Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Formal Security Analysis of the Web Payment APIs

Nils Wenzler

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Tim Würtele, M.Sc., Pedram Hosseyni, M.Sc.
Commenced:	September 10th, 2019
Completed:	March 10th, 2020

Abstract

The Web Payment APIs are a set of specifications by the W3C Web Payments Working Group that aim to offer a set of new and improved checkout mechanisms for the web.

Thousands of online shops provide customers with nearly endless possibilities of buying products. Although they differ in their products and customers, they all share the need for a checkout process to obtain customer information and a corresponding financial transaction. As these specifications strive to become the new standard for web payments, security is a crucial aspect.

In this work, we created an extended version of the Web Infrastructure Model by expanding it with the APIs and functionalities described in the aforementioned specifications of the Web Payment APIs. Within the model, we performed a formal security analysis which led to the discovery of a possible attack and additional vulnerabilities.

We offered mitigations against this attack and said vulnerabilities and showed that the resulting model satisfies the modeled security properties.

By doing so, we show that the resulting model guarantees that payments can only be performed in an authorized manner and that the integrity of the financial transactions is ensured.

After making the Chromium team aware of the found attack, they implemented our proposed mitigation approach and released a patch that was already distributed to millions of devices at the time of writing.

Kurzfassung

Mit Hilfe der Web Payment APIs will die W3C Web Payments Working Group einen Standard etablieren der schnellere, einfachere und sicherere Bezahlvorgänge im Internet ermöglichen soll. Die in den Spezifikationen beschriebenen Schnittstellen und Erweiterungen ermöglichen es dem Browser als Vermittler zwischen dem Käufer, dem Verkäufer und dem Anbieter der Zahlungsmethode aufzutreten. Da Sicherheit ein sehr wichtiger Aspekt be Zahlungsvorgängen darstellt, untersuchte diese Arbeit die Sicherheitseigenschaften der Web Payment APIs.

Dafür wurden innerhalb dieser Arbeit die Web Payment APIs in das Web Infrastructure Model integriert um dort formalisierte Sicherheitseigenschaften zu untersuchen.

Bei dieser Untersuchung wurden eine Angriffsmöglichkeit und zwei mögliche Schwachstellen innerhalb der Spezifikationen gefunden. Innerhalb des Models zeigen wir auf, durch welche Änderungen diese Angriffsmöglichkeit und die Schwachstelle behoben werden können, sodass die spezifizierten Sicherheitseigenschaften erfüllt werden.

Contents

List of Algorithms

1 Introduction

Several million online stores power the e-commerce industry of the modern web [17]. Although their checkout processes have a need for very similar information (name, address, phone number, email address, etc.) and very similar payment processes, a lack of common and standardized interfaces for these checkout processes forces shop owners to implement these themselves. Since customers have to provide their data to every single shop instance, their checkout experience is less convenient than it could be.

The W3C Web Payments Working Group [25] aims at improving this situation by currently specifying the Payment Handler API [19], the Payment Request API [23], the Payment Method Manifest [21], and the Payment Method Identifier [20]. We refer to these specifications by the name Web Payment APIs.

The combination of these specifications extends the browser with functionalities that allows it to become an intermediary that enables a convenient checkout process for the payer, the payee and the provider of the payment method (such as banks, Google Pay [11], PayPal [15], Apple Pay [2], etc.).

A main functionality of the specification lies within storing relevant customer information in the browser, such that it allows for a faster and more convenient checkout process for the payer over all online stores that use the APIs. Additionally, the browser is extended by Payment Handlers which are service worker scripts that are provided by payment method providers. These scripts can be installed on the browser to enable the browser to support the corresponding payment methods in a centralized manner. Besides convenience, the aforementioned specifications strive to improve the privacy and security of internet checkout processes.

The goal of this thesis is to perform an in-depth formal security analysis of the current state of the Web Payment APIs. This analysis is enabled by modeling the relevant components and protocols in the Web Infrastructure Model [8]. Most of this work will focus on extending the browser model of the Web Infrastructure Model, since most of the extensions of the Web Payment APIs concern the browser. Within the extended model, the intended security goals will be formalized and analyzed. These security goals are based on the security considerations of the specifications but extended with further goals to create a comprehensive set of security goals. By using the formalization of these security goals and the extended Web Infrastructure Model, a formal security proof of the Web Payment APIs will be provided.

Additionally, there is a need to extend the basic structure of the model of the browser within the Web Infrastructure Model to model structures that have not been modelled in the WIM before. The main addition lies within JavaScript promise based communication and communication flows within the browser itself. By providing an extension to the Web Infrastructure Model that enables such flows, we provide a foundation for further research based on the Web Infrastructure Model.

2 The Web Payment APIs

Within a common online checkout process, a merchant (e.g., the owner of an online store) obtains information such as the shipping address of a customer (the payer) and sets up a financial transaction at a payment provider (such as banks, Google Pay [11], PayPal [15], Apple Pay [2], etc.) for the payer to issue.

Nowadays, the single merchants implement their own checkout processes although they serve the same purpose. This situation is unfavorable for a customer/payer as well, since she has to repeatingly provide unchanging information such as the shipping address to each online store that she uses.

The Web Payment APIs introduce a set of features to modern user agents (e.g., browsers), that allow for a fast and standardized checkout process in the web.

Within this process, the browser takes the role of an intermediary that manages the communication between the three common stakeholders in such an online checkout process: the merchant, the payer and the payment method provider.

The browser offers to each stakeholder standardized interfaces for these checkout processes.

The APIs strive to fulfill the following goals:

Standardization Through the Web Payment APIs web checkout processes become standardized. This allows to introduce reoccurring checkout experiences and helps developers by simplifying the integration process.

Security By introducing payment handlers that can be installed to the browser separately, payment providers have the ability to introduce more complex and more secure protocols which can be updated independently of merchants. Furthermore, the specification introduces a trusted payment UI which is served directly by the browser.

Faster and more convenient checkout experiences By enabling browsers to act as intermediaries, they can introduce features such as storing the user's shipping address and their payment methods (such as credit cards). This makes the checkout process more convenient and faster. The time to check out is an economically relevant metric for merchants, since during checkout many users do abort and not complete their buy [5].

Privacy Through the Web Payment APIs, only the relevant information has to be submitted to the single parties. Furthermore, a merchant might not feel the need to force a customer to create a user account and store the associated user data indefinitely if a fast and easy checkout process can be enabled without such an account.

2.1 Involved Parties

The three main parties are the merchant, the payer and the payment provider. The following section explains which components are associated to the single parties and offers a short overview of how they interact.

2.1.1 The Merchant

The merchant serves a regular website with an online shop. For its checkout process, it uses the JavaScript APIs of the browser to create a payment request, check for support of the payment methods and the APIs, and to show and to complete a checkout process. Within the payment request, the merchant specifies which payment methods are supported in the checkout process.

2.1.2 The Payer

The payer/customer does interact with the Web Payment APIs through the UI of the browser. After the merchant created the payment request, the merchant can tell the browser to show the corresponding trusted payment UI upon a click of the user on a button of the website. The user is then presented with a separate browser UI, that shows the total, the selected items, a prefilled shipping address, email address and so forth. In the UI, the user can check the corresponding information, select a payment handler for a supported payment method, and submit the payment to the corresponding payment handler.

2.1.3 The Payment Method Provider

The payment method provider can use several components to enable its payment service for the user.

So-called payment handlers are scripts or features that enable a browser to support a payment method.

There are special standardized payment methods such as `basic-card` [4], which can be used to ask the user for credit card information. These are usually directly integrated into the browser and do not use external scripts.

Other payment methods are identified by URLs (such as `http://paypal.com/pay`). Payment handlers for such payment methods are either directly integrated into the browser (as for Apple Pay [2] in Safari [4]), or their support can be installed via external scripts into service workers that are registered under the payment method's URL.

Service workers are scripts that can be installed to a browser that run outside the scope of a window or a document. They allow browser to be extended with functionalities whose domain lies outside of a document scope. Examples for such features are serving offline versions of websites, dealing with push notifications and the for the Web Payment APIs relevant support for payment method providers.

For a single payment method, there might be several supported payment handlers registered. The configuration of a payment method can be obtained through its payment method manifest (see Section 2.2).

Furthermore, a single payment handler might support several payment methods. The supported payment methods of a payment handler are called its payment instruments.

2.2 APIs and Specifications of the Web Payment APIs

The Web Payment APIs are defined by the following set of documents.

The Payment Request API [23] The Payment Request API specifies the communication between the merchant's website and the user agent. The general payment process is started through this API. All relevant properties of the payment request are specified through this API. Total cost, shipping cost, the need for a contact phone number and the supported payment methods is a selection of the most relevant of said properties.

The Payment Handler API [19] The Payment Handler API specifies the interfaces through which a payment method provider can communicate through installed payment handlers with the user agent to process a payment request. Internally, payment handlers are service workers that are able to process payment requests through an extended service worker API. Note that earlier mentioned payment handlers that are directly implemented into the browser commonly do not use the Payment Handler API although they are called payment handlers nonetheless.

Payment Method Identifier [20] This specification describes how payment methods are identified in the Web Payment APIs. Besides predefined payment methods such as `basic-card` [4], Payment Method Identifier essentially are URLs.

Payment Method Manifest [21] The Payment Method Manifest specifies a process by which user agents can obtain necessary information to install payment handlers and verify their validity. Through this specification, user agents are enabled to automatically install payment handlers. When a user agent is asked to process a payment with a so-far unknown payment metod identifier, the user agent queries the payment method manifest by making a call to the URL of the payment method identifier. Under this URL, a link header references a json manifest file that offers the necessary information.

Payment Method: Basic Card [22] This specification defines the first specified preinstalled named payment handler that can be used to obtain credit card information of a user.

2.3 General Flow

To get a basic understanding of how these APIs commonly interact with each other during a payment process, a short informal description of a payment process with the Web Payment APIs is offered in Figure 2.1.

Note that all depicted actors are located within the browser. The second column "Browser/User Agent" depicts the implementation of the browser internal implementations of the Web Payment APIs.

