invoice id and two URLs, one for redirecting the user after the checkout and one to receive notifications. These URLs needs to be registered on the PayPal website before they are used.
2. The user optionally logs into PayPal and makes the payment.
3. The user is redirected by the TTP to the redirection URL of the SP. At this point the SP has not yet received the payment notification, so sets the status of the invoice to *"processing"*.
4. The TTP sends the IPN notification of the successful payment to the SP IPN endpoint. The notification contains the status of the payment, its invoice id, and more importantly, the merchant id of the merchant that received the payment and the amount that was actually payed. The

message contains a signature or MAC issued by the TTP that guarantees the message authenticity.

5. The SP verifies the IPN message by sending it back to the TTP. If the signature or MAC is valid, the SP receives a positive response and can set the status of the invoice to *"payed"*. It can later notify the user that the payment was successful.

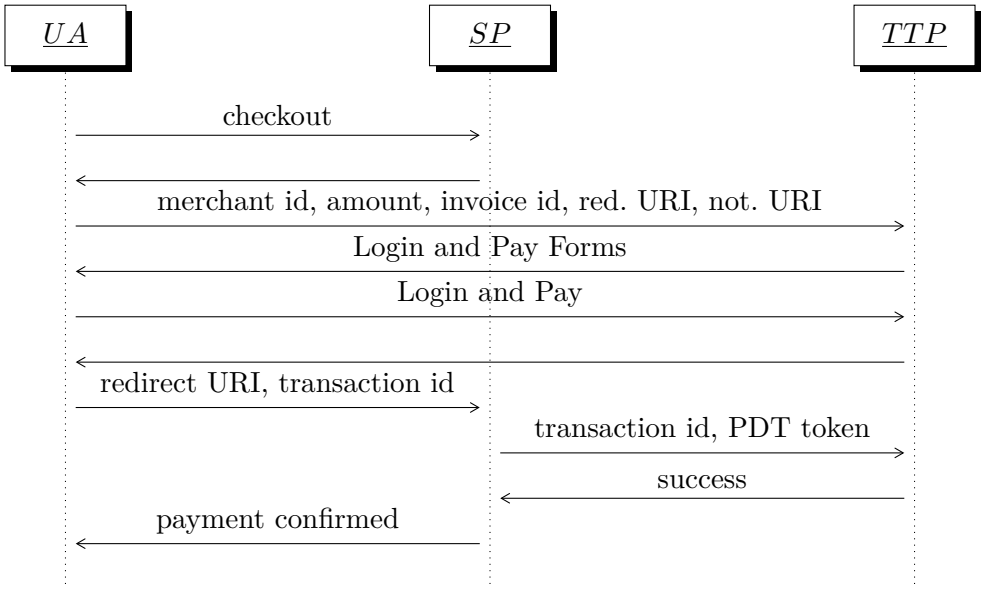


Figure 3: PayPal Standard PDT Flow

Figure 3 shows the main difference between the IPN and PDT protocol flows: here, after the checkout is confirmed by the user on TTP, she is redirected back to the SP with a transaction id and some other information regarding the current checkout process. The SP can verify this information by sending the transaction id to the TTP. Once the transaction is confirmed and verified the SP confirms the payment on the current invoice, without requiring the TTP to send an IPN notification.

2.2 LOGIC FLAWS IN MULTI-PARTY WEB APPLICATIONS

The term logic flaw or logic vulnerability refers to the class of vulnerabilities that, as opposed to the technical vulnerabilities, do not arise from unsafe coding practices or mistakes in input validation. Usually logic vulnerabilities are ways of using the legitimate processing flow of the application in a way that results in negative effects for the application itself.

Logic flaws still lack a unique and formal definition, but in the literature we agree that these vulnerabilities are the consequence of improper validation of the

business process of the application. This can refer to both the navigation between different pages and the data flow that links together these pages. In the first case, the application does not enforce a particular sequence of actions to perform a specific operation. The second case refers to the possibility of tampering with the values that need to be propagated between the different HTTP requests.

When applied to web protocols these criteria can be rephrased in terms of the three security challenges that Calzavara et al. [13] identify in the implementation of web protocols:

Definition 2.2.1. A logic flaw in a multi-party web application is a vulnerability that results from the violation of one of the following properties: 1. intended protocol flow 2. integrity of messages 3. secrecy of messages

The first property derives from the fact that protocols are usually specified as a sequence of message exchanges. Each party implementation needs to be robust with respect to the reception of out-of-order messages and terminate the protocol in case of violation or missing messages. All CSRF related vulnerability can be classified as protocol flow violations. Let's consider, for example, the social login CSRF [5]. Here a user is tricked on visiting a page that corresponds to the callback message of the OAuth 2.0 protocol (redirect to SP from TTP with auth. code in Figure 1). This redirection, on vulnerable clients, makes the victim log in as the attacker by finishing the protocol run starting from this mid-point in the protocol flow. A correct protocol flow enforcement would block this attack since the protocol run of the victim does not start from the first message.

Each party needs to ensure the integrity of the messages it receives: the implementation needs to detect tampering of the values that are sent as part of the protocol. An example of improper validation of messages is the shop-for-free vulnerability of a vulnerable *osCommerce* installation [25]. Here a malicious user changes the merchant identifier of the store before being redirected to the TTP (first redirection in both Figure 2 and 3). Changing the merchant identifier enables the attacker to replace it with its own account identifier to be able to pay himself instead of paying the merchant. Enforcing that the merchant identifier sent by the SP and later received in the IPN or PDT messages match the real merchant can effectively block this attack.

Usually, web-protocols rely on the confidentiality of credentials, tokens and nonces. The implementations need to enforce this secrecy and prevent the leakage of confidential information. Let's take for example the state leakage attack on OAuth 2.0 [17]. An attacker that can leak the state parameter of an honest user can bypass the CSRF protection enabled by this parameter. This attack can be prevented by maintaining the parameter confidential and unguessable.

2.3 FORMAL VERIFICATION OF WEB PROTOCOLS

Logic vulnerabilities are very difficult for automated scanners to detect. To be able to reason about the properties of protocol flow, integrity and secrecy, we need an appropriate model of the protocol.

To this end we describe web-protocols protocols using the applied-pi calculus [3] and verify their security properties using the ProVerif protocol verifier and the WebSpi [5] library. Each time a new protocol is modeled, its security properties are modeled using correspondence assertions and secrecy assertions. These properties can be classified in the three categories of Definition 2.2.1, but are usually more specific to the protocol that is being modeled. For example, an OAuth 2.0 auth. code needs to be issued before the access token for the same session, and an IPN notification sent from the PayPal server needs to correspond to a completed transaction. We will give more detail about these properties in chapter 5 and show some examples in chapter 6.

2.3.1 *ProVerif*

ProVerif [8] is a protocol verifier written by Bruno Blanchet. Its specification language is a variant of the applied-pi calculus [3], that is able to model communicating concurrent processes and data structures. The communication channels that are used by these processes are public, and assumed to be controlled by an attacker with "Dolev-Yao" capabilities [16]. Since the attacker has complete control of the channels, she can modify, inject and delete messages. Moreover, she is able to manipulate data and decrypt messages if she has the necessary key. Cryptography is assumed to be perfect, so the attacker is restricted to use the cryptographic primitives defined by the user. Proverif allows for the security properties of each protocol to be encoded in a formal way so that it is able to verify that the claimed properties hold.

PROCESSES AND MESSAGES

Table 1 show a summary of the ProVerif syntax for what concerns processes and messages. Terms are typed atomic names or variables that can be composed by pairing (n-tuples) or with data constructors. The types that can be used are `channel`s, `bitstring`s or user defined types. Processes can read or write terms to channels and store or retrieve them from tables (which are internally encoded as private channels). Fresh variables and nonces are created using the `new` keyword. `let` can introduce new bindings and decompose messages using pattern matching, that can also be used when receiving from channels or reading from tables. Processes can be run in parallel or replicated indefinitely.

$P, Q ::=$	**processes**
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\mathbf{new}(n : t); P$	name restriction
$\mathbf{in}(M, T); P$	message input
$\mathbf{out}(M, N); P$	message output
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$	term evaluation
$\mathbf{insert}\ d(M_1, ..., M_n); P$	insert record in table
$\mathbf{get}\ d(T_1, ..., T_n)\ \mathbf{in}\ P\ \mathbf{else}\ Q$	read record from table
$\mathbf{event}\ e(M_1, ..., M_n); P$	execute event
$R(M_1, ..., M_k)$	macro usage
$M, N ::=$	**terms**
a, b, c	names
x, y, z	variables
$(M_1, ..., M_k)$	tuple
$h(M_1, ..., M_k)$	constructor/destructor application
$M = N$	term equality
$M <> N$	term inequality
$M \&\& N$	conjuntion
$M \mid\mid N$	disjunction
$\mathbf{not}(M)$	negation
$T ::=$	**patterns**
$x : t$	typed variable
x	variable withut explicit type
$(T_1, ..., T_n)$	tuple
$= M$	equality test

Table 1: ProVerif Processes Syntax

CRYPTOGRAPHY: CONSTRUCTORS AND DESTRUCTORS

ProVerif uses constructor (function symbols) to model symbolic cryptography. All cryptographic operations are perfect black-boxes defined only in terms of constructor and destructors. To define symmetric encryption, for example, we would write

fun $senc(\mathsf{bitstring}, key) : \mathsf{bitstring}.$

reduc forall $b : \mathsf{bitstring}, k : key; sdec(senc(b, k), k) = b.$

where *key* is a user defined type. The encryption of a bitstring b with key k is given by $senc(b, k)$. The destructor *sdec* can only be applied to decrypt an encrypted message only if we know the correct key k. Moreover, a constructor is only reversable if a corresponding destructor is defined: hash functions are simply defined as constructor without destrcutor.

$$\textbf{fun } hash(\mathsf{bitstring}) : \mathsf{bitstring}.$$

Constructors and destructors are available to the attacker unless they are declared private (using the `[private]` keyword). Private constructors are useful to model secret algorithms that the attacker is not able to use or for key distribution.
A constructor may be declared as *data constructor* by using the keyword `[data]` in its definition. A data constructor can be constructed and decomposed by the attacker: it implicitly declares destructors for all of its fields and a pattern for destructuring it with pattern matching.

QUERIES AND VERIFICATION

ProVerif is able to prove reachability and secrecy properties, correspondence assertions and observational equivalence. We first annotate the process with *events* which mark important stages reached by the protocol but that do not affect its behavior. The process

$$\textbf{event } e(M_1, ..., M_n); P$$

defines an event e with n parameters and otherwise is equivalent to the execution of the process P. The parameters enable us to study relationship between the arguments of different events.
Proving reachability properties is the most basic capability of the tool, and with them it is possible to investigate which terms are available to the attacker, hence to evaluate (syntactic) secrecy of terms.

$$\textbf{query } attacker(M)$$

Where M is a ground term.
A more complete example of ProVerif query is

$$\textbf{query } x_1 : t_1, ..., x_n : t_n;\ \textbf{event}(e(M_1, ..., M_j)) \Longrightarrow \phi$$

where M_i are terms built by the application of constructors to the x_i variables. The query is satisfied if the event e is reachable and the formula ϕ is true. ϕ can be another **event**, $\textbf{inj} - \textbf{event}$, *attacker* or conjunction, disjunction and negation of those terms. When ϕ is another event e' the query is a *correspondence*

assertion [34] that express the relationship *"if an event e has been executed, then even e′ has been previously executed"*. The **inj − event** keyword is used to specify one-to-one correspondences between events. Authentication can be captured using (injective) correspondence assertions.
ProVerif translates the applied-pi calculus process into Horn clauses and tries to falsify the correspondence assertions. The correctness theorem of ProVerif guarantees that if the tool terminates and verification succeeds, no attack exists for an unbounded number of sessions and messages. When a query is false the tool reconstructs a trace that show how an attacker may be able to violate the query.

For a more complete coverage of the ProVerif capabilities, we refer the reader to the ProVerif manual [10].

2.3.2 *WebSpi*

WebSpi [5] aims to offer a set of idioms that are useful in modeling web applications and offer them as a ProVerif library. Figure 4 gives a schematic representation of a typical WebSpi model. Users surf the web using browsers and interact with web applications that are hosted by servers. A user can run multiple browser and a server can run multiple web applications.

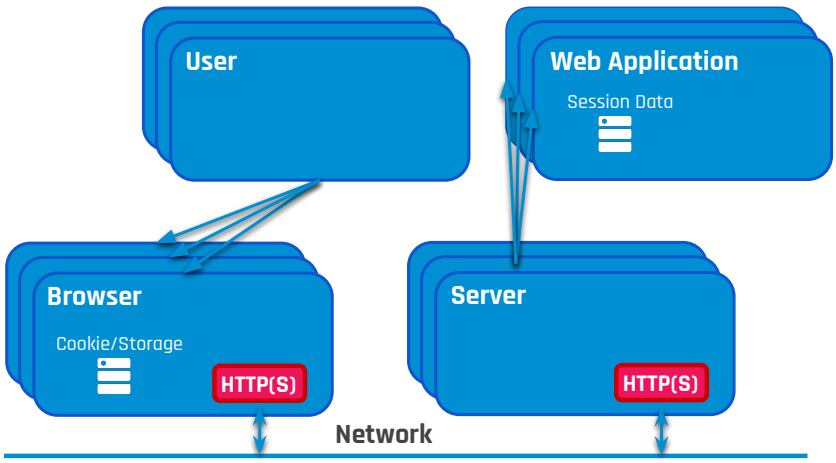


Figure 4: WebSpi Architecture Diagram

Browser and servers communicate using the HTTP(S) protocol over the public network (channel) using a detailed representation of HTTP messages. The term

$$\begin{aligned} &httpReq(uri(protocol, domainHost(subdomain, domain), path, params),\\ &\quad headers(referrer, cookies, notajax()),\\ &\quad httpGet()) \end{aligned}$$

represents, for example, a regular (non-AJAX) HTTP GET request with cookies. Cookies can be associated to specific paths of the domain and support the `Secure` and `HttpOnly` flags.

The standard behavior of servers and browsers,that includes TLS connections, cookie handling and HTTP headers are modeled by the `HttpServer()` and `HttpClient(b: Browser)` processes. The HTTPS connections are modeled by encrypting each HTTPS request with a fresh symmetric key, that is encrypted with the server public key. The response is encrypted using the same symmetric key that is received by the server. The `HttpClient` process models the browser behavior: it automatically handles *redirection* responses, appends the `referer` header to the navigation requests and handles the *Same Origin Policy* for AJAX request directed to the same domain or to servers accepting cross domain (`XDR` flag) requests. It provides API for making HTTP requests, AJAX fetches and reding/writing the cookie storage trough private channels associated with each browser. Writing to the `browserRequest(b)` channel, for example, makes the brower send an HTTP request and the `newPage(b)` channel is used to read the contents of the fetched page.

Modeling Web Applications Using WebSpi

To model a web application using the WebSpi library, typically the user needs to write three processes

- a server-side process representing the website, which interfaces with the `HttpServer` API (the `httpRequest` and `httpResponse` channels) and implements the different paths and pages of the web applications. This process might use ProVerif tables to implement server-side sessions and store data in databases.
- a client-side process that models the actions that are execute on each page by the `JavaScript` loaded by the browser. This process directly uses the `HttpClient` APIs to make AJAX calls or redirect the user to other pages.
- a user process representing the behavior of a human who uses a browser to access the web application. This process uses the `HttpClient` APIs to

click on links, fill forms etc. In some cases this process is combined with the previous one.

Customizable Attacker

The WebSpi library inherits from ProVerif the standard symbolic active (Dolev-Yao) attacker, that controls all public channels, can impersonate entities but cannot guess secrets or access private channels. Moreover, the attacker can create and modify data, encrypt and decrypt messages, but cannot break cryptography.

By default, all WebSpi APIs (channels, tables etc.) are private and can be accessed indirectly by the attacker through the `AttackerProxy` process. The user can set some global flags to enable or disable attacker access to WebSpi resources.

Moreover it is possible to directly modify the attacker capabilities by writing custom processes that mediates attacker access to Private resources. For example, to model XSS on a page, we can write a process that outputs that page on a public channel. This enables the attacker, that have access to all public channels to have a reference to that page so that she make AJAX requests from the compromised page using the browser API.

3

RELATED WORK

WPSE [13] extends the browser with a security monitor for web protocols that enforces the protocol flow, the integrity of messages and their confidentiality. A protocol is specified as a finite state automaton where the expected order of messages and their syntactical structure is defined. Each transition is guarded by the expected shape of the message at that point in the protocol run, a *secrecy policy* that specify which part of the message must be confidential among a set of origins and an *integrity policy* that defines runtime checks over parts of subsequent messages. This model is expressive enough to prevent many attacks that can be found in the literature. The authors, however, acknowledge that some classes of attack cannot be prevented by relying only on a browser-side security monitor. In particular, they define as out-of-scope all the attacks that do not deviate from the expected protocol flow, as for example the *automatic login CSRF* [5]; all the attacks that by their nature cannot be observed by the browser, as network attacks, attacks that are only observable on back channels and attacks that deviate from the protocol flow only on the server side (as, for example the HTTP variant of the *IdP mix-up* [18] attack).

OAuthGuard [21] is a browser extension that aims to prevent five types of attacks on OAuth 2.0 and OpenID Connect. It leverages the browser visibility of protocol messages to detect and prevent CSRFs, impersonation attacks, misuse of the authorization flow, unsafe transfers of tokens and secrets and privacy leaks. As with WPSE, this monitor cannot detect vulnerabilities that can be detected only by an analysis of the server-side, as with the *IdP mix-up* attack. Moreover, since it applies a fixed set of rules that are implementation agnostic, it needs manual whitelisting of the RPs (i.e., those using *gigya*) in which its mitigation (like strict referrer validation [6]) might break the intended protocol flow.

Even if Google recently is showing interest in extending its Chrome browser to monitor security protocols [2], it seems that the solution they are deploying is highly bound to their specific implementation of SAML SSO and is not capable of protecting other protocols or TTPs. Hence, in the general case, users that wants the protection enabled, needs to explicitly install a browser extension. This optional installation step requires the users to take action and trust a browser extension to intercept all the traffic of their security protocol runs. Moreover, a browser-side extension can be tampered with, or uninstalled, by those who have access to the local machine. A server-side monitoring mechanism, on the other

hand, can be deployed transparently form the user, that ideally, is not aware of it being in place and is not able to tamper with it or disable it.