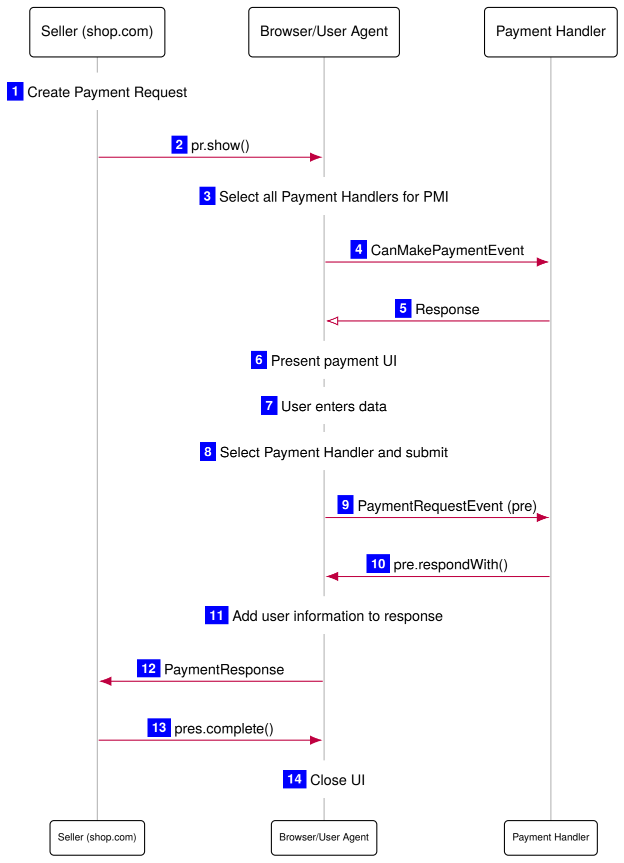

Figure 2.1 General flow of Web Payments

The checkout process starts through JavaScript code served by the merchants web server.

In this code, a Payment Request first needs to be created (Step $\boxed{1}$). All basic information is passed to the Payment Request at creation time. Among others, this information includes the total to pay and the payment method identifiers (PMIs) which the merchant accepts.

As soon as the user clicks e.g., a checkout button, the merchant's website calls the `show()` method of the Payment Request (Step $\boxed{2}$).

This tells the user agent to start the checkout process. In a next step, the user agent determines which installed payment handlers are able to process the requested payment.

These are a subset of the payment handlers that support a payment instrument with a payment method identifier (PMI), that is also present in the payment request (Step $\boxed{3}$). There are further aspects that might determine whether a payment handler is able to process a payment or not. Imagine for example a payment method that has an internal heuristic of whether a payment request seems trustworthy, or varying maximum supported totals depending on the shop being used. To cater to such needs, the Web Payment APIs use preflight CanMakePaymentEvents that ask all potential payment handlers whether they would support a concrete payment request (Steps $\boxed{4}$ and $\boxed{5}$). In the trusted payment UI only those payment handlers are displayed to the user, that respond with true to this event (Step $\boxed{6}$).

In this UI the user is able to select a payment handler and enter all relevant data such as her shipping address. Since this overlay can not directly be modified by anyone except the browser (especially not the merchant's website), this is also refered to as trusted UI.

Upon clicking a submit button in this trusted UI the user agent submits a PaymentRequestEvent to the selected payment handler (Step $\boxed{9}$).

The PaymentRequestEvent only contains the information of the payment request that is necessary for the payment handler to process the transaction.

The payment handler uses this information to trigger the processing of the financial transaction and returns a PaymentHandlerResponse (Step $\boxed{10}$).

The PaymentHandlerResponse contains payment provider specific details, that for example can later be used by the merchant to validate the transaction.

After receiving this information, the user agent merges the entered information (shipping address, email address, shipping option, etc.) of the user with the response of the payment handler and submits it to the merchants website as a PaymentResponse object through a JavaScript promise (Steps $\boxed{11}$ and $\boxed{12}$).

To acknowledge that the payment has been processed, the merchant's website afterwards calls the complete method of the payment response (Step $\boxed{13}$), which closes the user agent's overlay (Step $\boxed{14}$). This last step allows the merchant's website to validate whether the resulting data is in accordance to the merchant's requirements. If this is not the case, the merchant is able to trigger a retry, which is explained in detail in the following section.

2.4 Extended Flows

In addtion to the earlier presented minimalistic flow (Figure 2.1), there are several possibilities to divert from this flow. The following short descriptions offer an informal overview over the main features of the Web Payment APIs.

2.4.1 Retry

After the merchant has received the PaymentResponse, she does not have to call the complete method. In case of issues with the result, the merchant can decide to issue a retry of the payment. Imagine for instance that the merchant asks the user for an email address during checkout. If the merchant has a blacklist or whitelist of email addresses that are supported and the user submits an email address that is listed on said blacklist, the merchant can issue a retry telling the user to pick an email address that satisfies the requirements.

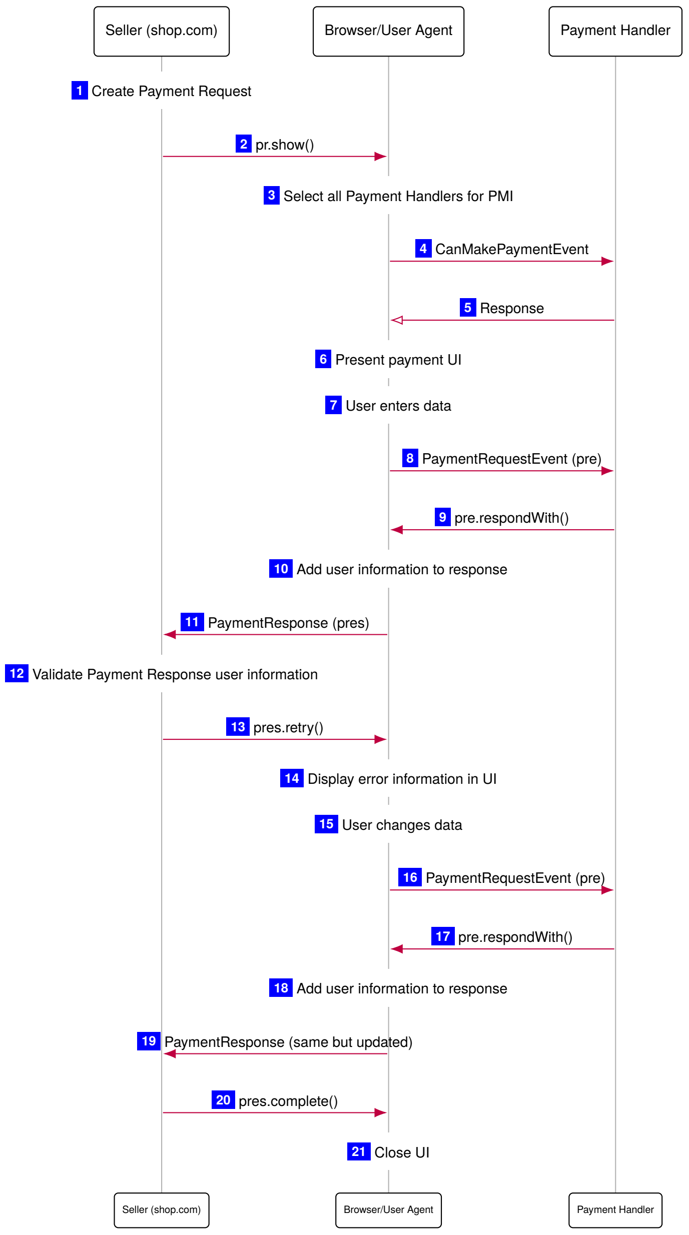

Figure 2.2 Flow of Web Payment with retry due to invalid user data

Figure 2.2 shows the corresponding flow of a retry during checkout. The initial flow of the checkout corresponds to one presented in the general flow (Step $\boxed{1}$ to $\boxed{11}$). In Step $\boxed{12}$, the merchant determines that a retry is necessary. The merchant therefore calls the `retry()` method of the payment response (Step $\boxed{13}$). The remainder of the flow (Steps $\boxed{14}$ to $\boxed{21}$) resembles a second checkout process.

The major differences are that the user is not allowed to select a different payment handler and that error information is displayed in the payment UI (Step $\boxed{14}$). The payment handler can not differentiate a retry from an original PaymentRequestEvent except for the reccurence of an earlier observed payment request id. The payment request id is determined on creation of the payment request either by the merchant or if none is provider by a randomly generated UUID [14].

2.4.2 Shipping Address Change

In some cases, the final total to be payed or the available shipping options (express delivery, registered/tracked packages, acknowledgement of receipt, etc.) depend on the selected shipping address. For such cases, the specification implements a mechanism which allows a merchant to react to an entered shipping address and potentially update the payment request accordingly.

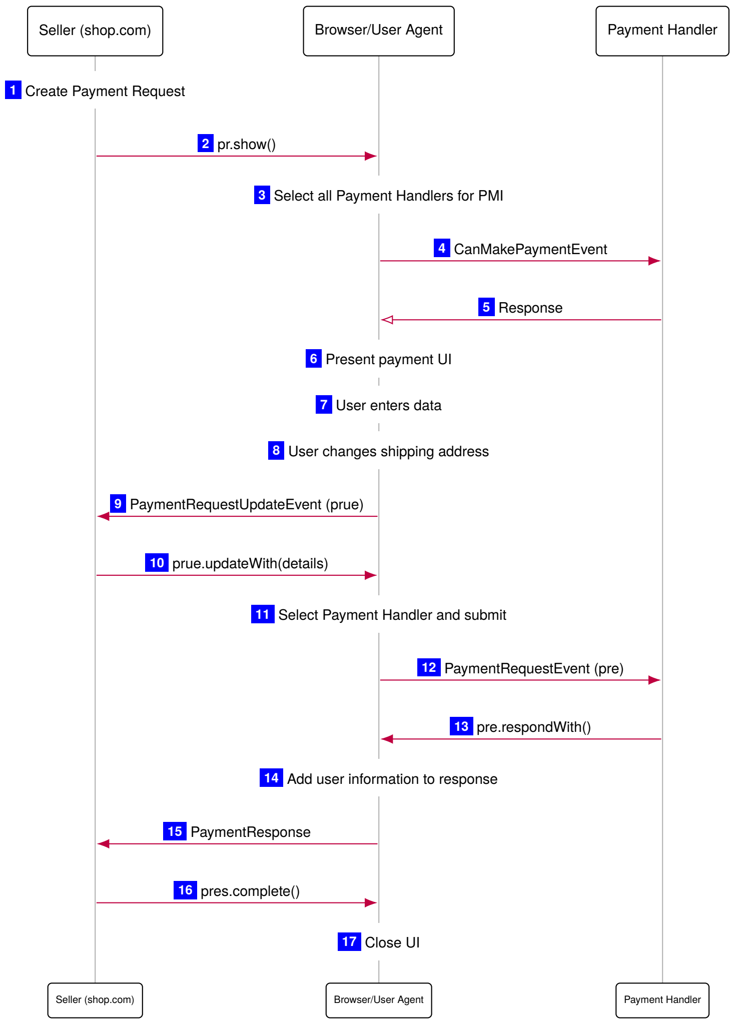

Figure 2.3 Flow of Web Payment with shipping address change

Figure 2.3 shows the corresponding flow in case of a shipping address change. The steps that differ from the flow in Figure 2.1 are the Steps $\boxed{8}$ to $\boxed{10}$. In Step $\boxed{8}$, the user manually selects or enters a shipping address to be used. Following this event, the merchant gets notified with an PaymentRequestUpdateEvent (Step $\boxed{9}$). This event contains a redacted shipping address that contains information such as the country and the postal code of the recipient. Furthermore, this event contains a handle, that allows a merchant to update the payment request's total, shipping options, display items, payment modifiers (such as credit card fees) and further payment method specific data. In Step $\boxed{10}$, the merchant uses this handle to update e.g., the available shipping options.