All these browser-side approaches are explicitly focused on SSO and do not consider online payment (CaaS) protocols, where a web attacker is not interested in attacking honest users but his goal is to attack the implementation of the server to, for example, shop for free. Hence, in this setting is essential that the monitor cannot be disabled or bypassed.

InteGuard [35] focuses specifically on the server-side of the integrator (SP), as its code appears to be more error-prone than that of the other protocol participants. InteGuard is deployed as a reverse proxy in front of the service provider that inspects invariant relations within the HTTP(S) messages before they reach the web server. Different types of invariants are automatically generated from the network traces of SSO and CaaS protocols and enable the monitor to link together multiple HTTP sessions in transactions. The transaction-specific invariants enable InteGuard to monitor *back channels*, the messages that are sent from the integrator to the trusted third party without being channeled by the browser. These invariants can also be used to capture transaction-related communication between the UA and the TTP that the reverse proxy cannot directly observe. The authors explicitly stress that InteGuard does not offer protection on the provider-side (TTP), expecting further efforts to be made in that direction. In fact the solution is so bound to the integrator side that is difficult to move to a different party. This, however, makes attacks such as some variants of the *unauthorized login by auth. code (or token) redirection* [5] out of scope. The InteGuard approach fails if the traces provided for invariant extraction are not diverse enough or if a specific invariant is not represented in the provided input. If a parameter does change only over long periods of time may not be captured by the input traces, so the proxy may generate false positives during the run-time monitoring phase, breaking honest protocol runs. This behavior can also be harmful if it creates false negatives as with the `sears.com` website, where during experiments the tool was not able to prevent exploitation.

AEGIS [14] synthesizes runtime monitors to enforce control-flow and data-flow integrity, authorization policies and constraints in web applications. The monitors are reverse proxies that are deployed between the users and the application and are generated by extracting invariants from a set of input traces. The tool is tailored to web applications where tasks are performed by humans. This two-party settings makes it less powerful when mitigating vulnerabilities in multi-party web applications as it ignores both the messages between the browser and the TTP and those in the back channels. It however can mitigate many vulnerabilities in MPWAs if those messages are not needed.

BLOCK [22] is a black-box approach for detecting *state violation* attacks. As with the previous approach it extracts a set of invariants from the network traces captured during the interaction of the user and the web application. These invariants are then enforced using a reverse proxy on the server-side. The authors do not explicitly mention web protocols, but through the type III invariants, that model relationships between subsequent request/response paris, many attacks on MPWAs can be mitigated as they can be classified as state violations. As with AEGIS, the fact that it ignores many messages in multi-party interactions makes this approach less powerful than the InteGuard solution. The author acknowledge that their approach cannot guarantee completeness and correctness of the inferred invariants, and that in practice human intervention is required to prevent false positives. An interesting future direction that is mentioned is however the automatic verification of likely invariants they extract.

In general, these works highlight that using network traces makes the resulting monitor sensitive to the choice and richness of the selected request/responce sequences. Moreover it does not guarantee completeness or correctness of the application model (invariants) that is enforced during run-time.

Guha et al. [19] apply a static control-flow analysis for `JavaScript` to construct a *request-graph*, a model of a well-behaved client as seen by the server application. They then use this model to build a reverse proxy that blocks the requests that violate the expected control flow of the application, and are thus marked as potential attacks. The authors explicitly mention soundness as one of the main limitations of their work. However a soundness proof would require a formal semantics for JavaScript and the DOM.

Pironti and Jürjens [26] propose a formally-based approach to generate security monitors for legacy implementation of cryptographic protocols. They present an *agent to monitor* (*a2m*) function that turns a spi-calculus specification of a protocol agent into a run-time monitor for its implementation. Assuming that a protocol definition is correct, and thus secure, they derive the specification of a single agent from its observable behavior, that is the messages transmitted over its channels. This specification is used to generate a monitor implementation, using the Spi2Java [27] framework, that observes the protocol sessions of the legacy implementation and stops the protocol execution if it detects any deviating behavior. Their approach is black-box in the sense that it does not require the source code of the application implementing the protocol, but is concerned only with the messages that are exchanged by it and the cryptographic operations executed on these messages. This approach leverages formal methods in the derivation of the monitor implementation, so a trustworthy monitor is obtained. The authors, however, do not provide a soundness proof of the *a2m* that guarantees

that the generated monitor process forwards only the protocol session that are accepted by the agent in the verified protocol specification.

The *a2m* function can only be applied to sequential cryptographic processes defined in the spi-calculus. Hence, this function cannot be directly applied to web protocols as protocol agents are usually [5] written as the parallel composition of multiple *handlers* that process a single application path, without explicitly enforcing a protocol flow. The flow is implicitly modeled by the invariants relations between subsequent messages that are enforced by the agents. Moreover, web protocols specifications rely on data constructors and generic data types, that are not supported by the spi-calculus.

4

DESIGN SPACE ANALYSIS

4.1 MPWA SECURITY MONITORING

We now discuss the deployment options for a security monitor for MPWAs. In general, we can have client-side or server-side monitors. The next sections discuss the details of the different possibilities.

4.1.1 *Browser Extensions*

Browser extensions are plugin modules, typically written in `JavaScript`, that extend the web browser with custom code that has access, through an API to the inner workings of the browser, from user interface to cookie management.

A browser extension is able to observe all HTTP(S) traffic from the browser to any website and is able to edit request/response headers and cookies, inspect or edit request parameters and url fragments, and hook `JavaScript` functions. A part from some API limitations that prevent[1]Google Chrome extensions from modifying the body of the HTTP responses [13], a browser extensions is the most flexible tool to be used as a security monitor for web applications, especially in guaranteeing *secrecy* of confidential values. The WPSE approach of hiding confidential values can, in fact, only be implemented using a browser extension. An extension can replace secret values with random placeholders before they have the opportunity of entering the DOM and replace them whith the actual value when they are sent to a list of whitelisted origins. This way only the placeholder is exposed to client-side scripts, preventing confidentiality violations from XSS or untrusted scripts.

However, Browser extensions suffer from three main limitations for what regards multi-party applications monitoring. The first is that, since that they are placed on the user browser, they have visibility only over the *browser relayed messages*, those HTTP(S) request/response pairs that are channeled through the browser. This makes an extension unable to observe values that travel in the *back channel* between the SP and the TTP. The second limitation, that we briefly mentioned in chapter 3, is the fact that the user is explicitly asked to install the extension if she wants the protection enabled. This makes it impossible to deploy a security

1 The limitation could be bypassed using `chrome.debugger` or `chrome.proxy` API, but is not trivial, and not ideal since a browser update could break this bypass.

mitigation transparently with respect to users, making it less effective, as we cannot guarantee that all users are protected. Finally, being installed on the client-side, an extension can be tampered with or uninstalled by those who have access to the client machine. A malicious user could disable the mitigation by uninstalling the extension or not installing it in the first place.

Client-side Proxies

A proxy installed on the client-side has strictly more capabilities that a web browser extension, as it is able to intercept the complete web traffic between the browser and every other website. It, however, suffers from the same limitations of a browser extension for what regards tampering resistance and transparency of installation/deployment. It is, in fact, very difficult for the average user to install and maintain a proxy, thus we will not consider it as a viable deployment option.

4.1.2 *Reverse Proxies*

A reverse proxy is a *proxy server* that acts on behalf of the server as an intermediary between it and the clients that want to access the web application. Reverse proxies are often used as *web application firewalls* (WAF), as they are able to inspect the HTTP(S) request/response pairs that reach the web-server before forwarding them or deciding to close the connection. This enables them to edit the requests before they reache the server and edit the content of the response before it reaches the client. This way headers, cookies, redirection responses, etc. can all be inspected and edited. Moreover, the deployment of a reverse proxy can be done transparently with respect to users, that do not notice its presence. With a similar solution we can guarantee protection to every user of the application.

This flexibility comes at the cost of being able to observe only the requests that are explicitly sent to the server. All the messages that are relayed by the browser but do not reach the server can only be observed indirectly by inspecting the web application response. The first request, for example, made by the UA to the TTP in any OAuth 2.0 flow is not received by a reverse proxy running on the SP. This proxy however, could be able to inspect the request parameters by parsing the response page that contains the URI the user clicks on to log into the TTP.

Another important limitation of reverse proxies resides in their inability to inspect values that do not leave the user browser, as for example *URL fragments*. This is important as the implicit flow of OAuth 2.0 depends heavily on this feature. When, for example, the token is returned from the TTP to the SP through a redirect, the user browser is redirected to an URL similar to

`https://SP/callback#token=U7HvIxxhxvIA`

When the SP website receives the HTTP request to this URL its reverse proxy is only able to see the URL without the fragment part (`https://SP/callback`), as the token remains on the browser to be processed by the `JavaScript` code of the SP. This code might subsequently send the token to the server, but it may also keep it only on the client side. Therefore, as the message is generated by the TTP through a redirect response to the UA, there is no indirect way for the SP to know the token value if the client-side code do not send it to the server.

Server-side (forward) Proxies

A server side solution for web protocol is not complete if we do not consider *back channels*. For this reason, following what InteGuard [35] proposes, when a reverse proxy is deployed on the integrator side (SP), it is necessary to join it with a forward proxy that is able to inspect the messages exchanged between the SP and TTP. This proxy needs to be connected to the main reverse proxy through an IPC (inter-process communication) mechanism as different messages belonging to the same protocol run, but received by different proxies, need to be considered together.

Throughout the rest of this thesis, when we generically refer to reverse proxy as MPWA monitors we consider the combination of a reverse proxy that monitors the *front channel* and a forward server-side proxy that monitors the *back channel*.

4.1.3 *ServiceWorker API*

The `ServiceWorker` API is a new browser capability that enables developers to define JavaScript workers that augment the web application with a way to manage its network connections even in case the device is not connected to the Internet. A `ServiceWorker` acts as a network proxy that intercepts all incoming and outgoing HTTP requests and can choose how to respond to them. It is intended to manage caching for the web application when the user is offline, but it also offers push notifications access and background sync APIs. `ServiceWorker`s are deployed by adding a `JavaScript` file to the web application and can be deployed transparently, as the user is not required to accept the installation, that happens in the background.

As it is able to intercept HTTP(S) messages, much like a reverse proxy on the server side is able to do, a `ServiceWorker` is a good candidate for web application security monitoring. Moreover, since it is executed on the client-side, it is able to access URL fragments and values that cannot be observed by a traditional

reverse proxy. This way a `ServiceWorker` is a browse-side defense that can be transparently deployed by the server-side.

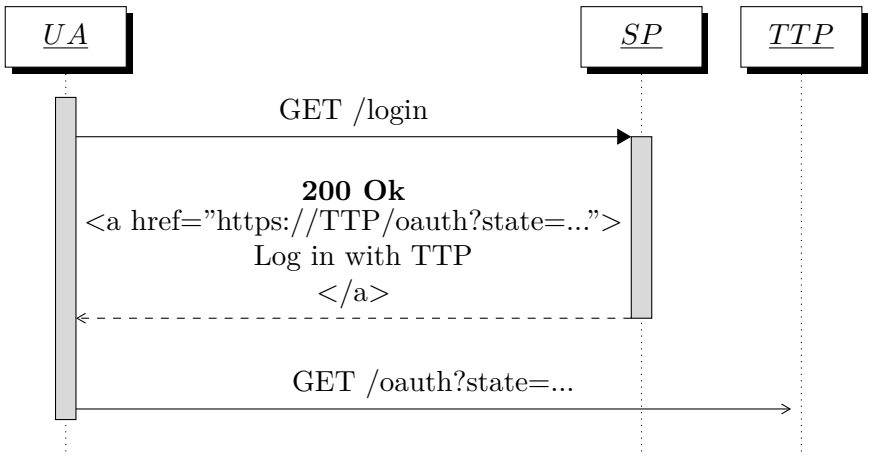
A part from inheriting all the limitations of the reverse proxy solution, `ServiceWorker`s suffer from a more important limitation: they are subject to the Same Origin Policy (SOP) as the API that is available to worker scripts for making HTTP requests is the same that is available to traditional scripts. This means that the `fetch` function cannot read cross-origin responses if they do not have a CORS header and that cookies are not accessible through headers [33]. The impossibility to read cookies does not impact multi-party security monitoring, as cookies are sent by default following the normal browser behavior. Each service worker is therefore implicitly bound to a user session and is not meant to handle multiple ones as a reverse proxy does. The SOP for cross-origin requests, on the other hand, limits the applicability of `ServiceWorker` as a general solution for web protocols.

As an example we present here two variations for integrating the OAuth 2.0 authorization protocols, and show how a `ServiceWorker` is only able to handle one of the two cases. We will consider the first two request/response pairs (round-trips) so the choice of the grant mode is not relevant. Our objective is to inspect the `state` parameter that is generated by the SP when the login url for the TTP is composed. Figure 5 shows two possibilities of generating the login url and presenting it to the user.

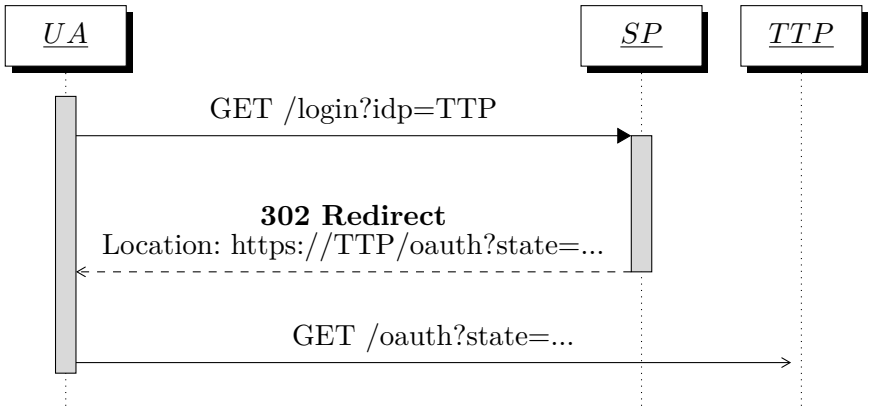
- In the first case the SP generates the URL and embeds it in the page showing a login button as a mean for the user to select which TTP use.
- In the second case the button does not contains the URL of the TTP but sends an HTTP request to the SP which generates the URL and redirects the user to the TTP.

In the first case the `ServieWorker` that is installed on the SP domain is able to read the response body before forwarding it to the browser. It can then parse the page, find the url and inspect the `state` parameter.

In the second case, however, the situation is different. A redirect response is handled by the `fetch` API differently from the page response, and it is necessary to use the `redirect_mode: manual` flag to be able to stop the browser from automatically following the redirect. The `ServiceWorker` needs to stop the redirect as the navigation event to TTP cannot be received by workers installed on the SP. Setting the redirect mode to manual, however, does not enable the worker to read the redirection URL. In fact the `fetch` specification [33] explicitly mention that if a redirection is cross-origin, the `url` field of the manual redirection response does not contain the correct URL, but the URL before the



(a) Embedding a link in the page



(b) Redirecting the user to the *TTP*

Figure 5: OAuth 2.0 first round-trip variations

redirection. This prevents the worker from accessing the URL contents and the `state` parameter unless the *TTP* has a CORS header on the login page.

4.1.4 *Discussion*

In summary, a browser extension is an effective security monitor as its capabilities enable us to inspect and possibly edit any kind of message, even sent to different origins. However its deployment method, that require explicit user interaction, and more importantly, is under its complete control makes this solution fall short when we require it to be present at all times.

`ServiceWorker`s enable us to deploy a transparent client-side mitigation by acting on the server side. This mitigation, however has some limitation in its capabilities, as it is limited by the SOP. Moreover, even if it is possible to

prevent untrusted scripts and XSS to tamper with it[2], it can be uninstalled by the user that controls the browser. This possibility is less harmful than the browser extension case, as we can, through `JavaScript`, check the presence of the worker script and install it if it has been uninstalled. So assuming that the `ServiceWorker` can be installed in every page of the application and that we invalidate the protocol runs in which the application do not detect the presence of a monitor, we can have more confidence that the monitoring script is installed.

This problem is solved by a reverse proxy, that is completely deployed on the sever-side and cannot be tampered with or bypassed. The reverse proxy is the most powerful tool for security monitoring, however it is limited by the fact that it can only see directly the requests that are sent to the monitored application. To make it a viable option for multi-party web application we need to collect indirect information about the protocol runs and join it with a proxy on the back channels. Still it cannot inspect URL fragments, that can be read by both `ServiceWorkers` and browser extensions.

For what regards the positioning of the monitors, the browser extension can inspect every browser relayed message but it cannot access back channels, that reverse proxies on both the SP and TTP can see. `ServiceWorker`s are effective on a subset of the messages that the extension can inspect, as, for example, even if they are deployed on the TTP, they are not able to inspect back channel requests as they are usually not made with full-fledged browser but using curl or similar HTTP libraries.

Given the capabilities and positioning options for server-side deployed monitors, we think that by using together `ServiceWorkers` and reverse proxies we can still have enough capabilities and visibility of the protocol messages to be able to mitigate most of the attacks that a browser extension is able to mitigate, but without the users to take action for the protection to be in place. The ideal deployment would be to install a single `JavaScript` file on the service provider and trusted third party to be able to mitigate most vulnerabilities and use proxies for the ones that cannot be detected by `ServiceWorkers`.

4.2 ATTACKS AGAINST MPWAS

We now investigate how the capabilities of the different type of monitor and their positioning choice affects their ability to mitigate attacks on web protocols.

Table 2 shows the list of attacks to both *OAuth 2.0* and *PayPal* that can be found in the literature. We focus in particular on those presented in [20, 23, 29, 5, 32, 18, 25, 30, 28]. For each attack we specify if it is possible to detect or prevent

2 Setting `serviceWorker` to `null` on both `navigator` and `navigator.__proto__` we can prevent the page from accessing the `ServiceWorker` API.

it with an extensions on the user agent (*ext*), a service worker (*sw*) or reverse proxy (*revp*) on the relaying party (SP) or the trusted third party. For a short summary of the attack that are considered in this chapter we refer the reader to Appendix A.