The remainder of the checkout flow stays the same. Such PaymentRequestUpdateEvents can be issued repeatedly and are also used in case of a change of shipping option and payer info (such as email address or phone number).

Note that pre-filled/auto filled values (values that are automatically filled by a browser) do not trigger such an event [12]. If a merchant does need the recipient's shipping address to determine the shipping cost, the correspondent payment request has to initialize the payment request with $options$.`requestShippingAddress` = $true$ and it initially has to offer no shipping option. This leads the browser to not auto filling the shipping address. When the user manually selects the shipping address afterwards, the corresponding PaymentRequestUpdateEvent is used to show the available shipping options.

2.4.3 Open Window

A payment handler might wish to ask the user for additional information during the checkout process. Such information could be a separate acknowledgment of the transaction or e.g., credentials to verify the identity of the payer. To allow payment handlers to obtain such additional information, the specification introduced an `openWindow`() method in the PaymentRequestEvent that is issued to the payment handler on submission of the payment by the user.

Figure 2.4 depicts such a payment checkout flow. By calling the `openWindow`() method, a payment handler might open a window which shares the same scope as its service worker (Step $\boxed{10}$). The opened website is displayed within the trusted UI of the browser [12]. Through this window, the user is able to perform the necessary interactions (Step $\boxed{11}$). In the specification, it is not defined how the user's information is made available to the payment handler afterwards. Theoretically, such a communication might not even be necessary if the opened website itself triggers the transaction. As in the reference implementation [18], we model that the window communicates with the payment handler through post messages (Step $\boxed{12}$). In the model the payment handler afterwards triggers the payment.

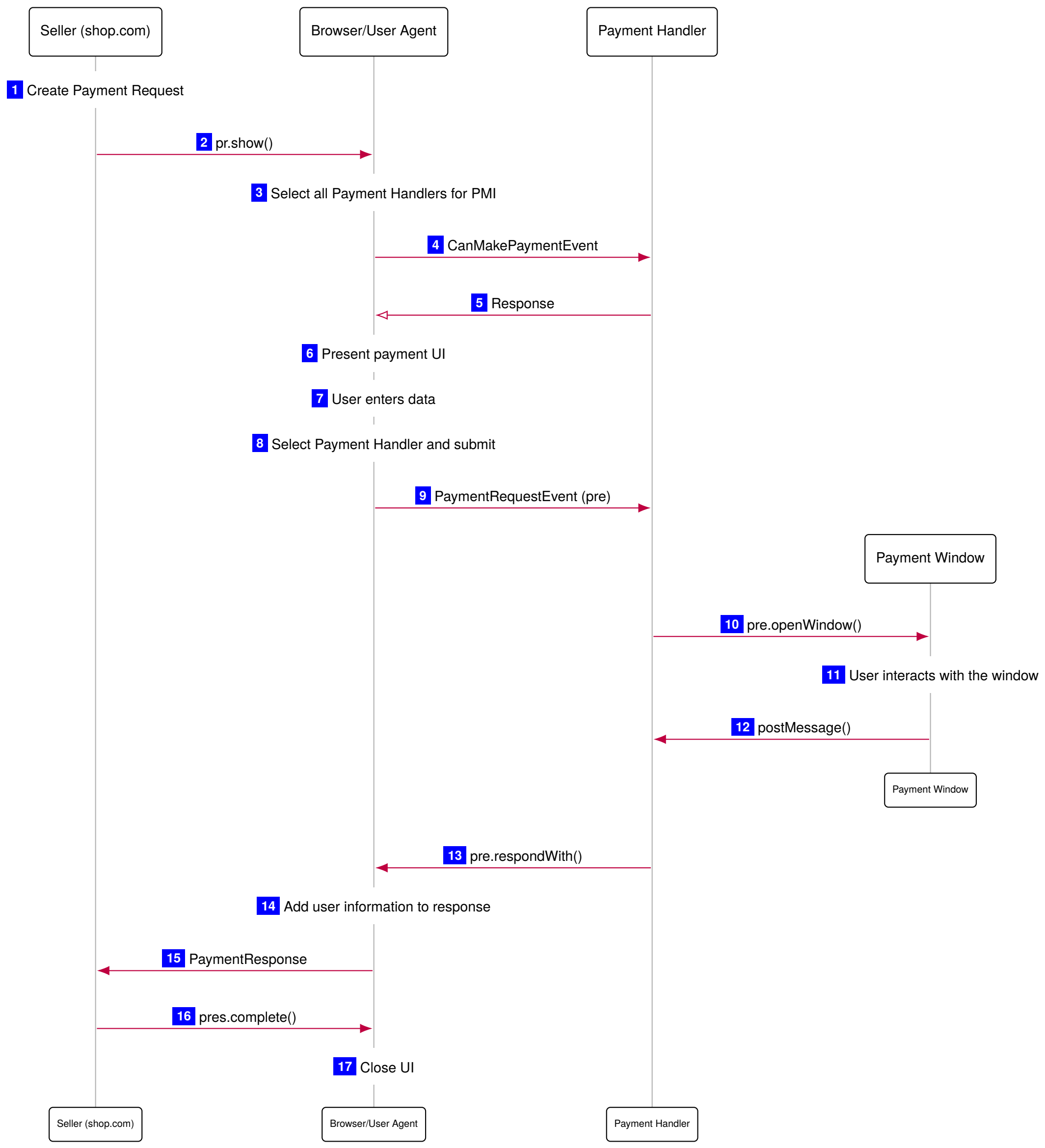

Figure 2.4 Flow of Web Payment including subwindow communication

2.4.4 Merchant Validation

Merchant validation was introduced to allow for merchant validation through the payment method provider. Through merchant validation, a payment provider verifies that a payment requet originates from the corresponding merchant and no other potentially malicious party. The specification is not very clear concerning the exact flow of merchant validation. But since it was mostly introduced to allow integration of Apple Pay [2], we present and model a flow which is based on the merchant validation flow of Apple Pay [3].

Figure 2.5 depicts the relevant flow containing merchant validation.

As soon as the payment request is shown, following Step $\boxed{2}$, the merchant validation starts. The browser blocks the payment UI until the merchant validation has completed. Depending on where the payment handler is located, a corresponding service worker (Step $\boxed{7}$) or the browser itself (Step $\boxed{8}$) issues a PaymentValidationEvent to the merchant's website. The PaymentValidationEvent contains a *validationUrl* that the merchant should call from its server [3] (not out of the user agent's context).

Therefore, the merchant's website passes the received validationUrl to its server (Step $\boxed{9}$), which then issues a request to the corresponding URL (Step $\boxed{10}$).

The corresponding payment provider server then shares a session token with the server of the merchant (Step $\boxed{11}$). The merchant's server afterwards passes this token back to the website of the merchant in the browser $\boxed{12}$.

With this data, the client can call the `comlete()` method of the PaymentValidationEvent (Step $\boxed{13}$). Afterwards, the token is validated in the payment handler, which again might either be implemented directly in the browser or as a separate service worker (Step $\boxed{14}$).

Merchant validation was not considered in this thesis since its status in the specification does not seem to be stable yet.

This is visible in the fact that in the current payment handler specification [19] the specification does not determine how such a Step as $\boxed{7}$ would be triggered.

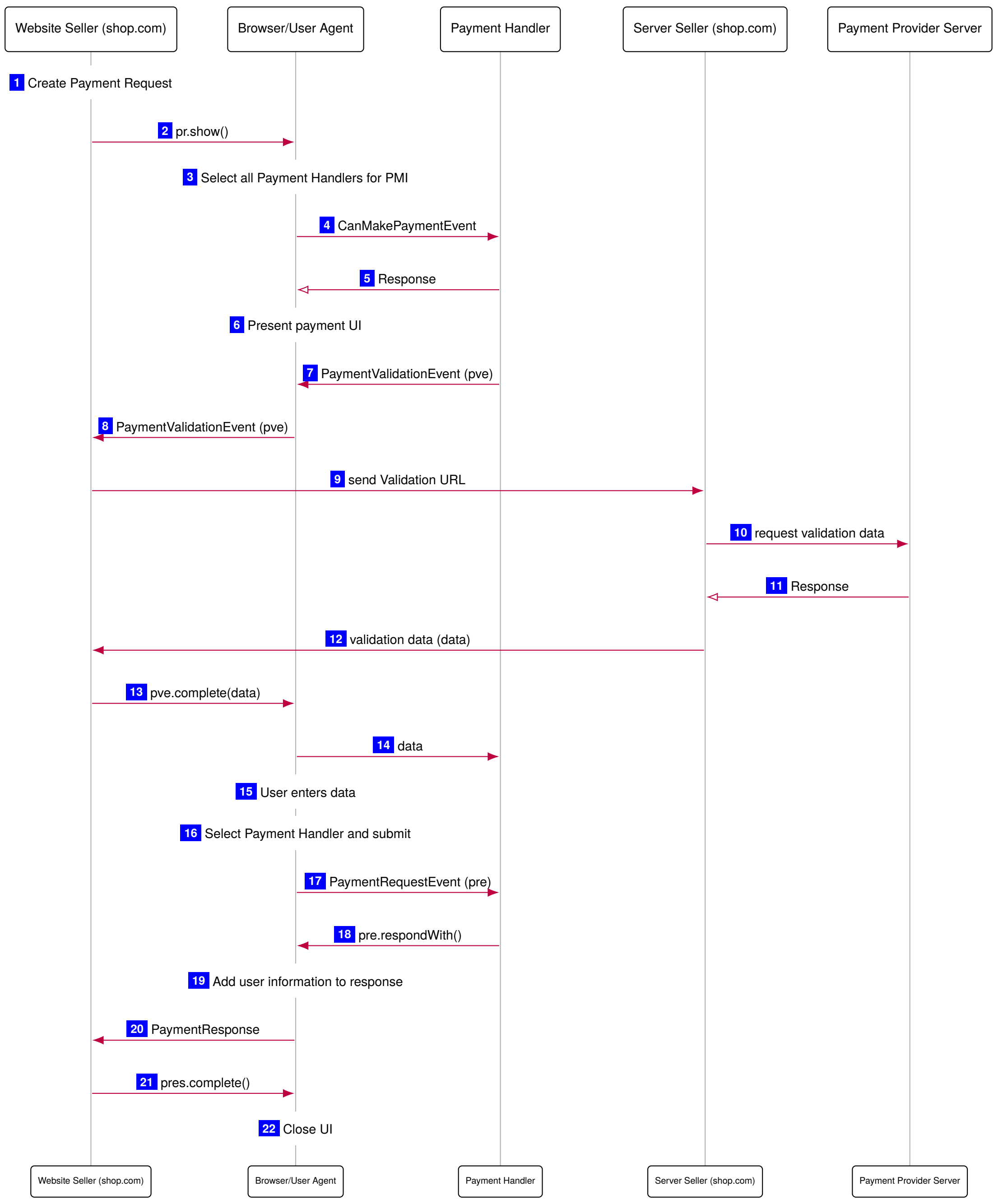

Figure 2.5 Flow of Web Payment with merchant validation

3 Overview over the model of the Web Payment APIs

In the following sections, we offer a short introduction into the Web Infrastructure Model [8] and describe how it was extended and instantiated to fit the needs of the analysis of the Web Payment APIs.

3.1 The Web Infrastructure Model

The Web Infrastructure Model (WIM), is a generic Dolev-Yao style model of the web and its infrastructure. The following short description is taken from Section 4 of [7][1].

The WIM is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM model defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers.

Communication Model. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a “pool” of waiting events and is delivered to one of the processes that listens to the event’s receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP request for the URI `http://ex.com/show?p=1` is represented as $r := \langle \mathsf{HTTPReq}, n_1, \mathsf{GET}, \mathsf{ex.com}, \mathsf{/show}, \langle\langle \mathsf{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$ where the body and the list of request headers is empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

[1]In its original publication, the model was named FKS in resemblance of the names of the main authors. Later on the model was renamed to the name used here, the Web Infrastructure Model. The presented section was adapted accordingly.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k' \rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

The so-called *attacker process* is a Dolev-Yao process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties.

A *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). An annotated example for a script can be found in the Appendix of [8]. Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

A *system* is a set of processes. A *configuration* of this system consists of the states of all processes in the system, the pool of waiting events, and a sequence of unused nonces. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example, $Q = (S, E, N) \longrightarrow (S', E', N')$ to denote the transition from the configuration (S, E, N) to the configuration (S', E', N'), where S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces.

A *web system* formalizes the web infrastructure and web applications. It contains a system consisting of honest and attacker processes. Honest processes can be web browsers, web servers, or DNS servers. Attackers can be either *web attackers* (who can listen to and send messages from their own addresses only) or *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A web system further contains a set of scripts (comprising honest scripts and the attacker script).

In our analysis of the Web Payment APIs, we consider a combination of network attackers and malicious merchant servers. The here presented model of the Web Payment APIs extends the WIM with extended definitions of the browser behaviours, server scripts, serivce worker scripts and regular scripts. These are not defined by the WIM model since they depend on the specific application, unless they become corrupted, in which case they behave like attacker processes and attacker scripts; browsers are specified by the WIM model (see below). The modeling of the Web Payment API specific additions, is outlined in Section 3.2 and with full details provided in Appendices A and B.

Web Browsers. An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the web browser. User credentials are stored in the initial state of the browser and are given to selected web pages when needed. Besides user credentials, the state of a web browser contains (among others) a tree of windows and documents, cookies, and web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded web page and contains (among others) a script and a list of subwindows (modeling iframes). The script, when triggered by the browser, is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and web storage data, and create iframes. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant web standards.

A browser can output messages on the network of different types, namely DNS and HTTP(S) (including XMLHttpRequests), and it processes the responses. Several HTTP(S) headers are modeled, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a so-called trigger message upon which the browser non-deterministically choses an action, for instance, to trigger a script in some document. The script now outputs a command, as described above, which is then further processed by the browser. Browsers can also become corrupted, i.e., be taken over by web and network attackers. Once corrupted, a browser behaves like an attacker process.

For a more detailed description of the Web Infrastructure Model, we refer to the appendices of [8].

3.2 Informal Description of the Web Payment API Model

The following sections give an informal description of the modeling of the Web Payment APIs and specifications. The relevant areas of the extension are on the one hand changes concerning the browser and on the other hand the model of the payment provider servers. Although the payment provider servers are not part of the APIs and specifications, a model is provided to allow for an investigation of the properties of the whole system.

3.2.1 The Browser

In the current version of the WIM [8], the browser already supports major functionalities such as navigating websites, the execution of scripts, WebSockets, WebRTC, AJAX requests, post message communication, and more.

A major difference had to be introduced to depict the new kind of internal communication within the browser. Within the Web Payment APIs, the three stakeholders of the merchant, the payer and the payment provider (represented through the payment handler), communicate through APIs of the browser. The communication uses the asyncronous pattern of JavaScript promises. Promises are basically objects that contain callback methods that are triggered in case of a reply. In case of the Web Payment APIs the browser's internal mechanisms trigger these callbacks based on corresponding interaction with its APIs. For example, when the user calls the `show()` method of a payment request, this returns a promise through which later on the browser will share the PaymentResponse object with the merhant's site.

This internal, not network but promise based, communication was a new feature that had to be introduced. Within its implementation, it is important to allow arbitrary asynchronous orderings of communication and event processing of these actors within the browser. The browser was extended with a set of internal *events*, that are non-deterministically selected and processed. These events are the mean of communication for all three stakeholders.

In this extension, one can imagine the browser itself becoming a 'sub-Dolev-Yao-model'. Within the browser, the actors have their own knowledge which is stored in their *scriptstate*s and communicate through the processing and creation of events.

Besides this general approach, one can group the extensions of the browser into two groups. Extensions of the browser's API to introduce the interfaces needed by the Web Payment APIs and specifications, and the extension of the browser to model service workers and their payment handlers. The interaction of the provider of the payment method within the browser is intended to be performed by payment handlers, which are so-called service worker.

The User Agent's Payment APIs. Since the browser becomes an active intermediary in the Web Payment APIs, it has to provide the necessary interfaces to the three main stakeholders: merchant, payer and provider of the payment method.

The merchant communicates with said APIs through new JavaScript interfaces. Therefore, its main extension lies within an extension of the available JavaScript interfaces within the RUNSCRIPT algorithm (Algorithm 2). Its main capabilities lie within creating PaymentRequests, and starting, updating, aborting and completing payment processes.

The payer's interaction is hidden within the internal processing of events within the browser. Within the model, no seperate actor such as the payer/user of the browser is modeled. User interaction is modeled as non-deterministic actions that can be chosen by the browser. Therefore, its most relevant sections are within the RUNSCRIPT algorithm (Algorithm 2) and the PROCESSEVENT algorithm (Algorithm 3). One of the most relevant aspects is the submission of a PaymentRequestEvent. This event represents the expression of the user's consent to pay.

The Model of Service Workers . Service workers [24] are JavaScript environments that can be installed on the browser. They run independently of any document or window. Although they can be used to solve a huge set of problems, within the Web Payment APIs they are used to implement the payment handlers. Payment handlers are JavaScript implementations provided by the payment method providers. Their task is to connect to the standardized interface of the browser to trigger transactions, while dealing with the payment method specific processes internally.

To provide an exhaustive model of the installation process and the feature set of service workers, would have gone beyond the scope of this work. Therefore, within this model two major simplifications have been introduced. First, within this model it is assumed that the needed service workers are preinstalled. Possible flaws within this process are considered to be out of scope of this work. They would originate from the service worker specification [24] and would not be payment handler specific.

Secondly, the modeled features of service workers are only the subset of features that is directly relevant to the implementation of the Web Payment APIs.

The concrete model of the service workers heavily leans on the existing model of scripts in the browser. They have their own scriptinputs and scriptstates. Since they do not run in a window context, they are not modeled as being part of windows but are stored as a direct property of the browser state $Z_{\text{Webbrowser}}$. Their most relevant algorithm is the RUNWORKER algorithm (Algorithm 4)

Service workers are able to communicate with other parts of the browser through the earlier explained event system, post messages and indirectly through network communication (such as XHR).

Payment handlers listen to PaymentRequestEvents which are fired after a user expresses consent to pay. Such a PaymentRequestEvent is processed and can be responded to with a PaymentHandlerResponse. This PaymentHandlerResponse is used to communicate details of the transaction and its status.

3.2.2 The Payment Provider Server

The payment provider server models a basic HTTPS server. It uses the template of the original WIM model [8].

The server offers three endpoints: `/index`, `/authenticate`, and `/pay`.

`/index` serves a website that allows a user to obtain an authentication token by calling `/authenticate`. Afterwards, this token is submitted to the relevant payment handler with a postMessage.

`/pay` expects an authentication token and the information necessary to perform a financial transaction.

The payment provider server keeps track of all requested transactions.

3.2.3 The Merchant Server

The payment provider server models a basic HTTPS server. It uses the template of the original WIM [8].

The server offers a single endpoint `/index`, that serves the script `script_arbitrary_merchant`.

The merchant server is not explicitly necessary in the model, since the network attacker could have served the page as well. For sake of completeness and better understandability it was added in this model though.

3.3 Security Properties

In this section, we present the security properties that are relevant for the Web Payment APIs. We use a similar categorization as presented in [6]. The three major aspects are: Session Integrity, Confidentiality/Privacy and Payment Integrity. We present these properties here in an informal way. For a formal representation of the security properties, we refer to Appendix C.