Attack	**UA**	**SP**		**TTP**	
	ext	*sw*	*revp*	*sw*	*revp*
OAuth 2.0					
Session swapping (A4[29], 10.12[20])	✓	✓	✓	×	×
Social login CSRF on stateless clients (V.C[5], 10.12[20])	✓	✓	✓	×	×
Token replay implicit mode ([28, 32] 10.16[20])	✓	✓	×	×	×
IdP Mix-Up attack (3.2[18]) (HTTPS variant)	✓	✓	✓	×	×
Code/State Leakage (III[17]) (3.3[18])	✓	✓	✓	×	×
Unauth. Login by Auth. Code Redirection (V.D[5], 10.6[20])	✓	×	×	×	✓
Resource Theft by Access Token Redirection (V.D[5])	✓	×	×	✓	✓
307 Redirect attack (3.1[18])	✓	×	×	✓	✓
Token/Code theft via XSS (A2[29])	✓	×	×	×	×
Access token eavesdropping (A1[29])	✓	✓	✓	×	×
Cross Social-Network Request Forgery (V.D[5])	✓	✓	✓	×	×
Naive session integrity attack (3.4[18])	✓	✓	✓	×	×
IdP Mix-Up attack (3.2[18]) (HTTP variant)	×	✓	✓	✓	✓
Open Redirector in OAuth 2.0 (4.2.4[23], 10.15[20])	✓	✓	✓	✓	✓
Force/Automatic login CSRF (V.C[5], A5[29])	✓	✓	✓	×	×
Login CSRF (attacking the connect request)	✓	✓	✓	×	×
Facebook implicit AppId Spoofing (4.2[31], #6[28])	×	×	×	✓	✓
Social login CSRF through AS Login CSRF (V.C[5])	✓	×	×	✓	✓
PayPal					
SP_M *PayeeId* replay in SP_T (#3[28], IV.A[25])	×	×	✓	×	×
T_1 at SP_T *Token* replay in T_2 at SP_T (#5[28], IV.A[25])	×	×	✓	×	×
NopCommerce gross change in PDT flow (III.1[30])	×	×	✓	×	×
NopCommerce gross change in IPN callback	×	×	✓	×	×

Table 2: Attacks on OAuth 2.0 and PayPal

In general we can see that in the SSO setting, an extension on the browser is the most powerful tool, as in most cases it can, on its own, detect and block most of the attacks. The table also shows that by using together a service worker or a reverse proxy on both the service provider and the TTP, it is possible to prevent the same attacks as an extension, but without having the user to explicitly install a plugin on his browser. This confirms our hypothesis that we can use server-side deployed monitoring with the same effectiveness of browser-side defenses.

Moreover, we have that in some cases a server-side monitor is able to detect some attacks that are not detectable by the browser side, especially when these attacks can only be detected on the TTP side. The only attack that requires an extension to be prevented is the *Token/Code theft via XSS*, which is mitigated by replacing secret values with random placeholders. This cannot be achieved by neither a reverse proxy or a `ServiceWorker` as it is impossible to replace these values back in the cross-origin requests that both solutions are not able to inspect.

The CaaS setting shows a different trend. Even if it is possible to detect the attacks on the browser side, as previously mentioned, it is not safe to do so. The attacker in this scenario is the user himself that tries to misuse the integrator to obtain a specific behavior, for example shopping for less or for free. This is very different from the SSO setting, in which the attacker usually misuses the integrator to attack another honest user to login as him or to log him as the attacker. We can say that in the SSO setting we want to protect the honest users, but in the CaaS case we want to protect the integrator from malicious users. A browser side protection, being it a plugin or a `ServiceWorker` is only able to protect honest users that interact with broken or non compliant integrations, but is useless if the bugs on the integrator are used to directly attack it, since user agent monitors can always be bypassed using different HTTP clients (as for example curl). We can also see that in this setting is generally necessary to inspect values transmitted on the *back channels*, since these are the only values that cannot be tampered with by malicious users of the integration. Moreover, CaaS integrations usually implement some mechanism of asynchronous notification (like PayPal IPN or Stripe Webhooks) that do not involve the browser, so security monitors for these integrations need to be aware of these messages to be able to detect or prevent attacks.

4.2.1 *Discussion*

Despite their power, browser-side security monitoring, in some settings (namely CaaS) does not offer a strong enough protection to prevent attacks on the integrator side. This can be seen in the *shop for free* attack in [25](IV.A), in which a malicious user replaces the *merchantId* value with its own account id to be able to pay himself instead of paying the merchant. A browser side monitor could be able to detect the attack but we cannot be sure that the attacker interacts with the integrator website using a monitored user agent.

For this reason, depending on the specific protocol that is being considered, it is necessary to be able to automatically select the appropriate protection between client-side and server-side. This choice should be based on the security properties that are needed to be satisfied for the protocol at hand: if a `ServiceWorker` is

enough to satisfy the security policy it is safe to deploy a browser-side security mechanism. If a browser-side protection cannot satisfy the security policy a reverse-proxy based security mechanism needs to be deployed. The security policy gives also some hints about the positioning of the monitors: having a high-level view of the protocol flow enables us to which entities needs to be protected for the properties to hold.

A single monitor might not be sufficient to cover all possible attacks. From table 2, however, we can see that by using an appropriate server-side (deployed) protection on both the TTP and the SP, we can mitigate all vulnerabilities that do not involve XSS. These monitors are independent from each other, as they mitigate different types of vulnerabilities. Each monitor, on its own, has enough visibility over the cross-origin messages to mitigate a subset of the all the possible flaws that affect the integration. By performing the union of these subsets we can confidently detect all the known logic flaws without using a browser extension.

In summary, an explicit security policy for each protocol gives us a way to reason about the required mechanisms that needs to be deployed for it to hold. In particular, as we will see later, a set of security properties, makes us confident about the choice in positioning of the monitor, the number of distinct monitors and the type of protection (client or server side) that is needed for the complete set of properties to hold on any possible implementation.

4.3 OUR IDEA

Differently from all the related work, we turn our attention to the TTP. The provider defines the protocol specification and the integration mechanism that websites need to use for incorporating it in their services. Typically, together with the documentation, major TTPs provide Software Development Kits (SDKs) to facilitate the integration for the SPs and to limit the possibilities to introduce coding errors. This however, has shown to be insufficient to guarantee security as these SDKs contain implicit assumptions [37, 32].

The TTP should be able to provide, together with the SDK and documentation for its protocol, a monitor that can be deployed on the integrator as either a ServiceWorker or a reverse proxy and that blocks each non-conforming connection. This monitor should be generic enough to be deployed without modification on an interoperable integrator and it should guarantee compliance with the protocol specification. To this end, the provider, that has the best knowledge of the protocol, is able to declare its security policy and to select the best possible configuration of monitors so that the protocol is secure. A TTP-side monitor, if necessary, can be directly deployed by the provider. A monitor on the SP, if

installed by the integrator, guarantees that every protcol run meets the TTP security policy.

5

PROPOSED APPROACH

We now define the steps that are needed to automatically generate security monitors from formally-verified specifications of web protocols. Figure 6 shows an overview of the entire process that starts with a protocol specification and ends with an executable monitor.

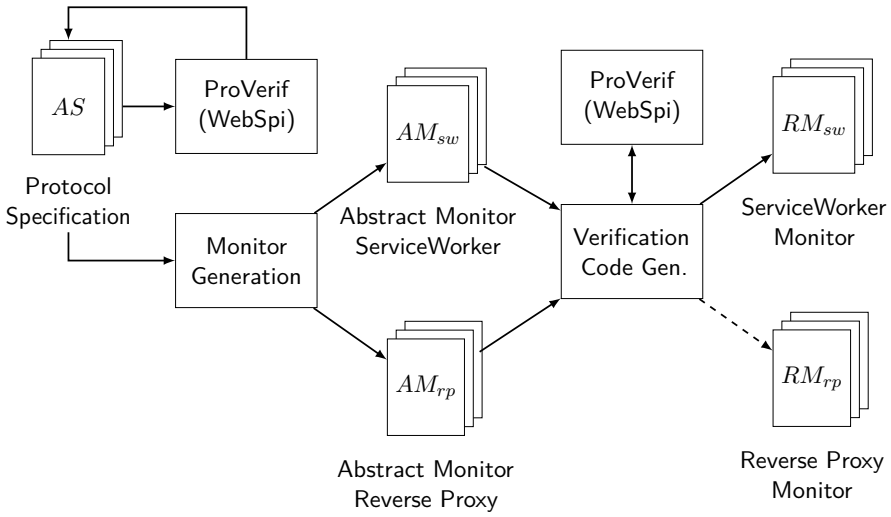


Figure 6: Monitor Generation Flow

PROTOCOL SPECIFICATION. Let us consider a generic web protocol. The trusted third party, which has the best knowledge of the service it provides, derives a specification of the web protocol based on the observable behavior of the entities it involves. This specification should be implementation agnostic and concern only protocol messages and invariant relations that need to be enforced for it to be secure. We will refer to this specification as the abstract system AS, as it abstractly specifies the ideal interaction between honest entities. A security policy P is part of the specification and defines which properties the protocol needs to satisfy to be correct. The security property needs to hold on the abstract system, even in presence of an all-capable attacker (Dolev-Yao and web attacker) $\mathcal{A}$:

$$\mathcal{A} \parallel AS \qquad \models P$$

This is proved using ProVerif.

ABSTRACT MONITOR GENERATION. The abstract system and the security policy are input to the monitor generation phase, that assume the security of AS with respect to P. This phase generates an abstract monitor $AM = AM^{rp}, AM^{sw}$ that respectively model a reverse proxy or a `ServiceWorker` installed on the selected entity. The entity that should be monitored is the input to the monitor generation function and can be selected between SP and TTP. The abstract version of the monitor is derived from AS and it enforces the invariant relations that are needed in the ideal protocol runs. However, the two variants (AM^{rp} and AM^{sw}) are constrained in the operations they can execute by the capabilities of the respective technology.

ABSTRACT MONITOR VERIFICATION. The two monitors are verified in isolation by checking, using ProVerif, that the same properties P that hold for the ideal protocol also hold on the system composed by a broken implementation of the monitored entity and the abstract version of the monitor.

The verification is ordered by the ease of deployment of the final monitor: we first start by verifying that P hold with the `ServiceWorker` monitor alone, then with the reverse proxy alone and finally with both monitors. When a solution in which P is satisfied is found, we stop and continue to the next phase using the satisfying configuration.

CODE GENERATION. In the final phase a real monitor RM is generated from the abstract monitor AM that is selected in the previous phase. The real monitor is obtained by translating the abstract specification into executable code. A different code generator is used for the different types of monitors:

- Reverse proxies are translated to `python` plug-ins for the mitmproxy [15] HTTP proxy.
- `ServiceWorker`s are translated to `JavaScript` files.

The translation phase is a one-to-one translation between a pi-calculus specification and executable code. The abstract monitor specification needs to be detailed enough to represent as closely as possible the real monitor, except for small implementation details. This way it is extremely easy to add another compilation target, to, for example, generate plug-ins for Apache or Nginx.

RUNTIME MONITORING. During run-time, a reverse proxy is deployed as a combination of two proxies, one for the front channel and one for the back channel, that capture requests to and from the SP and block non conforming connections. A `ServiceWorker` is deployed as a single script that is included in each page of the web application. This scripts installs the worker script if it was not already present. The worker signals an error to the user and block the connection if any deviation is detected.

5.1 PROTOCOL SPECIFICATION

As previously mentioned, the specification of the abstract system AS encodes the behavior of the honest participants to the protocol in a successful protocol run. Each participant describes the action that an ideal implementation would execute to be able to communicate with the other parties. It does not contain, however, implementation-specific details as it is only concerned with the observable behavior of the entity: the messages it exchanges and the integrity checks it executes on those messages. Optionally the entities can use a private storage when needed.

The protocol specification contains the encoding of the security policy P. The security policy is written in terms of correspondence assertions, reachability and secrecy properties related to the critical security events defined in the AS. Each security relevant action in the abstract system is labeled with an event, that can be referenced by P.

The pair $< AS, P >$ is specified as a ProVerif script which uses the WebSpi library. In particular, each server-side party is implemented as a server application in the WebSpi model and the `JavaScript` and user actions are encoded as client processes that use the *Browser* API. We, however, restrict this model so that the protocol entities are encoded only as two kind of processes: $ServerApp$ and $UserAgent$.

Definition 5.1.1. A $ServerApp$ process P with parameters fv is a ProVerif pi-calculus process with the following structure:

$$P(fv) = defs; (hp_1|hp_2|...|hp_n).$$

where $defs$ is a process composed of restrictions only, which defines local names for the entire application

$$defs; P = \textbf{let } x_1 : t_1 = M \textbf{ in } ... \textbf{ in let } x_2 : t_2 \textbf{ in } P$$

and $hp_1, ..., hp_n$ are *handler processes* with the following structure

$$\textbf{in}(httpServerRequest, ...); \textit{«path conditions»}; \textit{«content»};$$
$$\textbf{out}(httpServerResponse, ...).$$

where the *path conditions* are a sequence of let pattern matches or if statements which match the url path, and http request type. The *content* is a sequence of one or more of the following operations: if and let statements, insert or get from tables, creation of new names and event executions. The *path conditions* in a $ServerApp$ **must** me mutually exclusive.

A *ServerApp* is a general model of application with multiple paths, each of which supports a request type. Each path is handled separately by a different *handler* process. Since the path conditions are mutually exclusive, each execution of the ServerApp process is deterministic: once an *HttpRequest* is received, a single *HttpResponse* is produced. This encoding makes the parallel composition of handler processes equivalent to a more imperative `switch/case` statement.

Definition 5.1.2. A $UserAgent$ process $P(b : Browser)$ is the parallel composition of • *user-initiated actions* that start the loading of a page; • *page handlers* that act on the received page; with the the following structure:

$$\textbf{in}(newPage(b), ...); \textit{«path conditions»}; \textit{«content»};$$
$$\textbf{out}(action(b), ...).$$

where *action* can be • *pageClick*, • *browserRequest*, • *ajaxRequest*; and the **in** and *path conditions* are omitted for the user-initiated actions.

A $UserAgent$ process models the action that can be executed by the user. A user can initiate a protocol run by requesting a page or react to a received page by clicking the link or initiating another request.

The protocol parties are encoded as $ServerApp$s and $UserAgent$ processes by the protocol designer. These processes are executed by replicating the processes indefinitely in a network with a Dolev-Yao attacker and a web attacker modeled by the composition of a $MaliciousApp$, $OpenRedirector$ and the WebSpi attacker that can initiate requests to every page. Moreover, to simulate the possibility of fetching resources from unsafe origins, the $UAFetchResources$ process is run in parallel with each $UserAgent$. This process fetches a resource from a malicious application in the context of every page of the honest application. This way attacks such as the *state leak* of [17] can be detected.

The $MaliciousApp$, $OpenRedirector$ and $UAFetchResources$ are provided by our extension of the WebSpi library, together with the support for modeling the `referrer-policy`, url fragments and `ServiceWorkers`.

A ProVerif script that encodes the AS and P can be run to verify that the policy P is satisfied. All successive steps assume the protocol is correct and ProVerif terminates with a positive answer when executed with $< AS, P >$ as input.

5.2 MONITOR GENERATION

Given a ProVerif script modeling the pair $< AS, P >$ and the name of the process encoding the entity to be monitored E, we define two *monitor generation*

functions that generate the ProVerif process encoding respectively a reverse proxy and a `ServiceWorker` that enforce the integrity constraints that are encoded in the process entities in AS.

5.2.1 *Generating Reverse Proxy Monitors*

Figure 7 shows the $a2m^{rp}$ function, a modified version of the $a2m$ function of [26] that turns a agent specification written in the applied-pi calculus into a monitor specification. The rp superscript stands for *reverse proxy* as this function turns a $ServerApp$ (Def. 5.1.1) process into a reverse proxy for that application.

$$a2m^{rp}(P) = \mathcal{M}^{fv(P)\cup fn(P)}_{(\emptyset,\emptyset)}(P)$$

$$\begin{aligned}
\mathcal{M}^{K}_{(Q,D)}(\mathbf{0}) &= flushBuffers(Q); \mathbf{0} \\
\mathcal{M}^{K}_{(Q,D)}(P|Q) &= \mathcal{M}^{K}_{(Q,D)}(P) | \mathcal{M}^{K}_{(Q,D)}(Q) \\
\mathcal{M}^{K}_{(Q,D)}(\texttt{new } a; P) &= \mathcal{M}^{K}_{(Q,D)}(P) \\
\mathcal{M}^{K}_{(Q,D)}(\texttt{let } d = t \texttt{ in } P) &= \begin{cases} \texttt{let } d = t \texttt{ in } \mathcal{M}^{K\cup\{d\}}_{(Q,D)}(P) & \text{if } t \in K \\ \mathcal{M}^{K}_{(Q,D\cup\{\texttt{let } d=t\})}(P) & \text{otherwise} \end{cases} \\
\mathcal{M}^{K}_{(Q,D)}(\texttt{if } t \texttt{ then } P) &= \begin{cases} \texttt{if } t \texttt{ then } \mathcal{M}^{K}_{(Q,D)}(P) & \text{if } t \in K \\ \mathcal{M}^{K}_{(Q,D\cup\{\texttt{if } t \texttt{ then}\})}(P) & \text{otherwise} \end{cases} \\
\mathcal{M}^{K}_{(Q,D)}(\texttt{in}(c,d); P) &= \texttt{in}(c,d); \mathcal{M}^{K\cup\{d\}}_{(Q\cup\{(mch(c),d)\},D)}(P) \\
\mathcal{M}^{K}_{(Q,D)}(\texttt{out}(c,t); P) &= flushBuffers(Q); \texttt{in}(mch(c),t); \\
&\quad doChecks(K\cup\{t\}, D); \texttt{out}(c,t); \mathcal{M}^{K\cup\{t\}}_{(\emptyset,\emptyset)}(P)
\end{aligned}$$

Figure 7: Monitor Generation Function

The function is defined through an auxiliary function $\mathcal{M}$ that takes four arguments, the process P, the set of known terms K, the buffered messages Q and the delayed checks D. If the process P to be translated is a match or an assignment (`if` or `let`), the process is translated to the same match or assignment only if the considered terms are known ($\in K$), otherwise the check is delayed (thus added to D). If the agent creates a fresh term the monitor does not execute any action. When the agent receives a value from a channel c, the same operation is executed by the monitor and the data is buffered. This data will be later sent to the channel between the monitor and the agent $mch(c)$ (that corresponds to c) only when necessary or if P is the $\mathbf{0}$ process.The buffered data brings some new

terms in scope, so K is updated accordingly. When the process P is $\texttt{out}(c, t); P'$, all buffered data is forwarded to the agent, so that it can compute t. This term is then read by the monitor (from $mch(c)$) which, since there is a new term in scope, can apply all the delayed checks that refer to t with the *doChecks* function. The *doChecks* function, whose definition is omitted for brevity, selects and reorders the delayed checks so that they can be applied to the known terms. After the monitor have applied all the checks, the term t is sent to the channel c and the $\mathcal{M}$ function is applied to P' with the now freed up buffers. For brevity the cases in which P is an `insert` or `get` operation are not shown, since they are very similar to the `if` and `let` cases.