3.3.1 Session Integrity

Informally put, it must hold true that only a payer can issue payments of her account. No malicious agent should be able to issue payments of an unaware payer. In our extensions, we provide a model of the intended payments of each user. For the formal definition of the property refer to Section C.1, for the corresponding proof refer to Section D.2.

3.3.2 Confidentiality/Privacy

Informally put, it must hold true that no sensitive information should leak to merchants, payment method providers or other malicious third parties. This property is dependent on whether the payer did express payment intent or not. After a payer issues a payment, it is intended that for example a merchant learns the payer's address to perform shipping of goods. A formal analysis of the privacy and confidentiality properties of the Web Payment APIs would have went beyond the scope of this work. Nonetheless, we mention potential issues that were found during our work.

3.3.3 Payment Integrity

Intuitively, payment integrity describes that the payment that a user authorized is the payment that later on is performed. It should not be possible for a malicious agent to change data such as the receiver of the payment, the shipping address and the amount to be paid. This property is dependent on the users intents, that has to be compared to the execution of payments. The main aspects that we proof is that the total of a transaction matches the user's intent as well as the sender and the receiver of a transaction. For the formal definition of the property refer to Section C.2, for the corresponding proof refer to Section D.4 and Section D.3.

4 Attacks and Vulnerabilities

Through our formal analysis of the Web Payment APIs, we found one attack concerning payment integrity and two vulnerabilities, concerning privacy and payment integrity, that are in conflict with the formalized security properties. Within the following sections, we offer a description of these.

4.1 Double Charging Through Retry Mechanism

The retry mechanism allows the merchant to ask the user to retry the payment in case of an error during processing [13]. Such errors could be based on invalid credit cards, invalid shipping addresses, or user information such as the email address being invalid. The retry method is located as a method in the PaymentResponse object that the merchants website receives to either `complete()`, which closes the payment and its UI, or to `retry()`, which allows to restart a payment process of a given PaymentRequest and display error messages. In case of a retry, the payment process flow is very similar to the regular payment process starting with the `show()` method of the PaymentRequest object. For a further description of the retry flow, see Subsection 2.4.1.

Whether payment handlers could be changed during this retry mechanism was unclear to us during modeling. We therefore looked at the implementation provided by the Chromium browser [16]. In their implementation the user is allowed to change payment providers during retry.

During the proof of the security properties of the model, we found an attack based on this feature that allows a malicious merchant to trick a user to be charged twice during a single checkout flow with a retry.

After finding this issue, the authors of the specification did confirm, that it is not intended that a user might change payment handlers during a retry [10].

4.1.1 Assumptions

For this attack, we assume that it is allowed to change payment handlers during a retry of a payment process. This assumption is based on the behaviour of the reference implementation in the Chromium browser [16]. We chose this browser as a reference point because the Chromium browser is currently the only major browser that ships with support of the Payment Request API [23] and the Payment Handler API [19].

4.1.2 Scenario

A malicious merchant tries to charge a customer twice through malicious interaction with the Payment Request API.

From the user's perspective, the following happens: The user triggers a checkout process. She decides to pay with Payment Handler A. After the authorization, the Payment Request Interface tells her that Payment Handler A is currently unavailable and that she should choose a different payment handler. The user therefore selects a different handler, Payment Handler B, and authorizes the payment for B.

Through the attack of the malicious merchant, the user is charged twice. During the attack Payment Handler A remains unaware of the second payment. Payment Handler B is unaware of the fact, that this is a retry, and therefore treats the payment as a new payment request. Payment Handler B is unable to differentiate between a regular payment request and the retry.

Figure 4.1 shows the concrete flow of this attack.

As a first step (Step $\boxed{1}$), the malicious merchant creates a regular payment request. The user follows the regular payment checkout through the Steps $\boxed{2}$ to $\boxed{11}$. Notice that in step $\boxed{8}$, the user submits the payment to an arbitrarily chosen payment handler which we call Payment Handler A. When receiving the PaymentRequestEvent, the Payment Handler triggers the transaction through communication with the payment provider of Payment Handler A.

In a regular checkout, an honest merchant would validate the transaction in step $\boxed{12}$ and complete the payment request.

In this scenario, the malicious merchant ignores the successful transaction and triggers a retry through invocation of the retry method of the PaymentResponse object (Step $\boxed{13}$). With this invocation, the merchant reports `errorFields` that are displayed in the trusted user interface of the browser. In this scenario, the merchant displays an error message telling the user that Payment Handler A is currently unavailable and that a different one has to be chosen. The user complies expecting that within a single payment process only one payment will be issued. In step $\boxed{16}$, the browser therefore submits a new PaymentRequestEvent to a different payment handler called Payment Handler B. Payment Handler B can not detect that this request was issued as a retry, and therefore, issues a second financial transaction through the payment provider of Payment Handler B. In the remaining Steps $\boxed{17}$ to $\boxed{21}$, the payment process is completed regularly. To the user, there is no indicator in these steps that the payment was triggered a second time.

4.1.3 Mitigation

This scenario can be prevented by prohibiting a switch of payment handlers during a payment request retry.

This fix is already included in the model of the Web Payment APIs.

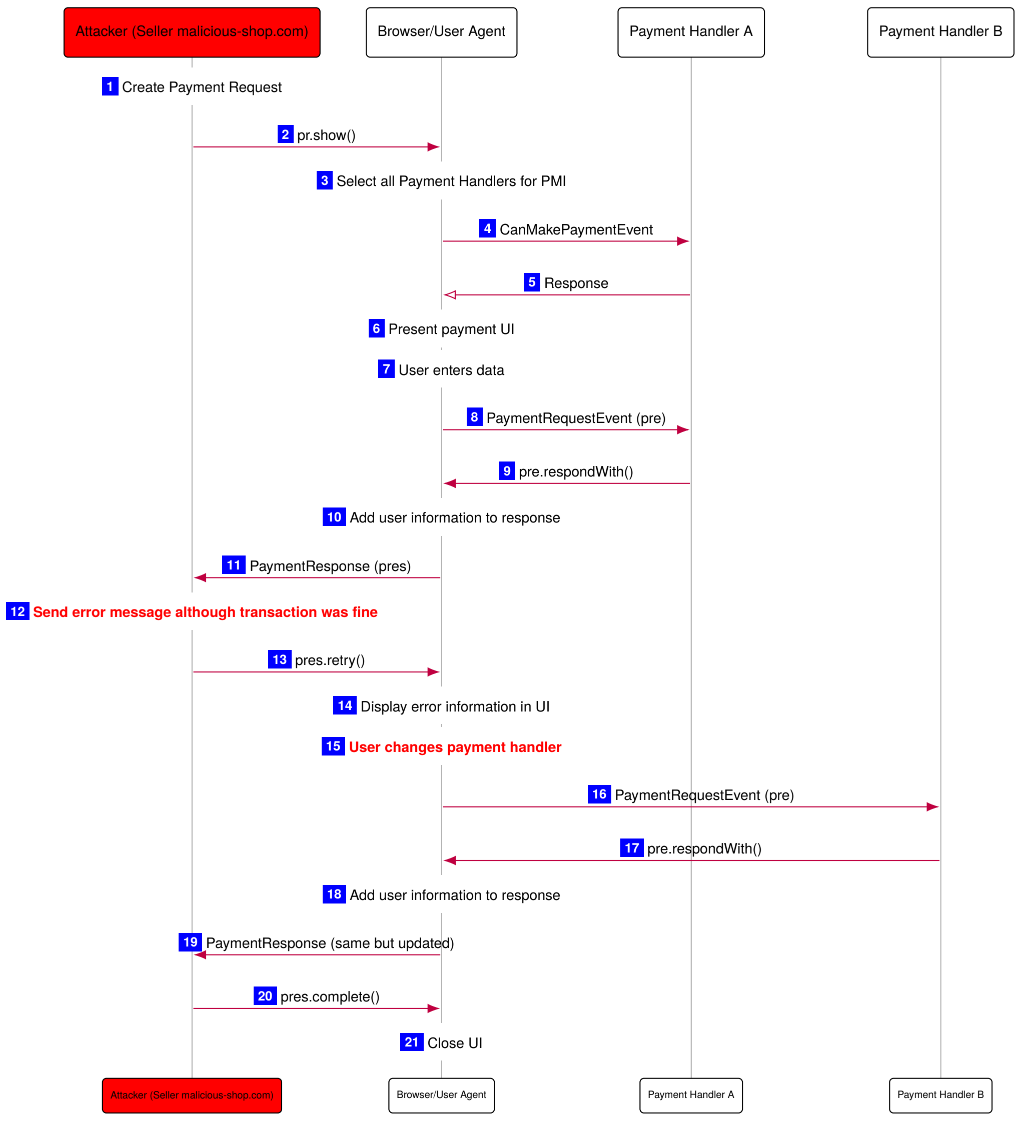
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler A
Payment Handler B
1 Create Payment Request
2 pr.show()
3 Select all Payment Handlers for PMI
4 CanMakePaymentEvent
5 Response
6 Present payment UI
7 User enters data
8 PaymentRequestEvent (pre)
9 pre.respondWith()
10 Add user information to response
11 PaymentResponse (pres)
12 Send error message although transaction was fine
13 pres.retry()
14 Display error information in UI
15 User changes payment handler
16 PaymentRequestEvent (pre)
17 pre.respondWith()
18 Add user information to response
19 PaymentResponse (same but updated)
20 pres.complete()
21 Close UI
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler A
Payment Handler B

Figure 4.1 Double charging through retry mechanism

4.1.4 Disclosure and Fix

This issue was disclosed to the Chromium Team on the 25th of November 2019. They classified the issue as of medium severity. Other security features implemented by the payment providers often only allow for interaction with validated merchants that could be blocked from interaction with their APIs as soon as their malicious behavior would be observed and reported. These security features can only prevent future exploitations, but can not mitigate the initial malicious behaviour of a merchant. The Chromium Team was very responsive upon disclosure. A patch fixing the issue has already been created and was integrated in the Chromium source base starting at version `80.x`.

4.2 Potential Issues Through Ambiguous Method Data

Payment handler specific information is passed to the payment handler through the payment request's *methodData* attribute. This happens in case of checking whether a payment can be made as a prepayment request, as well as in case of the submission of the payment [19]. Such *methodData* may include information such as a merchant identifier, from which a payment handler / payment method provider might infer the receiver of the payment, further instructions from the merchant to the payment handler, or further information on which exact types of payments are supported (e.g., MasterCard or Visa credit cards as in the specification of basic-card [22]).

During the proof of the payment integrity property, we realized that for one payment method identifier (PMI), several definitions of *methodData* for said PMI can be provided. These definitions are independent and might differ significantly. According to the Payment Request API specification [23] and the Payment Handler API specification [19], this leads to a scenario in which the payment handler obtains both definitions. Since neither of both specifications give a reasoning for this case, it is not specified how a payment handler should react to this case. If we model that the payment handler chooses their methodData non-deterministically in such cases, this can lead to situations where the receivers of a payment are chosen non-deterministically which breaks the underlying security assumptions.

We see this as a vulnerability that might lead to potential issues and maybe even attacks in the future. The impact highly depends on the behavior of the payment handler and the information that is submitted through the *methodData*. Especially when thinking about more complex named payment methods such as `basic-card`, that might highly depend on the information provided by *methodData*, this might become an issue.

Imagine that for example the inconsistency lies between the visible UI and the processing behaviour of the transaction. Furthermore, imagine two different senders would be specified in the ambiguous *methodData*. In such a case a payment handler might send the money to a different person then what was displayed in the UI.

Or imagine that a not yet existing payment method has a *methodData* value that determines whether a payment should be a one-time payment or a recurring scheduled payment. Furthermore, imagine a malicious merchant uses a deterministic behaviour of the browser and the payment handler to display a one-time payment and trigger a scheduled payment.

These issues would clearly violate the payment integrity property.

4.2.1 Assumptions

A payment handler / the browser is inconsistent in the way that it uses the *methodData* when several potentially ambiguous definitions are given. This assumption is supported by the fact that the specification does not require the payment handlers to handle such a case in a deterministic/consistent way.

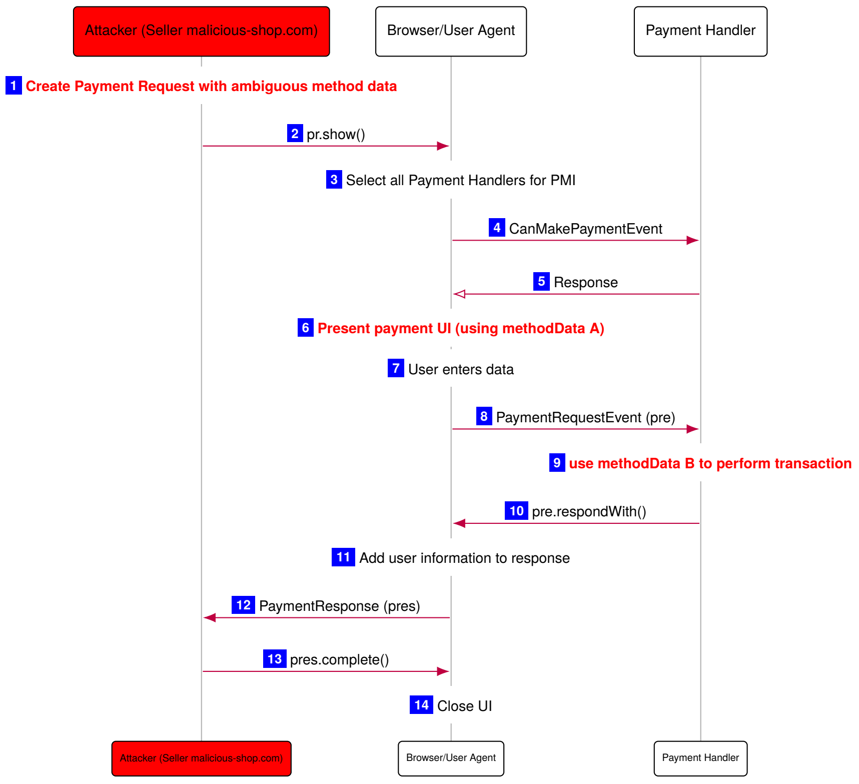
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler
1 Create Payment Request with ambiguous method data
2 pr.show()
3 Select all Payment Handlers for PMI
4 CanMakePaymentEvent
5 Response
6 Present payment UI (using methodData A)
7 User enters data
8 PaymentRequestEvent (pre)
9 use methodData B to perform transaction
10 pre.respondWith()
11 Add user information to response
12 PaymentResponse (pres)
13 pres.complete()
14 Close UI
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler

Figure 4.2 Potential issue with ambiguous methodDatas for a single payment method identifier

4.2.2 Scenario

Figure 4.2 shows the concrete flow of such a potential issue. This flow basically represents a general checkout, but with different behavior of the single parties. The attack starts with a malicious merchant that provides ambiguous method data to a payment request (Step $\boxed{1}$). Such ambiguous method data contains several definitions of data to use for a single payment method identifier. Since the specification does not define behavior in such a case, we assume that the implemented behavior of the browser or the payment handlers might be inconsistent. In the figure, we illustrate this by assuming that in Step $\boxed{6}$, the browsers UI of a payment method may rely on a different method data value (here called A) than the payment handler later on in Step $\boxed{9}$ (here called B).

4.2.3 Mitigation

We propose two potential mitigation approaches: Prohibition of ambiguous *methodData* values and specification of the value to use in such a scenario.

Prohibition of ambiguous *methodData* values, could be realized through checking whether the given payment method identifiers are specified multiple times when the payment request is constructed. In such a case the construction of a payment request would lead to an erroneous case and the abortion of the payment request. We recommend this mitigation approach since it resolves the root cause of potential issues.

Specification of the value to use, would simply mean extending the specification through a guideline, defining which value to use in such a case. In case of the paymentModifiers, the specification [23] contains exactly such a guideline, which suggests a “last-one-wins” approach. Such a guideline does not fully protect against such issues but at least may raise awareness of the issue.

Within the here presented model, the second method was used. In said cases the algorithms of the model use the first definition of a *methodData* for a given payment method identifier. We used this approach, to verify whether this second proposal does in deed resolve the issue.

4.3 Leak of Personal Data to Merchant Before Expression of Payment Intent

In a regular online shop setting, a merchant does have a need for obtaining personal data after a user did express intent to buy. To ship the product to the customer, a merchant needs the address of the customer. In the specification of the Payment Request API, this information is given to the merchant before the user did express intent. This is a vulnerability that the specification is aware of, nonetheless it might leak personal data in a way that a user does not expect.

4.3.1 Assumptions

We assume that the user expects that her information is only shared with the merchant after the payment was submitted.

4.3.2 Scenario

Figure 4.3 shows the flow of a regular shipping address change in which the described issue occurs.

In Step $\boxed{8}$, the user changes the shipping address in the payment UI. This might be enforced by initially providing no shippingOption in the payment request as described in Subsection 2.4.2. The browser afterwards provides the merchant with a PaymentRequestUpdateEvent that contains a redacted shipping address (containing e.g., the postal code) in Step $\boxed{9}$.

This is an issue since this event occurs before Step $\boxed{11}$ in which the user expresses his/her payment intent.

Additionally, the user is unaware of the fact that this communication with the merchant occurs.

4.3.3 Mitigation

In our opinion, the merchant does not have a need to obtain the shipping address in advance of the payment intent. The merchant could offer a predefined set of shipping areas and their corresponding prices. This would resolve the issue without providing the merchant with personal data before the payment intent was expressed.

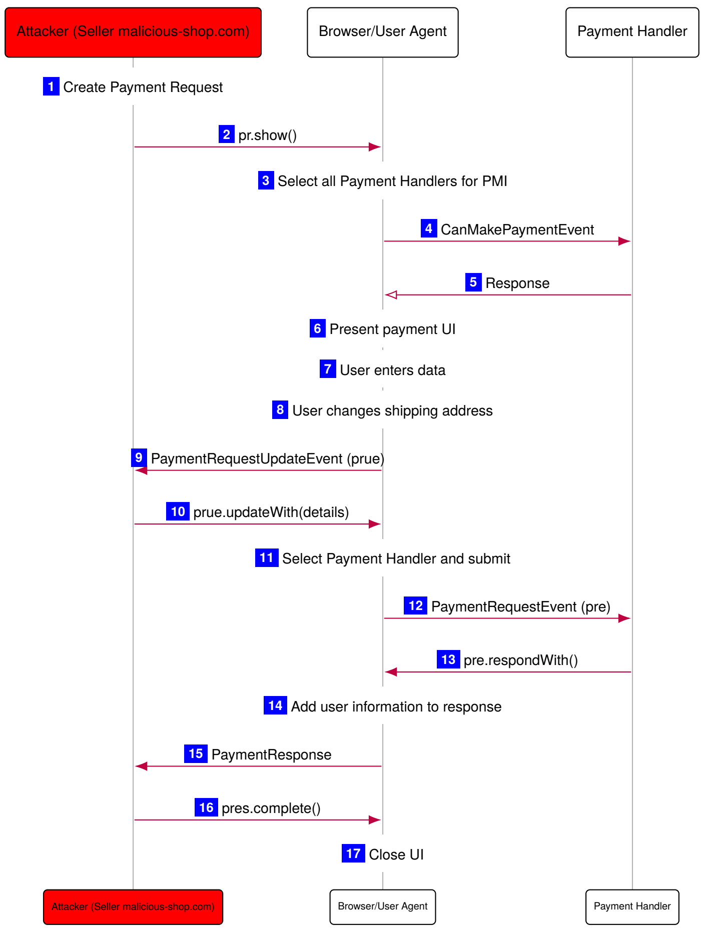
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler
1 Create Payment Request
2 pr.show()
3 Select all Payment Handlers for PMI
4 CanMakePaymentEvent
5 Response
6 Present payment UI
7 User enters data
8 User changes shipping address
9 PaymentRequestUpdateEvent (prue)
10 prue.updateWith(details)
11 Select Payment Handler and submit
12 PaymentRequestEvent (pre)
13 pre.respondWith()
14 Add user information to response
15 PaymentResponse
16 pres.complete()
17 Close UI
Attacker (Seller malicious-shop.com)
Browser/User Agent
Payment Handler

Figure 4.3 Leak of personal data to merchant before expression of payment intent

5 Conclusion and Outlook

Within this thesis, we modeled the Web Payment APIs and analyzed their security. We modeled the full environment of a web payment ecosystem: the browser APIs, the service workers, the payment handlers, the payment service provider servers, and the merchant servers. Within this model, we provided a formal definition of the relevant security properties and performed a security analysis by proving their fulfillment in the modeled system.