5.2.2 *Generating ServiceWorker Monitors*

The $a2m^{sw}$ function turns a agent specification encoded as a *ServerApp* (Def. 5.1.1), together with the relative *UserAgent* (Def. 5.1.2) process into a `ServiceWorker` monitor for that agent.

$$a2m^{sw}(UA, P) = s2c(UA, a2m^{rp}(P))$$

Being a client-side defense mechanism that act as a reverse proxy, a `ServiceWorker` monitor is function of both the server-side application and the client-side process. This way, a subset of both the integrity constraints of the server and the client side can be encoded into the final process. The $a2m^{sw}$ is defined as the composition of the $a2m^{rp}$ function and the $s2c$ function. The $s2c$ function turns a server-side application into a worker script that is executed on the client. This composition encodes the fact that the worker script should act as a reverse proxy that has more visibility on the values that can only be inspected by the client.

The $s2c$ function has four main responsibilities:

1. Rewriting the reverse proxy so that it is compatible with the `ServiceWorker` API. This includes rewriting it so that each handler process, instead of receiving a tuple from the *httpServerRequest* channel, receives an event from the $serviceWorkerFetch(b)$ channel. Each HTTP forwarding action is rewritten using a model of the `fetch` API, that uses the $rawRequest(b)$ and $serviceWorkerSendHttpResponse(b)$ channels.
2. Removing all the features and values that are available to the reverse proxy and cannot be accessed by the browser API. This includes the cookie and session handling. Session storage can however be kept by encoding the single implicit session of the service worker into a single constant session in the reverse proxy code. This constant session can be later removed and the code simplified.

3. Joining the client-side integrity constraints into the ServiceWorker process. This is done by inlining into the resulting process the parts of the UA process that refer to the same path conditions as each of the handlers and rewriting it so that there are no name clashes.
4. Turning the parallel composition of disjointed processes (as by the definition of *ServerApp*) into an equivalent chain of `if`/`else` statements. This is needed since the `JavaScript` language does not supports the concept of parallel processes (or threads) and we want the resulting service worker to model as closely as possible the final implementation.

The resulting process is a sequential process that encodes the actions that needs to be executed by a `ServiceWorker` that is activated by the browser whenever a request to the app origin and paths is sent or received. This monitor process uses the $ServiceWorker$ API defined in our extension of the WebSpi library.

WebSpi ServiceWorker API

To provide support for the `ServiceWorker` API, the WebSpi *HttpClient* process needs to be modified to insert an extra step in the processing of HTTP requests. First of all three new channels needs to be defined

fun $serviceWorkerFetch(Browser)$: **channel** [**private**].
fun $serviceWorkerResult(Browser)$: **channel** [**private**].
fun $serviceWorkerSendHttpResponse(Browser)$: **channel** [**private**].

these channels form the ServiceWorker public API accessible to worker scripts. The $serviceWorkerFetch$ represent the fetch event that is sent to a worker script when a request is made. The $serviceWorkerResult$ channel is used to retrieve the response from a $rawRequest(b)$ that is sent from the worker script. This corresponds to the `fetch` function call: a fetch is modeled as writing to $rawRequest$ and accessing the body of the response corresponds to reading from $serviceWorkerResult$. The $serviceWorkerSendHttpResponse$ channel is used to send an HTTP response to the browser from the worker script. This models the return value of the `JavaScript` callback function that is executed upon receiving a fetch event.

The *httpClient* process is modified so that every request that is directed to the scopes (origin and path) of the `ServiceWorker` is first sent to $serviceWorkerFetch$ to be processed by the active worker script. When the worker returns a response to the $serviceWorkerSendHttpResponse$ channel, the *httpClient* process sends this response to the browser *newPage* channel that models the browser loading a new page.

5.2.3 *Post Processing and Rewrite Rules*

The generated monitors might contain inefficient or redundant code, and they might not have some explicit type declarations that are needed for ProVerif to correctly execute. For this reason, a set of simple rewriting rules is applied to the monitors that are generated by the two $a2m$ functions. Then the missing types are filled-in when necessary. The post processing operations that are applied are:

- *Removing unused declarations.* When the process contains local variables that are never used, the corresponding let declaration is removed.
- *Constructor optimization.* When a destructor is used to extract some values from an object and the same object is constructed to be later used or sent to a channel, we rewrite the object construction to reuse the old object instead of creating a new one. When a pattern match is used in a in statement, a new variable needs to be introduced. As an example, the process

  $$\begin{aligned}&\mathsf{in}(\mathit{httpServerRequest}, (\mathit{uri}(= \mathit{https}(), = h, = \mathit{slash}(), \mathit{params}), ...));\\&...\\&\mathsf{out}(\mathit{httpServerResponse}, (\mathit{uri}(\mathit{https}(), h, \mathit{slash}(), \mathit{params}), ...)).\end{aligned}$$

  is rewritten as

  $$\begin{aligned}&\mathsf{in}(\mathit{httpServerRequest}, (u : \mathit{Uri}, ...));\\&\mathsf{let}\ \mathit{uri}(= \mathit{https}(), = h, = \mathit{slash}(), \mathit{params}) = u\ \mathsf{in}\\&...\\&\mathsf{out}(\mathit{httpServerResponse}, (u, ...)).\end{aligned}$$

  removing the useless construction of the Uri object in the last statement.
- *Folding if conditions.* Multiple sequential if conditions without the else branch are folded into a single if statement with the conjunction of all the conditions as the predicate.
- *Declaration inlining.* When a local variable is used only once, its definition is inlined into the body of the process.
- *Type Declaration.* Types are declared for each argument of all in statements where it is possible to infer those types from the context.

5.3 MONITOR CORRECTNESS

Let us consider a protocol specification (AS) in which we have three parties, a user agent UA, a relaying party SP and a trusted third party TTP. We assume

that for some properties P, P hold for all possible sessions and messages between the three parties in parallel with a Dolev-Yao attacker (and web attacker) $\mathcal{A}$.

$$\mathcal{A} \parallel TTP \parallel SP \parallel UA \qquad \models P$$

This can be shown using proverif by encoding P as correspondence assertions, reachability queries and secrecy queries. If proverif terminates with a positive answer, the correctness theorem in [9] guarantees that the query is true on all traces of the applied pi calculus processes in parallel with any other arbitrary attacker process.

We now generate the abstract monitors for the SP process

$$AM_{SP}^{rp} = a2m^{pr}(SP)$$
$$AM_{SP}^{sw} = a2m^{sw}(SP)$$

that respectively represent a reverse proxy monitor and a ServiceWorker. We want to show that if we run both these monitors in parallel with a broken implementation of the SP (SP_b) the properties P continue to hold. Let us define the systems in which these monitors are run in parallel with SP_b, the orther entities and the attacker. There are three possible configurations:

$$S^{sw} = TTP \parallel SP_b \parallel (AM_{SP}^{sw} \parallel UA) \parallel \mathcal{A}$$

where the `ServiceWorker` is installed on the honest user agent and the broken SP is left unprotected;

$$S^{rp} = TTP \parallel (SP_b \parallel AM_{SP}^{rp}) \parallel UA \parallel \mathcal{A}$$

where the broken SP is run behind a reverse proxy;

$$S^{*} = TTP \parallel (SP_b \parallel AM_{SP}^{rp}) \parallel (AM_{SP}^{sw} \parallel UA) \parallel \mathcal{A}$$

where both protections are active.

To verify which configuration satisfies P, we start from S^{sw}, run ProVerif and inspect the output. If all properties in P are satisfied, only the `serviceWorker` monitor is sufficient for the protocol to be secure with respect to P. If it is not the case the procedure continues to S^{rp} and then to S^{*}. The resulting monitor is thus the configuration for which every property in P is satisfied. When it is not possible to reach a satisfying configuration an error is shown to the user. Depending on the definition of SP_b it could be possible to prove that the procedure always terminates, but it is left as future work. In our experiments, however, we did not see any failure case in this procedure.

5.3.1 *The Broken SP Process*

The definition of SP_b is critical for our approach, as its implicitly encodes the assumptions we make on the level of compromise that the SP could be subject to for the monitor to correctly protect its users.

ORIGINAL IMPLEMENTATION

We trivially show that if $SP_b = SP$, $S \models P$. In this case, since SP_b is the proven-correct SP process, the presence of the monitor is irrelevant as it performs the same checks that the SP process performs.

NON-COMPLIANT IMPLEMENTATION

We derive from SP a non-compliant process $SP \setminus C$ by removing all the integrity checks that the entity SP does on its messages. Doing so we obtain a process that is interoperable with the other entities, but whose possible requests and responses are a superset of those produced by SP. Intuitively this process models an integrator website that may have implementation errors and may skip some integrity checks. Using proverif we can show that if $SP_b = SP \setminus C$, $S \models P$.

ARBITRARY PROCESS

We then want to prove the stronger property that if SP_b is an arbitrary process that does not leak any secrets (ie, values initially unknown to the attacker) on public channels, we still have that $S \models P$.

This arbitrary process is defined as capable of receiving and sending messages on any channel and do computation on the values in its knowledge. It however cannot write on public channels values that were not initially known to the attacker. This way it is able to generate arbitrary messages and write them to the SP private channels. We want to show that if such process plays the role of the SP_b, the traces permitted by the monitor AM_{RP} satisfy the properties P.

In ProVerif, however, is not possible to define any additional arbitrary processes except for the implicit Dolev-Yao attacker.

One possible formulation for an arbitrary process can be obtained by reusing the implicit attacker and encoding the absence of leakage by an anonymizer process $AN(s_i, s_o, p_i, p_o)$. The anonymizer process forwards messages to and from two pair of channels (one public (p_i, p_o) and one private (s_i, s_o)), anonymizing secret values that are forwarded through it from private to public channels and applying the inverse transformation to values that come from public channels and are directed to private channels. This process needs to be generated starting from the specification of the SP by inspecting the channels that are used and applying the anonymization when a value that is marked secret is sent through it. A secret value is replaced by a random handle and only this handle is shared with the attacker. These handles do not increase the knowledge of the attacker, that can still generate arbitrary messages composing the random values. An handle is

translated back only if it is channeled through the AN process. This way we can use the implicit attacker as a generator of arbitrary messages: $SP_b = (AN \parallel \mathcal{A})$. Another formulation could be obtained by encoding the arbitrary process into a ProVerif prolog predicate. This can be done by generating additional rules during the Horn clause generation phase of the ProVerif solving process.

Neither of these formulation is satisfactory and adds a large overhead with respect to the formal guarantees that the arbitrary process gives for the generated monitor. We decided, thus, to implement the SP_b process as the non compliant process $SP \setminus C$.

5.4 CODE GENERATIONS

Once the abstract monitor has been generated, if the verification phase ends with a positive answer the abstract monitor is guaranteed to satisfy the security policy P. The real monitor now needs to be generated by translating the AM to code. The translation has to preserve the security guarantees of the verification phase. To this end, our code generator maps the applied pi-calculus processes to code as closely as possible, so that it is trivial to prove that the real monitor executes the same operations that are executed by the abstract monitor.

The monitors that are generated by the two variants of the $a2m$ function of Section 5.2 respectively create monitors that use the *httpServer* and the *serviceWorker* WebSpi API. Writing and reading from the WebSpi provided channels is equivalent to calling a function or receiving an event with the same parameters. Moreover a sequential pi-calculus processes can be translated to code by translating each statement to an equivalent programming construct.

5.4.1 *Reverse Proxy: python*

The AM^{rp} process is translated to a `python` plug-in for mitmproxy. This proxy provides a powerful event-based scripting API which offers a convenient way to implement custom processing logic by implementing two callback function `request` and `response`. The `request` function is invoked whenever a new HTTP request reaches the proxy. The request data is saved into the `request` field of the `HTTPFlow` object that is passed to the callback function. In the same way, the `response` function is called whenever a response is sent through the proxy.

To translate as directly as possible the *httpServer* WebSpi API we used some condition variables to simulate the pi-calculus channels. Moreover we translated each handler process to a distinct python thread that is started when a request is received, as it happens in the pi-calculus. Since the path conditions for these

handlers is mutually exclusive (see Def. 5.1.1), we can be sure that at most one thread will be active during the processing of each request/response.

The interface from the mitmproxy API to the translated monitor has the following resposibilities:

- When a new request is received, if its target is one of the paths that is handled by the monitor, all threads are started and the condition variable for the request object is notified. We then wait for the request to be handled until a timeout.
- When a response is sent to the proxy, the condition variable of the response object is notified, waking the thread that was waiting for the response. We wait for the response to be processed by the thread until a timeout and we signal an error to the user if the processing has failed.

This behavior can be encoded as shown in Figure 8.

```
def request(flow: http.HTTPFlow) -> None:
    if flow.request.path.split('?')[0] not in PATHS: return

    CV[flow] = threading.Condition()
    ts = []
    for h in HANDLERS:
        PVThread(target=h, args=(flow,)).start()
    with CV[flow]: CV[flow].notify_all()
    with CV[flow]:
        if not CV[flow].wait(timeout=1.5):
            ctx.log.warn("<!> Wait timeout: request match error!")
            flow.response = http.make_error_response(
                500, "Wait timeout: request match error!")

def response(flow: http.HTTPFlow) -> None:
    if flow.request.path.split('?')[0] not in PATHS: return

    with CV[flow]:
        CV[flow].notify_all()
    with CV[flow]:
        if not CV[flow].wait(timeout=1.5):
            ctx.log.warn("<!> Wait timeout: response match error!")
            flow.response = http.make_error_response(
                500, "Wait timeout: response match error!")
    del CV[flow]
```

Figure 8: mitmproxy interface code

The timeout on the condition variable is used by the interface code to catch errors in the processing of the request or response. Since in the pi-calculus, a failed match terminates the process, we translate this behavior directly and catch the failure by waiting for the process to finish. If it does not finish the process has been terminated by a failed match.

A monitor process is thus translated by rewriting the in(*httpServerRequest*) statements as `wait` on the condition variable and an out(*httpServerResponse*) as a `notify_all` on the same condition variable. The in and out on the monitored channels are translated in the same way.

An example translation is shown in Figure 9 where we report the original process in the comments of the translated code. Here the monitored channels are `mC_1_in` and `mC_1_out` that represent respectively HTTP responses and requests sent to the monitored application.

```
1  def process_1(flow):
2      # in(httpServerRequest, (u:Uri, hs:Headers, req:HttpRequest,
           corr:bitstring))
3      with CV[flow]: CV[flow].wait()
4      assert flow.request is not None
5      u = urllib.parse.urlparse(flow.request.pretty_url)
6      ...
7      # out(mC_1_out, (u, hs, httpGet(), corr))
8      with CV[flow]: CV[flow].notify_all()
9      # in(mC_1_in, (=u, resp:HttpResponse, cp:CookiePair, refp:
           ReferrerPolicy, =corr))
10     with CV[flow]: CV[flow].wait_for(lambda: flow.response is not
           None)
11     resp = flow.response
12     cp = flow.response.cookies
13     refp = flow.response.headers['referrer-policy']
14     ...
15     # let resp1 = ... (* Edit the response *)
16     resp1 = ...
17     ...
18     # out(httpServerResponse, (u, resp1, cp, unsafeUrl(), corr))
19     flow.response.text = resp1
20     with CV[flow]: CV[flow].notify_all()
```

Figure 9: Example translation of a simple forwarding proxy

Back Channels

As mentioned in previous chapters, a monitor that is able to intercept messages on the back channel needs to be composed of two proxies. One reverse proxy that runs on the front channel between the web application and the clients and one that inspect the back channels, all the messages that are sent by the web application to the provider.

To do this we run two instances of mitmproxy, one in reverse mode and one in forward mode, connected through one or more IPC queues. Each request that is intercepted by the back channel proxy is sent to the reverse proxy through the queue. When the reverse proxy, in which the process translation is running, decides to forward the request, the request object is sent through the IPC queue

to the forward proxy that sends the request. This way we have two proxy servers where the forwarding logic is entirely encoded in the translated monitor process. This way an `in` from the monitored back channel is translated as a `get` from the IPC queue, an `out`(*httpServerRequest*) to the back channel is translated as a `put` to the IPC queue.

Figure 10 shows the translation of a process that monitors the back channel. In this code the IPC queue is split into two: `queue_req` to handle requests and `queue_res` to handle responses. Here `mC_2_in` and `mC_2_out` represent respectively http requests received from the back channel and http responses sent to the back channel.

```
1  # in(mC_2_in, (req_uri:Uri, hs:Headers, =httpGet(), ncorr))
2  __backchan_req = queue_req.get(timeout=1)
3  req_uri = urllib.parse.urlparse(__backchan_req['request'].
       pretty_url)
4  ...
5  # out(httpServerRequest, (req_uri, hs, httpGet(), ncorr))
6  queue_req_actions.put(__backchan_req)
7  # in(httpServerResponse, (=req_uri, tres:HttpResponse, cp1:
       CookiePair, rp1:ReferrerPolicy, =ncorr))
8  __backchan_res = queue_res.get(timeout=1)
9  tres = __backchan_res['response']
10 cp1 = __backchan_res['response'].cookies
11 ...
12 # out(mC_2_out, (req_uri, tres, cp1, rp1, ncorr))
13 queue_res_actions.put(__backchan_res)
```

Figure 10: Back channels using queues

This encoding however needs to support multiple sessions. This is done by additional logic on the back channel proxy. When multiple sessions are executed in parallel, the back channel has no means to know which front channel session the request that it is handling belongs to. Therefore it cannot send this request to a single entity. The solution is to send the request to every currently active front channel session that needs to filter the received requests and respond only to the one that is part of the transaction is handling. This is implemented by an additional processing step before sending the requests objects to the translated process.

5.4.2 *ServiceWorker: JavaScript*

The AM^{sw} process is translated to `JavaScript` code that uses the `ServiceWorker` API. This API defines a `fetch` event that is used by the worker script to listen to HTTP requests and uses the standard `fetch` API [33] of the browser to make new HTTP requests.

The translation to `JavaScript` is one-to-one mapping between the *serviceWorker* WebSpi channels and the `ServiceWorker` Browser API. In particular, as explained in section 5.2.2, each channel is mapped to a specific `JavaScript` operation. Moreover, the monitor is implemented as a sequence of if/else statements since multiple processes cannot be encoded in the target language. The only difference between the two is that the `ServiceWorker` API is asynchronous, so we make use of the `async/await` syntax to maintain the structure of the process after the translation. Each match failure is represented as an exception. All exception are caught to show an error message to the user.