During the proof and the corresponding work with the APIs, we found one attack that allows a malicious merchant to trick a payer into being charged twice using a vulnerability in the retry mechanism of the payment request. Since this bug existed in Google Chrome's implementation of the Web Payment APIs, we submitted a bug report and the Chromium Team resolved the issue as recommended in this thesis.

An additional vulnerability was found that might lead to issues depending on future implementations and extensions. This vulnerability is based on the possibility of defining ambiguous methodData fields for a single payment method identifier in the payment request. In such a case, the specification does not recommend a behavior which might lead to inconsistent choices of methodData values to use.

The last vulnerability that was considered in this thesis is a privacy issue which is already mentioned in the specification itself. A merchant might get hold of some personal information (e.g., postal code) in advance of the user's expression of payment intent. Since this nonetheless is a privacy issue in our opinion, we offered a mitigation which does prevent said issue.

Furthermore, we provided useful extensions to the Web Infrastructure Model, that allow to model browser internal promise based communication and offered foundations to provide an exhaustive model of service workers.

These foundations could be a valuable basis for adding an exhaustive model of service workers to the Web Infrastructure Model. The life-cycle of service workers and their APIs offer interesting features, that might introduce potential attacks and vulnerabilities as well.

Our formal analysis did not focus on the privacy of the stakeholders. Although we mention an obvious issue, we did not formally analyze privacy considerations. During a payment process, different types of privacy relevant information are used. Such information might contain the payer's address, the payer's installed and supported payment handlers, sensitive data stored in the methodData, sensitive data stored in the payment details, a payment's total, the existence of a payment, the shipping address, the payer's phone number, the payer's email address. During the specified payment process, this data is transmitted and exchanged between the parties that interact with each other. An in-depth analysis of privacy considerations of the Web Payment APIs therefore would be an interesting future work.

A The extended Web Infrastucture Model with Web Payment APIs

In the following, we will introduce the extensions and adaptions that were added to the Web Infrastructure Model to model the Web Payment APIs.

Figure A.1 gives a general overview of how the main mechanisms of the model interact within the browser during a checkout process.

The first three columns RUNSCRIPT, RUNWORKER and PROCESSEVENT depict the three major algorithms of the extension. In a non-malicious setting, RUNSCRIPT operates in the domain of the merchant. Within this algorithm, the APIs are provided that allow a merchant's website to create and to interact with payment requests.

RUNWORKER was introduced to model the service workers running the payment handlers. In a non-malicious setting, they operate in the domain of the payment method provider.

PROCESSEVENT models the intermediary work that is performed by the browser within the checkout process. As the name suggests, it processes events within the browser.

To allow for all possible orderings of execution, the browser was extended with a set of pending events. By adding this functionality, the browser becomes a Dolev-Yao style process itself. Within the browser, different scripts, service workers and the browser itself have different states and inputs, from which they derive events (messages) that are sent to each other.

A checkout process starts with a script calling the `PR_CREATE` command (step $\boxed{1}$). Through this command, a payment request is created and added to the paymentStorage of the browser. The script obtains a payment request nonce, through which it is able to reference the payment request in the future.

By calling the `PR_SHOW` command a script can trigger the functionality of a `show()` method of a regular payment request object (step $\boxed{2}$).

Within this functionality, a CANMAKEPAYMENT event has to be delivered to each relevant payment handler. Therefore, the corresponding events are added to the pending events in step $\boxed{3}$. Since from now on, the payment intent can be submitted, an additional event is added, that later on represents the user's click on the "pay" button in the payment UI.

In the steps $\boxed{5}$ and $\boxed{6}$, the CANMAKEPAYMENT events are delivered to the service workers through their scriptinputs.

Step $\boxed{7}$ corresponds to the user's click on the "pay" button in the payment UI. The processing of this event is of special importance, since the payment request's state at this time corresponds to the payment that the user intends to perform.

As in the specifications defined, the browser then submits a PAYMENTREQUESTEVENT to the service worker of the selected payment handler (steps $\boxed{8}$ to $\boxed{10}$).

The service worker processes the payment through interaction with the payment provider server, that is not depicted in this figure. After this, the service worker calls the PAYMENTHANDLERRESPONSE command (step $\boxed{11}$), that creates a PAYMENTHANDLERRESPONSE event ($\boxed{12}$) that is delivered to the browser where it is merged with the entered user data (steps $\boxed{13}$ and $\boxed{14}$). The resulting PAYMENTRESPONSE is then delivered back to the merchants script (steps $\boxed{15}$ to $\boxed{17}$).

Finally, the merchant's script cann call the `PRES_COMPLETE` command, that resembles the `complete`() method of a payment response and completes the checkout process (step $\boxed{18}$).

The here depicted flow is not comprehensive in the used functionalities. It is only meant to offer an easier introduction of the interaction of the components of the models.

Within the presented model, modified algorithms are always presented in its total. Therefore, some passage directly originate from the Web Infrastructure Model [8]. Unmodified passages are highlighted in this light gray, while new passages are highlighted in common black.

A.1 General Remarks

To keep the model compact and readable, the abbreviation *PRN* is used within the model in some places to denote *paymentRequestNonce*.

The functions $\mathtt{secretOfID}(id)$, $\mathtt{ownerOfID}(id)$ and $\mathtt{governorOfID}(id)$ are defined analogously to [8]. There they were defined in an OAuth context. In this model the payment provider servers act as governors of identities/identity providers.

The function $\mathtt{secretOfID}(id)$ describes a bijective mapping $\mathtt{ID} \to \mathtt{Passwords}$ that is only available to the honest browser that owns the ID $b = \mathtt{ownerOfID}(id)$ and the honest payment provider $pp = \mathtt{governorOfID}(id)$.

$\mathtt{ownerOfID}(id)$ and $\mathtt{governorOfID}(id)$ are adapted accordingly as in [8].

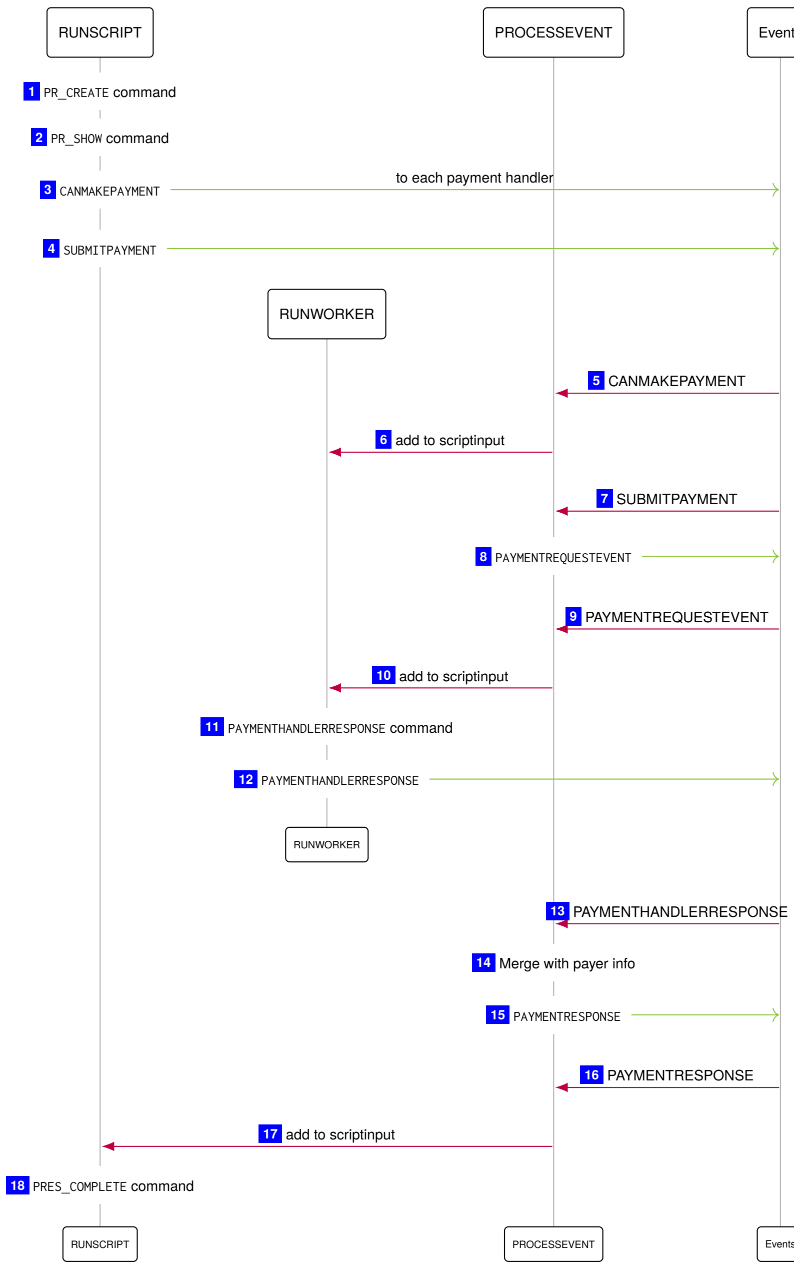

Figure A.1 General flow of Web Payments in the WIM extension

A.2 Browser

Definition A.2.1
The original definition of the set of states of a browser process $Z_{\text{Webbrowser}}$ is extended to fit the needs of the Web Payment APIs. We extend $Z_{\text{Webbrowser}}$ by the subterms serviceWorkers, paymentStorage, events and paymentIntents. A browser state is therefore defined through a term of the form:

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping,$$
$$sts, DNSaddress, pendingDNS, pendingRequests, wsConnections, rtcConnections,$$
$$\boldsymbol{serviceWorkers}, \boldsymbol{paymentStorage}, \boldsymbol{events}, \boldsymbol{paymentIntents}, isCorrupted\rangle$$

The subterms $windows, ids, secrets, cookies, localStorage, sessionStorage,$ $keyMapping, sts, DNSaddress, pendingDNS, pendingRequests, wsConnections,$ $rtcConnections$, and $isCorrupted$ are defined as in the original Web Infrastructure Model [8].