An example translation can be seen in Figure 11 where the original process is given as a comment on each statement. We can see that the structure of the process is maintained in the translated code. In particular we can see a single **in** statement at the beginning and a chain of if/else to select the path (here only one is shown).

```
1  /* in(serviceWorkerFetch(b), (u:Uri, cs__1000, ref, sw__p, sw__aj))
       */
2  self.addEventListener('fetch', function(event) {
3    let u = new URL(event.request.url);
4    /* let uri(=https(), =h, =loginpath(), q:Params) = u */
5    if (("https") == ((u).protocol.slice(0,-1)) && (h) == ((u).host)
         && (loginpath) == ((u).pathname)) {
6      let q = (u).search;
7      event.respondWith((async () => {
8        try {
9          /* out(rawRequest(b), (u, cs__1000, ref, sw__p, sw__aj)) */
10         var __fetchd = await fetch(event.request);
11         /* in(serviceWorkerResult(b), (=u, cs__1100:HttpResponse,
               ... */
12         let cs__1100 = __fetchd;
13         ...
14         /* out(serviceWorkerSendHttpResponse(b), (u, cs__1100,
               cs__1101, sw__foo, sw__corr)) */
15         return cs__1100
16       } catch (e) { return errorResponse('match error!'); }
17     })());
18   } else {
19     ...
20   }
21 });
```

Figure 11: ServiceWorker Monitor Example

5.4.3 *Concretization of Abstract Constructors*

The ProVerif specification of the *AS* defines its operations in terms of applications of constructors and destructors. These function symbols are used by the WebSpi

library to represent data-types, url paths, HTML pages, json values and more importantly GET and POST parameters. Since ProVerif does not support record types, each different combination of parameters and values needs to be encoded as its own data constructor (see Sec. 2.3.1). For example, the URL parameters of

```
https://TTP/oauth?client_id=...&redirect_uri=...&state=...
```

are represented as a data constructor with three fields

$$\textbf{fun}\ \mathit{oauthParams}(\mathit{Id}, \mathit{Uri}, \mathsf{bitstring}) : \mathit{Params}\ [\textbf{data}].$$

This encoding needs to be defined during protocol specification and each protocol might encode different collections of params as different types.

For this reason, data constructors cannot be automatically translated, as each one needs the corresponding translation code. Therefore, we provide facilities to define these translations as functions in the target language, while, for efficiency, we implement the library ones directly in the code generation pipeline.

The translated monitor, whenever it needs to apply a data destructor, calls the corresponding function to deconstruct an object into its components. These data construtors/destructors are defined in a configuration file that is provided by the user of the monitor. This file is the way in which the final user of the monitor can provide specific values and the implementation for the abstract constants and types that are defined in the code.

Thanks to the fact that destructors are directly written in the target language and that they represent a configuration parameter for the monitor, different implementations can be provided for different types of behavior. For example, let's consider the difference between *strict* and *lax* URL parameter matching. In strict mode, additional parameters that are present in the URL that are not present in the model represent deviating behavior, whereas in lax mode additional parameters are ignored. The choice between this two behaviors is not always easy and it represents a trade-off between security and compatibility. As an example, assume that we apply lax matching. The protocols where the parameters are a super-set of those present in the model are still considered valid. This behavior allows for implementations to add additional parameters for their specific needs and makes the monitor more compatible. However, using lax parameter matching makes the monitor vulnerable to attacks similar to those presented in [37] where additional parameters on specific messages can trigger logic flaws.

Another useful use-case for abstract constructors/destructors is the possibility of adding parameters when they are not present. Let's take for example the `state` parameter of OAuth 2.0. A non compliant implementation could fail to check for the invariant relation between `state`s or completely avoid to use it. With custom constructors and destructors a monitor (that is proved correct only in presence

of the `state`) can add this parameter when the underlying implementation does not provide one.

By giving the TTP or the final user the possibility to specify these additional behaviors through different implementations of the constructors and destructors provide more flexibility in deployment. This flexibility makes the final user or the provider decide between different trade-offs instead of imposing them as part of the monitor design.

6 CASE STUDIES AND EVALUATION

We now show how we can generate security monitors for OAuth 2.0 and PayPal. For simplicity we will consider the Facebook IdP and the PayPal flow that is used by the osCommerce [1] integration, but with minor modifications we can apply the same techniques to different IdPs or integrations.

In this chapter the terms service provider SP and relaying party RP will be used interchangeably, as will trusted third party TTP and identity provider in the SSO setting.

6.1 OAUTH 2.0 MONITOR

Let's start by considering a typical integration of the explicit mode OAuth 2.0 of the Facebook IdP. Figure 12 shows an high-level specification in *Alice & Bob* notation.

$$
\begin{array}{lllll}
(1) & UA & \rightarrow & RP & : \ \textbf{Req}\ (GET, RP, loginpath, \emptyset) \\
(2) & RP & \rightarrow & UA & : \ \textbf{Resp}\ Ok(\texttt{pagewithlink}(\\
 & & & & \quad uri(HTTPS, IdP, oauthpath, \{appid, reduri, tcode, \textbf{new}\ state\}))) \\
(3) & UA & \rightarrow & IdP & : \ \textbf{Req}\ (GET, IdP, oauthpath, \{appid, reduri, tcode, state\}) \\
(4) & IdP & \rightarrow & UA & : \ \textbf{Resp}\ 302Redirect(uri(reduri, \{\textbf{new}\ code, state\})) \\
(5) & UA & \rightarrow & RP & : \ \textbf{Req}\ (GET, host, path, \{code, state\}) \\
(6) & RP & \rightarrow & IdP & : \ \textbf{Req}\ (GET, IdP, tokenpath, \{appid, reduri, appsecret, code\}) \\
(7) & IdP & \rightarrow & RP & : \ \textbf{Resp}\ Ok(Json\{\textbf{new}\ token, tokentype, expiration\}) \\
(8) & RP & \rightarrow & UA & : \ \textbf{Resp}\ Ok(Success())
\end{array}
$$

Figure 12: Facebook OAuth 2.0 explicit mode

The complete flow is composed by four round-trips:

1. The UA connects to the RP that generates a social login URL with a fresh *state* parameter (messages 1,2)
2. The UA clicks on the link and it is redirected to the IdP, where, if it is already logged in, it is redirected to *reduri* with a freshly generated *code* an the *state* that was present in the request (messages 3,4)

3. The RP receives the *code* and *state* parameters from the redirected UA. It exchanges the *code* with the *token* sending it to the IdP, together with its *appsecret*. It then shows the user a success page. (messages 5,8)
4. The IdP receives the token request containing the *code* and answers with a valid *token*.

Note that the last two round-trips are nested: the RP waits for the IdP to send a response before sending its own to the UA.

The different parties, as required by the OAuth 2.0 specification [20], needs to enforce some additional constraints on the values that are exchanged. In particular:

1. The RP must ensure that the *state* that it receives in message 5 is the same it generates in message 2.
2. The RP must ensure that the *state* and *code* parameters remain confidential.
3. The IdP must ensure that the *appid* is registered, the *appsecret* refers to the *appid* and each *token* issued for an *appid* has a corresponding *code* that was issued in a previous message.
4. The IdP must ensure that the *reduri* parameter that is used in message 3 is the same that is used in message 6.

The *state* parameter is recommended in [20] as a CSRF prevention mechanism. As we will see later, ProVerif is able to check this behavior and confirm that the *state* is essential for the security of the protocol.

6.1.1 *Security Policy*

The security policy for the protocol requires the UA to authenticate with both RP and IdP, RP to authenticate with IdP, and that *code* and *state* are kept confidential. Moreover, a successful login must be preceded by an explicit login request.

The following properties encode the security policy for OAuth 2.0. These properties are written as correspondences between security-relevant events and can be directly translated to ProVerif queries.

1. Each time time the UA reaches the success page, there have been an explicit login.
2. Each time the RP receives a *token* from the IdP for a session, it must have generated the login link for that session.
3. Each time the IdP generates and returns a *token* there have been a corresponding *code* request.

4. Each time the RP answers with a success page there have been a *code* request to the IdP.
5. Each time the UA reaches the success page there both have been a *code* request to the IdP and an explicit protocol start on the RP.
6. Each time the UA reaches the success page both the RP and the IdP needs to agree on the fact that they reached the end of the protocol.
7. The *code* and *state* parameters needs to be kept secret for the entire duration of the multi-party interaction.

6.1.2 *Applied pi-calculus Specification*

We now encode the protocol as ProVerif applied pi-calculus processes that use the WebSpi library. In particular we model each entity that takes part in the protocol as separate a $ServerApp$ or $UserAgent$ process (Defs. 5.1.1, 5.1.2).

```
let UA(b:Browser) =
  (let loginURI = uri(https(), integratorcom, loginpath(), nullParams()) in
   out(browserRequest(b), (loginURI, httpGet())))
|
  (in (newPage(b), (p1:Page,
        = uri(https(), integratorcom, loginpath(), nullParams()), pagewithlink(sso_uri)));
   let uri(= https(), idp, = oauthpath(), codereqparams(aid, reduri, state)) = sso_uri in
   event ua_begin(b, integratorcom, idp, aid, state);
   out(pageClick(b), (p1, sso_uri, httpGet())))
|
  (in(newPage(b), (p2:Page, reduri:Uri, = success()));
   let uri(= https(), = integratorcom, = callbackpath(), coderesparams(code, state1)) = reduri in
   event ua_end(b, integratorcom, facebookcom, state1, code)).
```

Figure 13: UA Process

The UA process, reported in Figure 13, models the behavior of the user that, through the browser, interacts with the other parties. This process has a single *user-initiated* action and two *page handlers*. In particular the user can either start the protocol by visiting the login page of the RP `integrator.com`, click the facebook login link when it receives it from the RP or receive a success page and terminate the protocol. The statements `event` $ua\_begin(...)$ and `event` $ua\_end(...)$ are part of the security specification and explicitly label security relevant events on which correspondence assertions or reachability queries can be defined (see Sec.2.3.1).

We model the RP as shown in Figure14. Here we have two parallel *handler processes* that refer to the two paths `loginpath` and `callbackpath`. When the RP receives a request on its login path (`loginpath`) it generates a facebook login uri and returns a page containing the link. To do this it needs to generate a

```
let RPApp(h:Host, fb:Host) =
 let reduri = uri(https(), h, callbackpath(), nullParams()) in
 (
    (in(httpServerRequest, (u:Uri, hs:Headers, = httpGet(), corr:bitstring));
     let uri(= https(), = h, = loginpath(), = nullParams()) = u in
     let cj = getCookie(hs) in
     let cp = session_start(cj, corr) in
     new state:bitstring;
     insert RPSessions(cp, state);
     let fb_uri = uri(https(), fb, oauthpath(), codereqparams(appid, reduri, state)) in
     event rp_begin(h, fb, cp, appid, reduri, state);
     out(httpServerResponse, (u, httpOk(pagewithlink(fb_uri)), cp, unsafeUrl(), corr)))
|
    (in(httpServerRequest, (u:Uri, hs:Headers, = httpGet(), corr:bitstring));
     let uri(= https(), = h, = callbackpath(), coderesparams(code, state)) = u in
     get RPSessions(= getCookie(hs), = state) in
     let req_uri = uri(https(), fb, tokenpath(), tokenreqparams(appid, reduri, appsecret, code)) in
     new ncorr:bitstring;
     out(httpServerRequest, (req_uri, headers(noneUri, nullCookiePair(), notajax()), httpGet(), ncorr));
     in(httpServerResponse, (
        = req_uri, httpOk(tokenresjson(token)), cp1:CookiePair, rp1:ReferrerPolicy, = ncorr));
     event rp_end(h, fb, cp, appid, reduri, appsecret, state, code, token);
     out(httpServerResponse, (u, httpOk(success()), cp, noReferrer(), corr)))).
```

Figure 14: RP Process

fresh *state* value and save it to its session storage (represented by a cookie and a table). When it receives a request on its callback path (`callbackpath`) it needs to check that the *state* parameter matches the one that is stored on its session storage. This check, as we mentioned before, is required to prevent CSRFs on this page: we can easily see that by removing this check our security policy cannot be satisfied. Moreover, ProVerif can find a counter example where the policy can be violated by an attack that is very similar to the *session swapping* of [29]. After this check, the RP makes an HTTP request to the IdP sending the received *code* and obtaining a *token*, ending the protocol by sending the UA a success page. The $noReferrer$ constructor models the `referrer-policy` header that needs to be set on this page by the RP to prevent leakage of secrets by the `referer` header. If we remove the header replacing it with its default value $unsafeUrl$, ProVerif instantly finds a secrecy violations similar to the *state leak* of [17].

Finally, we model the IdP process as shown in Figure 15. As with the RP, this process handles two paths. When it receives a request on `oauthpath` from the UA it redirects it to the redirection uri ($reduri$) specified in the request together with the *state* parameter and a freshly generated *code* that is saved into its database. When it receives a request on `tokenpath` with a *code* it first checks that the code was previously issued for that *appid* and then it checks if the redirect URI in the request is the same it stored when it created the code.

```
let IDPApp(h:Host) =
    (in(httpServerRequest, (u:Uri, hs:Headers, = httpGet(), corr:bitstring));
     let uri(= https(), = h, = oauthpath(), codereqparams(= appid, reduri, state)) = u in
     let uri(= https(), rh, rp, = nullParams()) = reduri in
     new code:bitstring;
     insert IDPAuthCodes(h, appid, reduri, code);
     let nuri = uri(https(), rh, rp, coderesparams(code, state)) in
     event idp_begin(h, appid, reduri, code);
     out(httpServerResponse, (u, httpRedirect(nuri), getCookie(hs), corr)))
|
    (in(httpServerRequest, (u:Uri, hs:Headers, = httpGet(), corr:bitstring));
     let uri(= https(), = h, = tokenpath(), tokenreqparams(= appid, reduri, = appsecret, code)) = u in
     get IDPAuthCodes(= h, = appid, = reduri, = code) in
     new token:bitstring;
     event idp_end(h, appid, appsecret, reduri, code, token);
     out(httpServerResponse, (u, httpOk(tokenresjson(token)), getCookie(hs), corr))).
```

Figure 15: IdP Process

This check is required by the specification to prevent the fact that an attacker can manipulate the redirect URI to get access to a valid authorization code. We can see that by removing this check our security policy cannot be satisfied and ProVerif shows a trace that is similar to the *Unauthorized Login by Auth. Code Redirection* attack of [5]. After this check a fresh *token* is generated and returned to the RP in a JSON object.

By encoding the security policy as correspondence assertions and running the three processes in parallel with the Dolev-Yao attacker and some predefined processes that model a web attacker, ProVerif can prove that all security properties hold. The complete model is reported in Appendix B.1.

6.1.3 *Monitor Generation*

Now that we have a verified protocol specification that encodes each entity, the invariants between the messages and a security policy P, we automatically generate run-time monitors for the SP and TTP parties that satisfy the security policy.

SP Monitor

We start from the RP process and generate the two monitors AM^{rp}_{RP} and AM^{sw}_{RP}. These two monitors are verified by executing ProVerif on the system in which the RP party in interoperable but does not enforce any invariant (RP_b). The result of the verification phase show that a `ServiceWorker` monitor is sufficient for the security properties in P to hold.

The AM^{sw}_{RP} `ServiceWorker` monitor is shown in Figure 16. The monitor

```
let RPServiceWorker() =
 let reduri = uri(https(), h, callbackpath(), nullParams()) in
 in(serviceWorkerFetch(b), (u:Uri, cs_1000, sw_ref:Uri, sw_p:Page, sw_aj:Ajax));
  let (= httpGet()) = cs_1000 in
  let (uri(= https(), = h, = loginpath(), = nullParams())) = u in
  (
    out(rawRequest(b), (u, cs_1000, sw_ref, sw_p, sw_aj));
    in(serviceWorkerResult(b), (= u, cs_1100:HttpResponse, cs_1101:ReferrerPolicy,
                                sw_foo:XDR, sw_corr:bitstring));
    let (httpOk(pagewithlink(uri(= https(), = fb, = oauthpath(),
                           codereqparams(= appid, = reduri, state))))) = cs_1100 in
    insert MRPSessions(state);
    out(serviceWorkerSendHttpResponse(b), (u, cs_1100, cs_1101, sw_foo, sw_corr)))
 else let (uri(= https(), = h, = callbackpath(), coderesparams(code, state))) = u in
  (
    get MRPSessions(= state) in
    out(rawRequest(b), (u, cs_1000, sw_ref, sw_p, sw_aj));
    in(serviceWorkerResult(b), (= u, cs_1200:HttpResponse, cs_1201:ReferrerPolicy,
                                sw_foo:XDR, sw_corr:bitstring));
    let (httpOk(success())) = cs_1200 in
    out(serviceWorkerSendHttpResponse(b), (u, cs_1200, noReferrer(), sw_foo, sw_corr))).
```

Figure 16: RP ServiceWorker Process

receives fetch events from the $serviceWorkerFetch$ channel, it pattern-matches the request object to verify that the incoming request is a `GET` request and then branches on the URL. Each branch represents a *handler process* where the a single path of the application is considered. In the login path `loginpath` the ServiceWorker fetches the URL, it checks that the returned page contains a login URL and saves the state parameter. The final response that is forwarded to the browser is the result of the fetch call. We can see that the `ServiceWorker` differs from the $RPApp$ process in that it does not need to handle multiple sessions. Here, the cookie handling code is not present and the session data table is rewritten as a single-entry table. In the callback path the worker process checks that the state that is present in the request is the same that was stored in the $MRPSessions$ table. If it is the case, the process fetches the page and forwards it to the browser. If it is not the case the process terminates and the connection is closed.

This code, as we described in section 5.4, can be directly translated to `JavaScript` to be run by the browser. The translation is a one-to-one mapping between the pi-calculus process and the *JavaScript* implementation, so it is trivial to show that the real monitor performs the same operations of the abstract process. This, together with the ProVerif-provided proof that the abstract `ServiceWorker` satisfies P, allows us to say that the generated real monitor satisfies the same properties P.

The generated `JavaScript` code can be found in Appendix B.1.2. This monitor needs to be configured to be deployed on a real SP. The configuration defines the values of the symbols and constructors that are used throughout the code. An example configuration is shown in Figure 17. Here we define the actual values for the paths of the web applications, the two host-names, the *appid* and the $MRPSessions$ table.

```
1 const h = "nonexistentwebsite.lol"
2 const fb = "www.facebook.com"
3 const loginpath = "/login.php"
4 const callbackpath = "/fb-callback.php"
5 const oauthpath = "/v3.2/dialog/oauth"
6 const appid = "758951591140140"
7
8 let db = new zango.Db('SW', { MRPSessions: ['col_1'] })
9 let MRPSessions = db.collection('MRPSessions')
```

Figure 17: ServiceWorker Monitor Configuration

TTP Monitor

Let us now consider the TTP. When we apply the monitor verification phase to the two generated monitors AM^{sw}_{TTP} and AM^{rp}_{TTP}, the only configuration that satisfies the security policy P is the one where the TTP monitor is a reverse proxy. This is the case because the IdP does not interact only with the user browser, but it also receives the auth. code from the RP on the *back channel*. A ServiceWorker would only be able to inspect the requests that are made in the front channel using a real web browser, while a reverse proxy can inspect all requests that reach the server. In general, when some constraints needs to be enforced between different channels, a reverse proxy is needed. For example the constraint that the *reduri* parameter must be the same in both the code request and the token request (respectively `oauthpath` and `tokenpath`), can only be enforced by a reverse proxy, since the first request is sent by the user browser and the second by the SP.

The AM^{rp}_{TTP} reverse proxy monitor is shown in Figure 18. Here the structure of the monitor process matches the one of the $IDPApp$ process from which it is generated. Each *handler process* refers to a single path, it receives an HTTP request and it send it to the monitored application using the `mC_1_out` channel. The response from the application is received from the `mC_1_in` channel and forwarded to the client. The monitor process uses the $MIDPAuthCodes$ table to save the issued auth. codes when the monitored IdP generates a new code. It then checks the validity of the code it receives before forwarding the request to the server, that issues the token. If the *code* does not match a previously issued

```
let IDPMonitor(h:Host) =
 (in(httpServerRequest, (cs_1000:Uri, hs:Headers, = httpGet(), corr:bitstring));
  let (uri(= https(), = h, = oauthpath(),
       codereqparams(appid, uri(= https(), rh, rp, = nullParams()), state))) = cs_1000 in
  out(mC_1_out, (cs_1000, hs, httpGet(), corr));
  in(mC_1_in, (cs_1103:Uri, cs_1100:HttpResponse,
            cs_1101:CookiePair, cs_1102:ReferrerPolicy, = corr));
  let (httpRedirect(uri(= https(), = rh, = rp, coderesparams(code, = state)))) = cs_1100 in
  let (= getCookie(hs)) = cs_1101 in
  let (= uri(https(), h, oauthpath(),
       codereqparams(appid, uri(https(), rh, rp, nullParams()), state))) = cs_1103 in
  insert MIDPAuthCodes(h, appid, uri(https(), rh, rp, nullParams()), code);
  out(httpServerResponse, (cs_1103, cs_1100, cs_1101, cs_1102, corr)))
|
 (in(httpServerRequest, (cs_1000:Uri, hs:Headers, = httpGet(), corr:bitstring));
  let (uri(= https(), = h, = tokenpath(),
       tokenreqparams(appid, reduri, appsecret, code))) = cs_1000 in
  get MIDPAuthCodes(= h, = appid, = reduri, = code) in
  out(mC_1_out, (cs_1000, hs, httpGet(), corr));
  in(mC_1_in, (cs_1203:Uri, cs_1200:HttpResponse,
            cs_1201:CookiePair, cs_1202:ReferrerPolicy, = corr));
  let (httpOk(tokenresjson(token))) = cs_1200 in
  let (= getCookie(hs)) = cs_1201 in
  let (= uri(https(), h, tokenpath(),
       tokenreqparams(appid, reduri, appsecret, code))) = cs_1203 in
  out(httpServerResponse, (cs_1203, cs_1200, cs_1201, cs_1202, corr))).
```

Figure 18: IdP Monitor Process

code or the *reduri* parameter does not match the one that was used to issue the code, the process terminates and the connection is closed.

As for the `ServiceWorker` this monitor can be directly translated to a formally-verified `python` plug-in for mitmproxy. The resulting real monitor does not need to be joined by a forward proxy as the TTP does not initiate connections to the other entities.

6.2 PAYPAL E-COMMERCE PLATFORM MONITOR

Let us consider the integration of the PayPal Payment Standard API into the open-source OSCommerce [1] e-shop application. Figure 19 shows the high-level specification of the protocol in Alice & Bob notation.

The complete flow of the protocol is composed of 5 round-trips.

1. The UA connects to the RP to initiate the checout procedure. The RP fills a form with the checkout information to be sent to the TTP and presents the user a button that submits this form to PayPal. (Messages 1, 2)

$$
\begin{array}{lllll}
(1) & UA & \rightarrow & RP & : \ \textbf{Req}\ (GET, RP, checkoutpath, \emptyset) \\
(2) & RP & \rightarrow & UA & : \ \textbf{Resp}\ Ok(\texttt{filledForm}(payUri, \\
 & & & & \quad \{merchantId, amount, \textbf{new}\ invoiceId, notUri, retUri\})) \\
(3) & UA & \rightarrow & TTP & : \ \textbf{Req}\ (POST, TTP, payUri, \\
 & & & & \quad \{merchantId, amount, invoiceId, notUri, retUri\}) \\
(4) & TTP & \rightarrow & UA & : \ \textbf{Resp}\ Ok(JS(returnTo = retUri)) \\
(5) & UA & \rightarrow & RP & : \ \textbf{Req}\ (GET, retUri) \\
(6) & RP & \rightarrow & UA & : \ \textbf{Resp}\ Ok(Processing()) \\
 & & & & \\
(7) & TTP & \rightarrow & RP & : \ \textbf{Req}\ (POST, RP, ipnpath, \\
 & & & & \quad \{merchantId, amount, invoiceId, \textbf{new}\ payerId, \textbf{new}\ verisign\}) \\
(8) & RP & \rightarrow & TTP & : \ \textbf{Req}\ (POST, RP, webscrpath, \\
 & & & & \quad \{merchantId, amount, invoiceId, payerId, verisign\}) \\
(9) & TTP & \rightarrow & RP & : \ \textbf{Resp}\ Ok(Verified()) \\
(10) & RP & \rightarrow & TTP & : \ \textbf{Resp}\ Ok(Acknowledged())
\end{array}
$$

Figure 19: OSCommerce PayPal Standard Integration

2. The UA clicks on the button and it is redirected to the TTP, where, if it is already logged in, it is asked to pay the selected amount. After the payment the user is redirected back to the RP using some `JavaScript` code. (Messages 3, 4)
3. The user is redirected to the RP that changes the state of the current invoice to *processing*. (Messages 5, 6)
4. The TTP sends the RP an IPN notification to its IPN path. The message contains the merchant id to which the payment was made, the payed amount, the invoice id, a payer id and a signature. The RP responds with a 200 status code to acknowledge the notification. (Messages 7, 10)
5. The RP verifies the IPN data by sending it back to the TTP, that verifies the signatures and, in case everything is correct, responds with a positive answer. (Messages 8, 9)

Note that the last two round-trips are nested: the RP waits for the IdP response before sending it its acknowledgment.

The different parties need to enforce the following constraints:

1. The integrator website must ensure that the merchant identifier which received the payment is the one corresponding to the shop owner.
2. The integrator website must ensure that the amount that is actually being payed by the user is the same that has been filled in the form in message 2.

3. The integrator website must ensure that the IPN information is correct. This is done by verifying it with the TTP.
4. The TTP must ensure that the IPN data it receives from the RP during verification is valid and corresponds to the data that was sent in the IPN notification.

The security policy for the protocol requires that for each completed transaction both the RP and the TTP agree on the transaction information. The following properties encode the security policy for the PayPal standard intergation:

1. When an IPN notification signaling the end of a transaction is verified correctly by the RP, a PayPal checkout procedure needs to be completed at the TTP. Moreover, both the RP and TTP must agree on the fact that the transaction is verified and on the values of the amount, invoice id and payer id.
2. Each time an IPN message is verified by the RP, there have been an explicit transaction start (checkout) at RP
3. Each time an IPN message is verified by the RP, there have been a successful checkout at TTP.

These properties can be directly encoded as ProVerif queries.

By encoding the protocol actors as $ServerApp$ and $UserAgent$ processes and translating the security policy into correspondence assertions, ProVerif can prove that in presence of a Dolev-Yao and a web attacker, all the security properties hold. The complete applied pi-calculus ProVerif model can be found in Appendix B.2.

A verified run-time monitor can be generated for the integrator website RP. As we saw with the monitor for the OAuth 2.0 IdP, this monitor cannot be implemented with a `ServiceWorker`, so we need to generate a reverse proxy. This proxy needs to also consider *back channels* to be able to check that the information that is transmitted by the IPN notification is the same that was generated by the RP.

6.3 EVALUATION

To evaluate the effectiveness of the generated monitors we selected a pool of vulnerable applications that use the OAuth 2.0 protocol or that integrate the PayPal payment system. Our evaluation was performed on vulnerable integration of those protocols, where the vulnerability was known and the attack could be tested. To this end we developed some vulnerable web applications and choose some open source projects with known integration vulnerabilities.

In particular, we tested the OAuth 2.0 monitors on three variants of a specifically designed web application that allows, through a plug-in system, to integrate different SSO providers and introduce vulnerabilities in those integrations.

Two of the three versions integrate the Facebook OAuth 2.0 API through its PHP SDK and are vulnerable to, respectively, the *session swapping* attack of [29] and the *state leak* of [17]. The third version is a correct integration of a custom identity provider which is vulnerable to the *auth token redirection* attack of [5].

We tested the PayPal monitors with version 2.3.1 of the osCommerce e-shop platform, which is vulnerable to the *shop-for-free* and *shop-for-less* vulnerabilities [25] and version 1.60 of the NopCommerce e-shop platform which is vulnerable to a *shop-for-less* vulnerability [35].

While performing some testing on the web we also discovered a public website (`overleaf.com`) that is vulnerable to a variant of the *session swapping* attack of [29] on its Google OAuth 2.0 integration.

6.3.1 *Results*

We ran our vulnerable OAuth application in parallel with the generated monitors. We generated a `ServiceWorker` monitor for the integartor SP and a reverse proxy monitor for the TTP website when testing our vulnerable TTP implementation. All three attacks are successfully mitigated when the monitors are active, while they can be exploited when the monitors are turned off.

In the same way all vulnerabilities of the e-commerce platforms are mitigated by the generated monitors. The monitors for osCommerce and NopCommerce are both reverse/forward proxies that inspect the *back channels*. We can test that a `ServiceWorker` is not able to mitigate these vulnerabilities by generating a `ServiceWorker` and running the exploit against the integrations. In both cases the `ServiceWorker` proved insufficient to secure the protocol as the verification phase of the monitor generation suggested.

The tests on the `overleaf.com` website proved successful only after a modification to the monitor. It was the case that the integration could not be monitored by a `ServiceWorker` as the check on the state parameter could not be executed. This was because the state parameter was sent to the UA inside a cross-origin redirect (see Sec. 4.1.3). By adding an additional handler process to the RP protocol specification that encodes this behavior and also applying the *strict referrer validation* in the initial protocol specification, the new generated `ServiceWorker` monitor was able to mitigate the vulnerability by only looking at the referrer header. This highlight how the initial protocol specification needs to be general enough to cover all possibilities of integration.

6.3.2 *Future Evaluation Plan*

The evaluation of these monitors shows promising results by being able to successfully mitigate all the known vulnerabilities. However, we discovered that different integrations of the same TTP might be implemented in slightly different ways. The protocol specification for that TTP needs to consider these variations. For this reason further testing needs to be done for evaluating the compatibility of the generated monitors and how we can encode the integration differences in a general way.

We plan to select the top 5 OAuth 2.0 IdP and test the compatibility of the monitors executed on the first 10000 websites of the Alexa list that integrate these IdPs. This will give us a measure of the difference in integration in the real world and how we can better generalize the monitors.

From the point of view of the TTP, however, these differences might only be related to an under-specification of the integration mechanism. If the TTP requires a single integration mechanism and provides an SDK that enforce this mechanism, these differences might become irrelevant as every website integrates the protocol in the same standard way.

We plan to apply the same evaluation strategy to other vulnerable integration of the PayPal payment system, such as the TomatoCart and OpenCart projects [25] and to another major payment system, for example Stripe.

7

CONCLUSIONS

This thesis explored the design challenges of a runtime monitor for web protocols. In particular, we identified three different deployment options and showed that it it possible to mitigate all the attacks that can be found in the literature on SSO and CaaS integrations by server-side deployed monitoring. We introduced ServiceWorkers as a security mechanism and discussed their applicability to runtime monitoring of multi-party web applications.

With this knowledge, we designed a black-box approach to generate formally-verified monitors for web protocols that takes as input a protocol specification and outputs reverse proxy or ServiceWorker monitors. We used the ProVerif protocol verifier and a principled approach to transfer the security properties of the initial specification to a monitored system. This way, these monitors are guaranteed to satisfy the security policy that was defined during the protocol specification, even when paired with non-compliant or broken implementation. We implemented this monitor generation pipeline as a stand-alone tool that is able to generate verified monitor from a ProVerif specification of web protocols.

We evaluated the effectiveness of the generated monitors by testing different known vulnerable applications. These tests showed promising results as the generated monitors are able to mitigate all the vulnerabilities of these applications.

7.1 LIMITATIONS AND FUTURE WORK

In the current prototype of our tool, the TTP is required to manually write the specification of the web protocol by first describing the behavior of the involved entities, then by providing a security policy. Even if we restrict the definition of these processes to $ServerApp$ and $UserAgent$ processes, the TTP is still required to write a complete ProVerif script. Moreover, we need it to encode all the possible variations for implementing the integration.

This task can be simplified by providing a specification language that hides low-level details and generalizations. This language might enable the TTP to write the protocol specification declaratively, as we did in Chapter 6 with the *Alice & Bob* notation. A preprocessor that translates this declarative language to the ProVerif processes can be easily implemented, since the pi-calculus processes written using the WebSpi library contain a lot of boilerplate code.

Another approach that might be used to reduce the effort needed for the protocol specification could be the application of a model extraction technique. An automatic model extraction phase, however, could lead to the creation of false invariants or incorrect models. An interesting future work would be to devise a semi-automatic method to extract and manually refine the protocol specification. This can assist the provider during the protocol specification phase, while minimizing the possibility of introducing modeling errors. A starting point for this development could be the AUTHSCAN [4] tool, that automatically extracts ProVerif models from web applications.

Another interesting research direction is the study of the interaction between multiple monitors. Multiple communicating monitors could provide more security guarantees than independent ones, however, a safe communication channel between them needs to be established. In chapter 4 we showed that all the attacks that can be found in the literature can be mitigated by only using independent monitors. However, ServiceWorkers cannot always be used unless the TTP adds a CORS header to, for example, its login page. An interesting use of multiple communicating monitors could be to enable the CORS header on the TTP only when needed by the ServiceWorker.

Finally, one of the main limitations of all the approaches that deal only with the HTTP(S) request and response pairs of web protocols is their inability to inspect the `JavaScript` code. Some modern client-side SDKs, like the new Facebook Login SDK, are beginning to optionally use the `postMessage` browser API instead of relying on URL fragments. A future work direction is to investigate how these APIs can be monitored by browser extensions or by `ServiceWorkers`.

A

ATTACKS ON OAUTH 2.0 AND PAYPAL

A.1 ATTACKS ON OAUTH 2.0

SESSION SWAPPING / SOCIAL LOGIN CSRF / TOKEN REPLAY [29, 32, 5, 28, 20]

Session swapping and some variants of CSRF exploit the lack of contextual binding between the login endpoint and the callback endpoint. This is often the case in clients that do not provide a *state* parameter or do not strictly validate it. The *state* parameter is typically a value that is bound to the user session that is sent to the client when redirecting back the user from a successful login on the IdP. Usually these attack start with the attacker signing in to the IdP and obtaining either a *code* or an access *token*. An honest user is then forced, through CSRF, to send the attacker SSO credential to the RP, which makes the honest user log in on RP with the attacker identity at IdP.

IDP MIX-UP ATTACK [18]

The IdP Mix-Up attack confuses the RP about which IdP the user chose at the beginning of the login process. This attack requires the RP to implement explicit user intention tracking, that is, storing the user intention to login to a particular IdP in its session. The attack has two variants, the first that requires a network attacker and applies in the cases in which the first protocol message is sent through an HTTP connection; the second that requires only a web attacker and applies in the cases in which HTTPS is used in every protocol message. The HTTP variant of the attack can be mounted as follows: The honest user select to log in on RP using the honest IdP, HIdP. The attacker intercepts the request directed to the RP and and replaces HIdP with a malicious IdP, AIdP, and then modifies the response of RP containing the redirect to AIdP to contain a redirect to HIdP (together with RP `client_id` at HIdP). The user authenticates with HIdP and is redirected to RP that thinks that the user was authenticating with AIdP, so it sends the obtained *code* to AIdP, leaking it to the attacker. In the HTTPS variant, the user wants to login with AIdP, which redirects him to login at HIdP. The attack then proceeds as in the HTTP variant with the attacker receiving the HIdP issued *code* at AIdP.

CODE/TOKEN REDIRECTION ATTACKS [5, 20]

Token and code redirection attacks exploit the lack of strict validation of the `redirect_uri` parameter and involve its manipulation by the attacker. These

attacks can be applied even if the IdP does lax (wildcard) validation of the redirection uri in case the client contains an open redirector. The attack can be mounted as follows: the honest user starts the SSO flow with a `redirect_uri` that has been manipulated to redirect him to an attacker controlled website. The user authenticates with the IdP and is redirected to the attacker, which receives the SSO credential. In explicit mode, the received *code* can be sent by the attacker to the honest RP redirection uri to obtain the user *token*. In implicit mode the *token* is directly sent to the attacker during the redirect.

CODE/STATE LEAKAGE [17, 18]

In explicit mode, after the user has authenticated with the IdP, he is redirected to the RP with the *code* and the *state* as GET parameters. If the response of this request is a page that contains a resource hosted by an attacker controlled website, when the user browser fetches the resource, it sends the full URI of the current page in the Referrer header. This way both *code* and *state* are leaked to the attacker. The same scenario applies if the RP callback page contains a link to an attacker's website, since the Referrer is sent on page clicks.

ACCESS TOKEN EAVESDROPPING [29]

This attack consist in eavesdropping the access token by sniffing on the unencrypted communication between the browser and the RP.

NAIVE SESSION INTEGRITY ATTACK [18]

The attack breaks the session integrity property for RPs that integrate multiple IdPs and use naïve user intention tracking, that is, have a different `redirect_uri` for each IdP. The attack can be mounted as follows: The attacker starts a session with an honest IdP, HIdP, and obtains a *code* or *token*. When a honest user want to login with AIdP (a malicious IdP), AIdP redirects him to the redirection uri of HIdP with the attacker SSO credential and the state generated by RP. The RP then believes that the user logged in at HIdP since it uses naïve user intention tracking. So the user is logged is as the attacker at RP. The attack can be prevented by either using the explicit user intention tracking or by using a different state parameter for each IdP.

CROSS SOCIAL-NETWORK REQUEST FORGERY [5]

If an RP supports social log in with multiple IdPs but uses the same endpoint for all of them (JanRain, GigYa), a malicious IdP can confuse the RP about which IdP the user wants to login with. This attack is similar to the IdP Mix-Up of [18] but it applies only when the website uses naïve user intention tracking (multiple `redirect_uri`) instead of the explicit tracking (saving the user intention in the session) required by the mix-up attack to work. A malicious IdP, AIdP, could redirect the user to an honest IdP, HIdP, with its own `redirect_uri`. When the RP receives the *code* at AIdP redirection uri assumes that it comes from the

malicious IdP, so it is sent to AIdP to obtain a *token*. This way the *code* is leaked to the attacker.

FACEBOOK IMPLICIT APPID SPOOFING [31, 28]

A malicious SP can spoof the `client_id` value of the honest SP in the facebook implicit flow to obtain a valid *token* of the victim. This attack can be mitigated by whitelisting the `redirect_uri` for each `client_id`.

FORCE/AUTOMATIC LOGIN CSRF [5]

If the login form at RP has no CSRF protection, a malicious website can redirect the user to the login request and force the beginning of an SSO flow. If the IdP authorized silently the user because he already logged in at IdP and gave permission to the RP app, the protocol completes without any interaction from the user. This way an attacker is able to force a user to log in at RP even if he did not wish to. The attack is mitigated by implementing a CSRF protection on the client or by requiring explicit user consent each time a token is requested.

OPEN REDIRECTOR IN OAUTH 2.0 [23, 20]

If the IdP allows the SP to register only part of the `redirect_uri` parameter, an attacker can use an open redirector on SP to create a redirection rui that can pass the validation and obtain the SSO credential of the victim.

Moreover an attacker can use the `redirect_uri` parameter to abuse the IdP as an open redirector if it does not validate the parameter or redirects the user on failure.

SOCIAL LOGIN CSRF THROUGH AS LOGIN CSRF [5]

A malicious website can bypass the login step at IdP by sending its own credentials if the authorization server is not protected against login CSRF. If the user subsequently clicks on a social login button on RP, he will be logged into RP as the attacker

307 REDIRECT ATTACK [18]

A malicious RP can get the honest user credentials at IdP if IdP uses the wrong HTTP redirection status code. If the IdP that is used for the login uses the 307 redirect status code and it redirects the user immediately after the user entered his credentials, those credentials can be sent to the attacker in the body of the redirect response.

TOKEN/CODE THEFT VIA XSS [29]

An attacker can inject a malicious script into any page of an RP to initiate an implicit mode login flow at IdP and obtain the authorization *token*. This attack is possible only if the user is already logged in at IdP and have already given permissions to the RP app, so that the implicit flow does not require any user intervention.

A.2 ATTACKS ON PAYPAL

SP$_M$ *payeeid* REPLAY IN SP$_T$ [28, 25]
When redirected to PayPal for the checkout process an attacker can replace the seller PayPal account with another account under his control, paying himself for the product he was buying on the merchant website. If the SP website does not check the *PayeeId* the payment was made to during the IPN flow with PayPal, it marks the transaction as completed and verified.
T$_1$ AT SP$_T$ *token* REPLAY IN T$_2$ AT SP$_T$ [28, 25]
During the PayPal Express flow, an attacker can replay the $PayerId$ and $Token$ parameters, that are sent to the SP when PayPal redirects the user back to the SP, to complete any successive transaction. The attack can be mounted as follows: The attacker first buys a cheap item in the SP store, capturing the $PayerId$ and $Token$ parameters of at the redirect URI. It the logs in again adding an expensive item to the cart and visiting the redirection uri replaying the previously saved values. This way the SP is convinced that the transaction successfully happened at TTP and marks the transaction as completed.
nopcommerce GROSS CHANGE IN PDT FLOW AND IPN CALLBACK [30]
During the PayPal Standard PDT flow, an attacker can arbitrarily change the value of the *gross* field in the message directed to PayPal and shop for less. If the SP website does not check that the *gross* field that is returned on the back channel during the last message of the PDT flow corresponds to the one that it expects, an attacker can pay an arbitrary amount of money for each item in the shop. The attack can be mounted as follows: the attacker modifies the value of the *gross* field before being redirected to PayPal, where he pays the amount defined in the modified *gross* value. The attacker is the redirected to the SP website with a transaction id. The SP website sends the transaction id to PayPal receiving the transaction data and marking the transaction as completed.

B

MODELS AND CODE LISTINGS

B.1 OAUTH 2.0 - FACEBOOK

B.1.1 *ProVerif Model*

```
1
2  fun loginpath(): Path[data]. (* = "/login" *)
3  fun oauthpath(): Path[data]. (* = "/v[0-9]\.[0-9]/dialog/oauth" *)
4  fun callbackpath(): Path[data]. (* = "/callback" *)
5  fun tokenpath(): Path[data]. (* = /v[0-9]\.[0-9]/oauth/access_token
       *)
6
7  fun codereqparams(Id, Uri, bitstring): Params[data]. (* = {"
       client_id", "redirect_uri", "state"} *)
8  fun coderesparams(bitstring, bitstring): Params[data]. (* = {"code
       ", "state"} *)
9  fun tokenreqparams(Id, Uri, Secret, bitstring): Params[data]. (* =
       {"client_id", "redirect_uri", "client_secret", "code"} *)
10
11 fun success(): bitstring[data].
12 fun tokenresjson(bitstring): bitstring[data].
13 fun pagewithlink(Uri): bitstring[data].
14
15 const appid:Id.
16 const appsecret:Secret [private].
17
18 free facebookcom: Host.
19 free integratorcom: Host.
20
21 fun mkCookie(bitstring): Cookie [private].
22
23 letfun session_start(cj:CookiePair, corr:bitstring) =
24   let cookiePair(session_cookie,path_cookie) = cj in
25   if session_cookie <> nullCookie() then
26     cj
27   else
28     cookiePair(mkCookie(corr), nullCookie()).
29
30 let mkserver(h:Host) =
31    (new sk:privkey;
32      let pubk = pk(sk) in
33      insert serverIdentities(orig(https(),h), pubk,sk,xdr());
```

```
34      out(pub,pubk)).
35
36  (* Setup a malicious server *)
37  let MaliciousServer() = mkserver(mallory).
38
39  (* App that leaks to the attacker everything it receives *)
40  let AttackerLeakApp(h:Host) =
41    !(in(httpServerRequest,(u:Uri,hs:Headers,r:HttpRequest,corr:
          bitstring));
42      let uri(=https(), =h, p, q) = u in
43      out(pub, (u, hs, r))).
44
45  (* events *)
46  event rp_begin(Host, Host, CookiePair, Id, Uri, bitstring).
47  event rp_end(Host, Host, CookiePair,Id, Uri, Secret, bitstring,
        bitstring, bitstring).
48  event idp_begin(Host, Id, Uri, bitstring).
49  event idp_end(Host, Id, Secret, Uri, bitstring, bitstring).
50  event ua_begin(Browser, Host, Host, Id, bitstring).
51  event ua_end(Browser, Host, Host, bitstring, bitstring).
52
53  (* reachability queries: every event must be reachable *)
54  query h:Host,idph:Host, reduri:Uri, b:Browser, id:Id, c:CookiePair,
         sec:Secret,
55        state:bitstring, code:bitstring, token:bitstring;
56     event( rp_begin(h,idph,c,id,reduri,state) );
57     event( rp_end(h,idph,c,id,reduri,sec,state,code,token) );
58     event( idp_begin(idph,id,reduri,code) );
59     event( idp_end(idph,id,sec,reduri,code,token) );
60     event( ua_begin(b,h, idph, id, state) );
61     event( ua_end(b, h,idph,state, code) );
62     event( rp_end(h,idph,c,id,reduri,sec,state,code,token) ) &&
63     event( idp_end(idph,id,sec,reduri,code,token) ) &&
64     event( ua_end(b, h,idph,state, code) ).
65
66  (* correspondence assertions *)
67  query h:Host, idph:Host, c:CookiePair, aid:Id, sec:Secret,reduri:
        Uri, state:bitstring,code:bitstring, token:bitstring,
68        b:Browser;
69        event( ua_end(b, h, idph, state, code) ) &&
70        event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) ) ==>
              event( rp_begin(h,idph,c,aid,reduri,state) ).
71
72  query h:Host, reduri:Uri, aid:Id, sec:Secret, code:bitstring, state
        :bitstring, token:bitstring;
73        event( idp_end(h,aid,sec,reduri,code,token) ) ==> event(
              idp_begin(h,aid,reduri,code) ).
74
```

```
75 query b:Browser, h:Host, idph:Host, id:Id, state:bitstring, code:
       bitstring;
76       event( ua_end(b, h,idph, state, code) ) ==> event( ua_begin(b
             , h,idph, id, state) ).
77
78 query h:Host, idph:Host, c:CookiePair, code:bitstring, aid:Id, sec:
       Secret,reduri:Uri, state:bitstring, token:bitstring,
79       b:Browser;
80       event( ua_end(b, h, idph, state, code) ) &&
81       event( idp_end(idph,aid,sec,reduri,code,token) ) ==> event(
             rp_begin(h,idph,c,aid,reduri,state) ).
82
83 query h:Host, idph:Host, c:CookiePair, aid:Id, sec:Secret, code:
       bitstring, state:bitstring, token:bitstring,reduri:Uri,
84       b:Browser;
85       event( ua_end(b, h, idph, state, code) ) &&
86       event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) ) ==>
              event( idp_begin(idph,aid,reduri,code) ).
87
88 query h:Host, idph:Host, b:Browser, c:CookiePair, aid:Id, code:
       bitstring, state:bitstring, token:bitstring, reduri:Uri;
89       event( ua_end(b, h, idph, state, code) )
90       ==> event( idp_begin(idph,aid,reduri,code) ) && event(
             rp_begin(h,idph,c,aid,reduri,state) ).
91
92 query h:Host,idph:Host, b:Browser, c:CookiePair,aid:Id, sec:Secret,
        code:bitstring, state:bitstring, token:bitstring, reduri:Uri;
93       event( ua_end(b, h, idph, state, code) )
94       ==> event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) )
95       && event( idp_end(idph,aid,sec,reduri, code,token) ).
96
97 (* Secrecy of the tokens and states *)
98 query code:bitstring, token:bitstring, b:Browser, h:Host, idph:Host
       , aid:Id, sec:Secret, c:CookiePair, state:bitstring, reduri:Uri
       ;
99       event( ua_end(b, h,idph, state, code) )
100   && event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) )
101   && event( idp_end(idph,aid,sec,reduri, code, token) )
102   && attacker(code);
103      event( ua_end(b, h,idph,state, code) )
104   && event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) )
105   && event( idp_end(idph,aid,sec,reduri, code, token) )
106   && attacker(token);
107      event( ua_end(b, h,idph,state, code) )
108   && event( rp_end(h,idph,c,aid,reduri,sec,state,code,token) )
109   && event( idp_end(idph,aid,sec,reduri, code, token) )
110   && attacker(state).
111
112
```

```
113 table RPSessions(CookiePair, bitstring).
114
115 let RPApp(h:Host, fb:Host) =
116  let reduri = uri(https(), h, callbackpath(), nullParams()) in
117   (
118     (in(httpServerRequest,(u:Uri,hs:Headers,=httpGet(),corr:
        bitstring));
119      let uri(=https(), =h, =loginpath(), =nullParams()) = u in
120      let cj = getCookie(hs) in
121      let cp = session_start(cj, corr) in
122      new state:bitstring;
123      insert RPSessions(cp,state);
124      let fb_uri = uri(https(), fb, oauthpath(), codereqparams(appid
         , reduri, state)) in
125      event rp_begin(h, fb, cp, appid, reduri, state);
126      out(httpServerResponse, (u, httpOk(pagewithlink(fb_uri)), cp,
         unsafeUrl(), corr)))
127   |
128     (in(httpServerRequest,(u:Uri,hs:Headers,=httpGet(),corr:
        bitstring));
129      let uri(=https(), =h, =callbackpath(), coderesparams(code,
         state)) = u in
130      get RPSessions(=getCookie(hs), =state) in
131      let cp = getCookie(hs) in
132      let req_uri = uri(https(), fb, tokenpath(), tokenreqparams(
         appid, reduri, appsecret, code)) in
133      new ncorr:bitstring;
134      out(httpServerRequest,
135           (req_uri,headers(noneUri, nullCookiePair(), notajax()),
              httpGet(), ncorr));
136      in(httpServerResponse, (=req_uri, httpOk(tokenresjson(token)),
          cp1:CookiePair, rp1:ReferrerPolicy, =ncorr));
137      event rp_end(h, fb, cp, appid, reduri, appsecret, state, code,
          token);
138      out(httpServerResponse, (u, httpOk(success()), cp, noReferrer
         (), corr)))).
139
140
141 table IDPAuthCodes(Host,Id, Uri, bitstring).
142
143 let IDPApp(h:Host) =
144 (
145     (in(httpServerRequest,(u:Uri,hs:Headers,=httpGet(),corr:
        bitstring));
146      let uri(=https(), =h, =oauthpath(), codereqparams(=appid,
         reduri, state)) = u in
147      let uri(=https(), rh, rp, =nullParams()) = reduri in
148      new code:bitstring;
149      insert IDPAuthCodes(h,appid, reduri, code);
```

```
150       let nuri = uri(https(), rh, rp, coderesparams(code, state)) in
151       event idp_begin(h, appid, reduri, code);
152       out(httpServerResponse, (u, httpRedirect(nuri), getCookie(hs),
              unsafeUrl(), corr)))
153   |
154       (in(httpServerRequest,(u:Uri,hs:Headers,=httpGet(),corr:
              bitstring));
155       let uri(=https(), =h, =tokenpath(), tokenreqparams(=appid,
              reduri, =appsecret, code)) = u in
156       get IDPAuthCodes(=h,=appid, =reduri, =code) in
157       new token:bitstring;
158       event idp_end(h, appid, appsecret, reduri, code, token);
159       out(httpServerResponse, (u, httpOk(tokenresjson(token)),
              getCookie(hs), unsafeUrl(), corr)))).
160
161
162   let UA(b:Browser) =
163     (let loginURI = uri(https(), integratorcom, loginpath(),
          nullParams()) in
164      out(browserRequest(b), (loginURI, httpGet())))
165   |
166     (in (newPage(b),(p1:Page,=uri(https(), integratorcom, loginpath()
          , nullParams()),pagewithlink(sso_uri)));
167      let uri(=https(), idp, =oauthpath(), codereqparams(aid, red_uri,
           state)) = sso_uri in
168      event ua_begin(b, integratorcom, idp, aid, state);
169      out(pageClick(b), (p1, sso_uri, httpGet())))
170   |
171     (in(newPage(b),(p2:Page, reduri:Uri, =success()));
172      let uri(=https(), =integratorcom, =callbackpath(), coderesparams
           (code,state1)) = reduri in
173      event ua_end(b, integratorcom, facebookcom, state1, code)).
174
175
176   let OAuth() =
177       (
178         mkserver(facebookcom) | mkserver(integratorcom)
179       | (!IDPApp(facebookcom))
180       | (!RPApp(integratorcom, facebookcom))
181       | !(in(pub, b:Browser);
182           !UAFetchResources(b,mallory) | !NoServiceWorker(b) | !UA(b)
                )).
183
184   set reconstructDerivation = false.
185   process (
186       BrowserProcess() | HttpServer() | Network()
187       | OAuth()
188       | MaliciousServer()
189       | AttackerLeakApp(mallory)
```

```
)
```

B.1.2 *Generated ServiceWorker*

```
/* in(serviceWorkerFetch(b), (u:Uri, cs__1000, sw__ref:Uri, sw__p:Page, sw__aj:
    Ajax)) */
self.addEventListener('fetch', function(event) {
  try {
    let u = new URL(event.request.url);
    let cs__1000 = event.request;
    let sw__ref = event.request.referer;
    /* let =httpGet() = cs__1000 */
    if (!(("GET") == ((cs__1000).method))) throw new MatchFail();
    /* let uri(=https(), =h, =loginpath(), =nullParams()) = u */
    if (("https") == ((u).protocol.slice(0,-1)) && (h) == ((u).host) && (loginpath
        ) == ((u).pathname) && ("") == ((u).search)) {
      event.respondWith((async () => {
        try {
          /* out(rawRequest(b), (u, cs__1000, sw__ref, sw__p, sw__aj)) */
          var __fetchd = await fetch((u).href, {
            method: (cs__1000).method,
            headers: event.request.headers,
            mode: event.request.mode
          });
          /* in(serviceWorkerResult(b), (=u, cs__1100:HttpResponse, cs__1101:
              ReferrerPolicy, sw__foo:XDR, sw__corr:bitstring)) */
          let cs__1100 = __fetchd;
          let cs__1101 = __fetchd.headers['referrer-policy'];
          /* let httpOk(pagewithlink(uri(=https(), =fb, =oauthpath(),
              codereqparams(=appid, =uri(https(), h, callbackpath(), nullParams
              ()), state)))) = cs__1100 */
          if (!((cs__1100).status == 200)) throw new MatchFail();
          var __fetchd_body = await (cs__1100).clone().text();
          var __pagewithlink_match = (new RegExp((u).protocol.slice(0,-1)+"://"+
              fb+oauthpath+"\\?"+"(?=.*?client_id=(?:[^&]+))(?=.*?redirect_uri
              =(?:[^&]+))(?=.*?state=(?:[^&]+))[^\\\"\\' ]+")).exec(__fetchd_body);
          if (!((__pagewithlink_match))) throw new MatchFail();
          if (!(("https") == ((new URL((__pagewithlink_match)[0])).protocol.
              slice(0,-1)))) throw new MatchFail();
          if (!((fb) == ((new URL((__pagewithlink_match)[0])).host))) throw new
               MatchFail();
          if (!((oauthpath) == ((new URL((__pagewithlink_match)[0])).pathname))
              ) throw new MatchFail();
          if (!((appid) == (parseQuery((new URL((__pagewithlink_match)[0])).
              search)['client_id']))) throw new MatchFail();
```

```
31          if (!(("https") == ((new URL(parseQuery((new URL((__pagewithlink_match)
                [0])).search)['redirect_uri'])).protocol.slice(0,-1)))) throw new
                MatchFail();
32          if (!((h) == ((new URL(parseQuery((new URL((__pagewithlink_match)[0])
                ).search)['redirect_uri'])).host))) throw new MatchFail();
33          if (!((callbackpath) == ((new URL(parseQuery((new URL((
                __pagewithlink_match)[0])).search)['redirect_uri'])).pathname)))
                throw new MatchFail();
34          if (!(("") == ((new URL(parseQuery((new URL((__pagewithlink_match)[0])
                ).search)['redirect_uri'])).search))) throw new MatchFail();
35          let state = parseQuery((new URL((__pagewithlink_match)[0])).search)['
                state'];
36          /* insert MRPSessions(state) */
37          await MRPSessions.insert({col_1: state})
38          /* out(serviceWorkerSendHttpResponse(b), (u, cs__1100, cs__1101,
                sw__foo, sw__corr)) */
39          var __response = (cs__1100)
40          if (cs__1101)
41            __response = await appendHeader(__response, 'referrer-policy', cs__1101
                )
42          return cleanResponse(__response);
43        } catch (e) {
44          console.log(e)
45          console.log('Match error: '+event.request.url)
46          return errorResponse('match error!');
47        }
48      })());
49    } else {
50      /* let uri(=https(), =h, =callbackpath(), coderesparams(code, state)) = u
            */
51      if (("https") == ((u).protocol.slice(0,-1)) && (h) == ((u).host) && (
            callbackpath) == ((u).pathname)) {
52        let code = parseQuery((u).search)['code'];
53        let state = parseQuery((u).search)['state'];
54        event.respondWith((async () => {
55          try {
56            /* get MRPSessions(=state) in */
57            var __table_res = await MRPSessions.findOne({col_1: {$eq: state}})
58            if (!(__table_res)) throw new MatchFail();
59            if (!((state) == (__table_res.col_1))) throw new MatchFail();
60            /* out(rawRequest(b), (u, cs__1000, sw__ref, sw__p, sw__aj)) */
61            var __fetchd = await fetch((u).href, {
62              method: (cs__1000).method,
63              headers: event.request.headers,
64              mode: event.request.mode
65            });
66            /* in(serviceWorkerResult(b), (=u, cs__1200:HttpResponse, cs__1201:
                  ReferrerPolicy, sw__foo:XDR, sw__corr:bitstring)) */
67            let cs__1200 = __fetchd;
```

```
68              let cs__1201 = __fetchd.headers['referrer-policy'];
69              /* let httpOk(success()) = cs__1200 */
70              if (!((cs__1200).status == 200)) throw new MatchFail();
71              var __fetchd_body = await (cs__1200).clone().text();
72              /* out(serviceWorkerSendHttpResponse(b), (u, cs__1200, cs__1201,
                    sw__foo, sw__corr)) */
73              var __response = (cs__1200)
74              __response = await appendHeader(__response, 'referrer-policy', 'no-
                    referrer')
75              return cleanResponse(__response);
76            } catch (e) {
77              console.log(e)
78              console.log('Match error: '+event.request.url)
79              return errorResponse('match error!');
80            }
81          })());
82      }
83    }
84  } catch (e) {}
85 });
```

B.2 PAYPAL - OSCOMMERCE

```
1
2  fun ipnpath(): Path [data]. (* = "/ext/modules/payment/paypal/
       standard_ipn.php" *)
3  fun checkoutpath(): Path [data]. (* = "/checkout_confirmation.php"
       *)
4  fun callbackpath(): Path [data]. (* = "/checkout_process.php" *)
5  fun webscr(): Path [data]. (* = "/cgi-bin/webscr" *)
6
7  free integratorcom: Host. (* = "integrator.com" *)
8  free paypalcom: Host. (* = "www.sandbox.paypal.com"*)
9
10 free integratorMerchantId: Id. (* = "alberto.lupo@business.example.
       com" *)
11
12
13 type Amount.
14 type Invoice.
15 fun filledForm(Params): HTMLtag [data].
16 fun webscrData(Id, Amount, Invoice, Uri, Uri): Params [data]. (* =
       {"business", "amount", "invoice", "notify_url", "return"} *)
17
18 fun ipnData(Id, Amount, Invoice, Id, bitstring): Params [data].
19
20 fun jslink(Uri):bitstring [data].
```

```
21 fun verified():bitstring [data].
22 fun success(): bitstring [data].
23 fun empty(): bitstring [data].
24
25 (* Queries *)
26 event rp_begin(Id, Amount, Invoice, Uri, Uri).
27 event rp_checkout_done().
28 event rp_ipn_verified(Id, Amount, Invoice, Id).
29 event paypal_checkout_complete(Id, Amount, Invoice, Id).
30 event paypal_verified(Id, Amount, Invoice, Id).
31 event ua_begin(Uri, Id, Amount, Invoice, Uri, Uri).
32 event ua_end().
33
34 query id:Id, amt:Amount, inv:Invoice, ntf:Uri, ret:Uri; event(
       rp_begin(id, amt, inv, ntf, ret) ).
35 query event( rp_checkout_done( ) ).
36 query id:Id, amt:Amount, inv:Invoice, payerid:Id; event(
       rp_ipn_verified(id, amt, inv, payerid) ).
37 query id:Id, amt:Amount, inv:Invoice, payerid:Id; event(
       paypal_checkout_complete(id, amt, inv, payerid) ).
38 query id:Id, amt:Amount, inv:Invoice, payerid:Id; event(
       paypal_verified(id, amt, inv, payerid) ).
39 query payuri:Uri, id:Id, amt:Amount, inv:Invoice, ntf:Uri, ret:Uri;
        event( ua_begin(payuri, id, amt, inv, ntf, ret) ).
40 query event( ua_end( ) ).
41
42 query payuri:Uri, id:Id, amt:Amount, inv:Invoice, ntf:Uri, ret:Uri,
        payerid:Id;
43       event( paypal_checkout_complete(id, amt, inv, payerid) )
44    && event( paypal_verified(id, amt, inv, payerid) )
45    && event( rp_ipn_verified(id, amt, inv, payerid) ).
46
47
48 query id:Id, amt:Amount, inv:Invoice, payerid:Id, ntf:Uri, ret:Uri;
49       event( rp_ipn_verified(id, amt, inv, payerid) )
50   ==> event( rp_begin(id, amt, inv, ntf, ret) ).
51
52 query id:Id, amt:Amount, inv:Invoice, payerid:Id, ntf:Uri, ret:Uri;
53       event( rp_ipn_verified(id, amt, inv, payerid) )
54   ==> event( paypal_checkout_complete(id, amt, inv, payerid) ).
55
56
57 (* Utils *)
58 fun mkCookie(bitstring): Cookie [private].
59
60 letfun session_start(cj:CookiePair, corr:bitstring) =
61   let cookiePair(session_cookie,path_cookie) = cj in
62   if session_cookie <> nullCookie() then
63     cj
```

```
64    else
65      cookiePair(mkCookie(corr), nullCookie()).
66
67  let mkserver(h:Host) =
68     (new sk:privkey;
69       let pubk = pk(sk) in
70       insert serverIdentities(orig(https(),h), pubk,sk,xdr());
71       out(pub,pubk)).
72
73  (* Processes *)
74  table Transactions(Invoice, Amount).
75
76  let OSCommerce(h:Host) =
77    let webscruri = uri(https(), paypalcom, webscr(), nullParams())
         in
78    let returi = uri(https(), h, callbackpath(), nullParams()) in
79    let notifyuri = uri(https(), h, ipnpath(), nullParams()) in
80  (
81    (in(httpServerRequest, (u:Uri, hs:Headers, =httpGet(), corr:
         bitstring));
82     let uri(=https(), =h, =checkoutpath(), =nullParams()) = u in
83     let cp = getCookie(hs) in
84     new amt: Amount;
85     new inv: Invoice;
86     insert Transactions(inv, amt);
87     event rp_begin(integratorMerchantId, amt, inv, notifyuri, returi
         );
88     let resp = httpOk(formGen(filledForm(webscrData(
         integratorMerchantId, amt, inv, notifyuri, returi)),
         webscruri)) in
89     out(httpServerResponse, (u, resp, cp, unsafeUrl(), corr)))
90  |
91    (in(httpServerRequest, (u:Uri, hs:Headers, =httpGet(), corr:
         bitstring));
92     let uri(=https(), =h, =callbackpath(), =nullParams()) = u in
93     let cp = getCookie(hs) in
94     event rp_checkout_done();
95     let resp = httpOk(success()) in
96     out(httpServerResponse, (u, resp, cp, unsafeUrl(), corr)))
97  |
98    (in(httpServerRequest, (u:Uri, hs:Headers, req:HttpRequest, corr:
         bitstring));
99     let uri(=https(), =h, =ipnpath(), =nullParams()) = u in
100    let httpPost(ipnd) = req in
101    let ipnData(merchantId, amt, inv, payerId, verisign) = ipnd in
102    let cp = getCookie(hs) in
103    (* validate merchant id *)
104    if merchantId = integratorMerchantId then
105    (* validate invoice and amount *)
```

```
106     get Transactions(=inv, =amt) in
107     (* validate ipn data *)
108     new ncorr: bitstring;
109     out(httpServerRequest, (webscruri, headers(noneUri,
            nullCookiePair(), notajax()), httpPost(ipnd), ncorr));
110     in(httpServerResponse, (=webscruri, httpOk(=verified()), cpXX:
            CookiePair, rpXX:ReferrerPolicy, =ncorr));
111     event rp_ipn_verified(merchantId, amt, inv, payerId);
112     let resp = httpOk(empty()) in
113     out(httpServerResponse, (u, resp, cp, unsafeUrl(), corr)))).
114
115 table ValidSignatures(bitstring, Id, Amount, Invoice, Id).
116
117 let PayPal(h:Host) =
118 (
119   (in(httpServerRequest, (u:Uri, hs:Headers, req:HttpRequest, corr:
          bitstring));
120     let uri(=https(), =h, =webscr(), =nullParams()) = u in
121     let httpPost(webscrData(mch, amt, inv, ntf, ret)) = req in
122     let cp = getCookie(hs) in
123     (* ... login/pay  ... *)
124     new verisign: bitstring;
125     new payerId: Id;
126     insert ValidSignatures(verisign, mch, amt, inv, payerId);
127     event paypal_checkout_complete(mch, amt, inv, payerId);
128      (* Send ipn *)
129      (let ipnd = ipnData(mch, amt, inv, payerId, verisign) in
130       new ncorr:bitstring;
131       out(httpServerRequest, (ntf, headers(noneUri, nullCookiePair()
              , notajax()), httpPost(ipnd), ncorr));
132       in(httpServerResponse, (=ntf, httpOk(=empty()), cpXX:
              CookiePair, rpXX:ReferrerPolicy, =ncorr)))
133      (* redirect to returi *)
134     |(let d = jslink(ret) in
135       out(httpServerResponse, (u, httpOk(d), cp, unsafeUrl(), corr))
              ))
136 |
137   (in(httpServerRequest, (u:Uri,  hs:Headers, req:HttpRequest, corr
          :bitstring));
138     let uri(=https(), =h, =webscr(), =nullParams()) = u in
139     let httpPost(ipnData(merchantId, amt, inv, payerId, verisign)) =
            req in
140     let cp = getCookie(hs) in
141     get ValidSignatures(=verisign, =merchantId, =amt, =inv, =payerId
          ) in
142     event paypal_verified(merchantId, amt, inv, payerId);
143     let resp = httpOk(verified()) in
144     out(httpServerResponse, (u, resp, cp, unsafeUrl(), corr)))).
145
```

```
146
147 let UA(b:Browser) =
148 (
149    (let checkouturi = uri(https(), integratorcom, checkoutpath(),
          nullParams()) in
150     out(browserRequest(b), (checkouturi, httpGet())))
151  |
152    (in (newPage(b),(p:Page, u:Uri ,d:bitstring));
153     let uri(=https(), =integratorcom, =checkoutpath(), =nullParams
          ()) = u in
154     let formGen(filledForm(webscrData(mch, amt, inv, ntf, ret)),
          payuri) = d in
155     event ua_begin(payuri, mch, amt, inv, ntf, ret);
156     out(pageClick(b), (p, payuri, httpPost(webscrData(mch, amt, inv
          , ntf, ret)))))
157  |
158    (in (newPage(b),(p:Page, u:Uri ,d:bitstring));
159     let uri(=https(), =paypalcom, =webscr(), =nullParams()) = u in
160     (* ... login/pay ... *)
161     let jslink(returi) = d in
162     out(pageClick(b), (p, returi, httpGet())))
163  |
164    (in (newPage(b),(p:Page, u:Uri ,d:bitstring));
165     let uri(=https(), =integratorcom, =callbackpath(), =nullParams
          ()) = u in
166     let (=success()) = d in
167     event ua_end())).
168
169
170 (* Network *)
171 process
172         BrowserProcess() | HttpServer() | Network() |
173         mkserver(paypalcom) | !PayPal(paypalcom) |
174         mkserver(integratorcom) | !OSCommerce(integratorcom) |
175         (in(pub, b:Browser); (!NoServiceWorker(b) | !UA(b)))
```

BIBLIOGRAPHY

[1] osCommerce. URL https://www.oscommerce.com/.

[2] Coming may 7th, 2018: A more secure sign-in flow on chrome. URL https://gsuiteupdates.googleblog.com/2018/04/more-secure-sign-in-chrome.html.

[3] Martın Abadi and Cédric Fournet. Private authentication. In *Theoretical Computer Science 322(3):427–476. September 2004. Special issue on Foundations of Wide Area Network Computing. Parts of this work were presented at PET'02 (LNCS 2482) and ISSS'02 (LNCS 2602)*, September 2004.

[4] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Annual Network & Distributed System Security Symposium (NDSS)*. The Internet Society, 2013.

[5] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 247–262, June 2012. doi: 10.1109/CSF.2012.27.

[6] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455782. URL http://doi.acm.org/10.1145/1455770.1455782.

[7] Karthikeyan Bhargavan, Cedric Fournet, Andrew Gordon, and Riccardo Pucella. Tulafale: A security tool for web services. 07 2004. doi: 10.1007/978-3-540-30101-1_9.

[8] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. volume 82, pages 82–96, 02 2001. ISBN 0-7695-1147-3. doi: 10.1109/CSFW.2001.930138.

[9] Bruno Blanchet. Automatic verification of correspondences for security protocols. *J. Comput. Secur.*, 17(4):363–434, December 2009. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1576303.1576304.

[10] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial. URL https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

[11] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *In International Conference on Availability, Reliability and Security (ARES*, 2012.

[12] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally-secure protocol specifications. In *2nd Conference on Principles of Security and Trust (POST 2013), volume 7796 of LNCS*, pages 63–82. Springer, 2013.

[13] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: Fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, Baltimore, MD, 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/calzavara.

[14] Luca Compagna, Daniel dos Santos, Serena Ponta, and Silvio Ranise. Aegis: Automatic enforcement of security policies in workflow-driven web applications. 03 2017. doi: 10.1145/3029806.3029813.

[15] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. URL https://mitmproxy.org/. [Version 4.0].

[16] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983. ISSN 1557-9654. doi: 10.1109/TIT.1983.1056650.

[17] D. Fett, R. Küsters, and G. Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, Aug 2017. doi: 10.1109/CSF.2017.20.

[18] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1204–1215, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978385. URL http://doi.acm.org/10.1145/2976749.2978385.

[19] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 561–570, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526785. URL http://doi.acm.org/10.1145/1526709.1526785.

[20] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL https://rfc-editor.org/rfc/rfc6749.txt.

[21] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. *CoRR*, abs/1901.08960, 2019. URL http://arxiv.org/abs/1901.08960.

[22] Xiaowei li and Yuan Xue. Block: A black-box approach for detection of state violation attacks towards web applications. pages 247–256, 12 2011. doi: 10.1145/2076732.2076767.

[23] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 Threat Model and Security Considerations. RFC 6819, January 2013. URL https://rfc-editor.org/rfc/rfc6819.txt.

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

[25] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. 02 2014.

[26] Alfredo Pironti and Jan Jürjens. Formally-based black-box monitoring of security protocols. pages 79–95, 02 2010. doi: 10.1007/978-3-642-11747-3_7.

[27] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: automatic cryptographic protocol java code generation from spi calculus. volume 1, pages 400– 405 Vol.1, 02 2004. ISBN 0-7695-2051-0. doi: 10.1109/AINA.2004.1283943.

[28] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. Attack patterns for black-box security testing of multi-party web applications. In *NDSS*, 2016.

[29] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. pages 378–390, 10 2012. doi: 10.1145/2382196.2382238.

[30] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online – security analysis of cashier-as-a-service based web stores. In *2011 IEEE Symposium on Security and Privacy*, pages 465–480, May 2011. doi: 10.1109/SP.2011.26.

[31] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. IEEE Computer Society, May 2012. URL https://www.microsoft.com/en-us/research/publication/signing-me-onto-your-accounts-through-facebook-and-google-a-traffic-guided-security-study-of-commercially-deployed-single-sign-on-web-services/.

[32] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating sdks: uncovering assumptions underlying secure authentication and authorization. pages 399–414, 08 2013.

[33] Web Hypertext Application Technology Working Group (WHATWG). Fetch: Living standard. URL https://fetch.spec.whatwg.org/.

[34] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, May 1993. doi: 10.1109/RISP.1993.287633.

[35] Luyi Xing, Yangyi Chen, Xiaofeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. the 20th annual network. 02 2013.

[36] Ronghai Yang, Guanchen Li, Wing Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. pages 651–662, 05 2016. doi: 10.1145/2897845.2897874.

[37] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting single sign-on SDK implementations via symbolic reasoning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1459–1474, Baltimore, MD, 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/yang.

[38] Q. Ye, G. Bai, K. Wang, and J. S. Dong. Formal analysis of a single sign-on protocol implementation for android. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 90–99, Dec 2015. doi: 10.1109/ICECCS.2015.20.