The remaining sub terms have the following form:

- *serviceWorkers $\subset^{\langle\rangle}$ ServiceWorkerRegistrations is a list of service worker registrations. This subterm stores all information concerning service workers, such as their state, their scope and their inputs.*
- *paymentStorage $\in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ stores references to payment relevant objects that may be accessed later on through scripts, service workers and the browser*
- *events $\subset^{\langle\rangle}$ $\mathcal{T}_{\mathcal{N}}$ stores a list of events that the browser may process at any time. This set of events is used to allow for arbitrary orderings of asynchronous events.*
- *paymentIntents $\in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ stores the information, which transactions were intended by the user. The nonces refer to the relevant payment requests and the terms contain the information of what totals, receiver etc. were intended in which order. The term is structured as a list of payment request objects containing the total and the receiver of all intended payments of a single payment request with the corresponding payment request nonce. This list of historical intents is necessary because through the retry mechanism an arbitrary amount of intents can be expressed within one payment request.*

An initial state s_0^b of a browser is defined as in the original WIM [8]. The new subterms are initialized as follows:

- *s_0^b.serviceWorkers $\equiv$ a non-deterministically sampled, finite set of service worker registrations. Note, that these service worker registrations can be partitioned into honest and malicious service workers by use of their trusted flag. Furthermore, the trusted flag is constrained as follows: If there exists a payment provider server that is honest and serves the scope of the serviceWorker, then the trusted flag must be $\top$. This constraint must be added to keep the configuration in accordance to the payment method manifest [21]. If a non-trusted serviceworker was served over the same scope as the trusted and honest payment provider, the whole trust construct of service workers and payment handlers would be broken directly.*
- *s_0^b.paymentStorage $\equiv \langle\rangle$*
- *s_0^b.paymentIntents $\equiv \langle\rangle$*

Placeholder	Usage
ν_{14}	Algorithm 2, lookup key for payment request
ν_{15}	Algorithm 3, placeholder for sensitive payer information
ν_{16}	Algorithm 2, lookup key for payment request events
ν_{17}	Algorithm 3, placeholder for shipping information

Table A.1 List of placeholders added for use in browser algorithms.

Table A.1 shows the newly introduced placeholders in the browser model. It extends the placeholder table of the browser as provided in [8]. The newly introduced placeholders either serve as placeholders for lookup nonces or as placeholders for sensitive information that an attacker should not be able to obtain.

Helper Functions

Given a browser state s, PaymentObjects(s) denotes the set of all pointers to objects stored in s.paymentStorage.

Given a browser state s, PaymentRequests(s) denotes the set of all pointers to objects stored in s.paymentStorage that originate in Line 96 of Algorithm 2.

Extension of Windows

The window term is extended with a boolean flag *paymentRequestShowing*. A window therefore is a term of the form $w = \langle nonce, documents, opener, paymentRequestShowing\rangle$ with *nonce*, *documents* and *opener* defined as in the original model and *paymentRequestShowing* $\in \{\top, \bot\}$. A window term is always initialized with w.*paymentRequestShowing* $= \bot$.

A.2.1 Browser Payment APIs

The main algorithm of the web browser (Algorithm 1) is only extended in a few places. Its main additions are the support for running service workers and processing the internal browser events.

Algorithm 2 models the execution of a script with the function RUNSCRIPT. Since all earlier functionalities and interfaces are still exposed to the scripts, a big part of the algorithm stays untouched.

The only old functionality that was adapted in the script is concerning the delivery of post messages. Service workers can receive post messages in the new model as well. Therefore, if not given a window nonce but a service worker nonce, the browser submits the message to the corresponding service worker.

The biggest additions in the script are the extensions of the script APIs by the Web Payment APIs. PR_CREATE, PR_SHOW, PR_CANMAKEPAYMENT, and PR_ABORT model the corresponding JavaScript functions create(), show(), canMakePayment() and abort() of a PaymentRequest. PRES_COMPLETE, and PRES_RETRY model the corresponding JavaScript functions complete() and retry() of the PaymentRequestResponse object. PR_GET models later reading of fields of a payment request if changed. PR_UPDATE_DETAILS models the updateWith() method of the PaymentRequestUpdateEvent. To keep the model simple, this method is directly exposed to the PaymentRequest in this model, without a need for a PaymentRequestUpdateEvent.

Algorithm 1 Web Browser Model: Main Algorithm.

```
Input: $\langle a, f, m \rangle, s$
 1: let $s' := s$
    ———— Check if browser is corrupted ————
 2: if $s.\mathtt{isCorrupted} \not\equiv \bot$ then
 3:     let $s'.\mathtt{pendingRequests} := \langle m, s.\mathtt{pendingRequests} \rangle$   → Collect incoming messages
 4:     let $n \leftarrow \mathbb{N}$
 5:     let $m'_1, \dots, m'_n \leftarrow d_V(s')$   → Create $n$ new messages nondeterministically.
 6:     let $a'_1, \dots, a'_n \leftarrow \mathsf{IPs}$
 7:     stop $\langle \langle a'_1, a, m'_1 \rangle, \dots, \langle a'_n, a, m'_n \rangle \rangle, s'$
    ———— Receive trigger message ————
 8: if $m \equiv \mathtt{TRIGGER}$ then
 9:     let $switch \leftarrow \{\mathtt{script}, \mathtt{worker}, \mathtt{urlbar}, \mathtt{reload}, \mathtt{forward}, \mathtt{back}, \mathtt{event}\}$
10:     let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ such that $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
          ↪ if possible; otherwise stop   → Pointer to some window.
11:     let $\overline{tlw} \leftarrow \mathbb{N}$ such that $s'.\overline{tlw}.\mathtt{documents} \neq \langle\rangle$
          ↪ if possible; otherwise stop   → Pointer to some top-level window.
12:     if $switch \equiv \mathtt{script}$ then   → Run some script.
13:         let $\overline{d} := \overline{w} +^{\langle\rangle} \mathtt{activedocument}$
14:         call $\mathsf{RUNSCRIPT}(\overline{w}, \overline{d}, s')$
15:     else if $switch \equiv \mathtt{worker}$ then   → Run some service worker.
16:         let $sw \leftarrow s'.serviceWorkers$
17:         let $\overline{sw} \leftarrow \mathcal{P}(\mathbb{N})$ such that $s'.\overline{sw}.nonce = sw.nonce$
18:         call $\mathsf{RUNWORKER}(\overline{sw}, s')$
19:     else if $switch \equiv \mathtt{urlbar}$ then   → Create some new request.
20:         let $newwindow \leftarrow \{\top, \bot\}$
21:         if $newwindow \equiv \top$ then   → Create a new window.
22:             let $windownonce := \nu_1$
23:             let $w' := \langle windownonce, \langle\rangle, \bot \rangle$
24:             let $s'.\mathtt{windows} := s'.\mathtt{windows} +^{\langle\rangle} w'$
25:         else   → Use existing top-level window.
26:             let $windownonce := s'.\overline{tlw}.nonce$
27:         let $protocol \leftarrow \{\mathtt{P}, \mathtt{S}\}$
28:         let $host \leftarrow \mathsf{Doms}$
29:         let $path \leftarrow \mathbb{S}$
30:         let $fragment \leftarrow \mathbb{S}$
31:         let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
32:         let $url := \langle \mathtt{URL}, protocol, host, path, parameters, fragment \rangle$
33:         let $req := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, host, path, parameters, \langle\rangle, \langle\rangle \rangle$
34:         call $\mathsf{HTTP\_SEND}(\langle \mathtt{REQ}, windownonce \rangle, req, url, \bot, \bot, \bot, s')$
35:     else if $switch \equiv \mathtt{reload}$ then   → Reload some document.
36:         let $url := s'.\overline{w}.\mathtt{activedocument}.\mathtt{location}$
37:         let $req := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, url.\mathtt{parameters}, \langle\rangle, \langle\rangle \rangle$
38:         let $referrer := s'.\overline{w}.\mathtt{activedocument}.\mathtt{referrer}$
39:         let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\mathtt{nonce}, s')$
40:         call $\mathsf{HTTP\_SEND}(\langle \mathtt{REQ}, s'.\overline{w}.\mathtt{nonce} \rangle, req, url, \bot, referrer, \bot, s')$
41:     else if $switch \equiv \mathtt{forward}$ then
42:         $\mathsf{NAVFORWARD}(\overline{w}, s')$
43:     else if $switch \equiv \mathtt{back}$ then
44:         $\mathsf{NAVBACK}(\overline{w}, s')$
```

```
45:         else if $\mathit{switch} \equiv \texttt{event} \wedge s'.\texttt{events} \neq \langle\rangle$ then
46:             let $e \leftarrow s'.\texttt{events}$
47:             let $s'.\texttt{events} := s'.\texttt{events} \setminus \{e\}$
48:             PROCESSEVENT($e, \overline{w}, s'$)
    ———— Change corruption status ————
49:     else if $m \equiv \texttt{FULLCORRUPT}$ then    $\rightarrow$ Request to corrupt browser
50:         let $s'.\texttt{isCorrupted} := \texttt{FULLCORRUPT}$
51:         stop $\langle\rangle, s'$
52:     else if $m \equiv \texttt{CLOSECORRUPT}$ then    $\rightarrow$ Close the browser
53:         let $s'.\texttt{secrets} := \langle\rangle$
54:         let $s'.\texttt{windows} := \langle\rangle$
55:         let $s'.\texttt{pendingDNS} := \langle\rangle$
56:         let $s'.\texttt{pendingRequests} := \langle\rangle$
57:         let $s'.\texttt{sessionStorage} := \langle\rangle$
58:         let $s'.\texttt{cookies} \subset^{\langle\rangle} \mathsf{Cookies}$ such that
              $\hookrightarrow$ $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content.session} \equiv \bot)$
59:         let $s'.\texttt{isCorrupted} := \texttt{CLOSECORRUPT}$
60:         stop $\langle\rangle, s'$
    ———— Plain-text messages ————
61:     else if $m \in \mathsf{DNSResponses}$ then    $\rightarrow$ DNS response
62:         if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs}$
                  $\hookrightarrow$ $\vee\ m.\texttt{domain} \not\equiv \pi_2(s.\texttt{pendingDNS}[m.\texttt{nonce}]).\texttt{host}$ then
63:             stop
64:         let $\langle \mathit{reference}, \mathit{message}, \mathit{url} \rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
65:         if $\mathit{url}.\texttt{protocol} \equiv \texttt{S}$ then
66:             let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{message}, \mathit{url}, \nu_3, m.\texttt{result} \rangle$
67:             let $\mathit{message} := \mathsf{enc_a}(\langle \mathit{message}, \nu_3 \rangle, s'.\texttt{keyMapping}[\mathit{message}.\texttt{host}])$
68:         else
69:             let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{message}, \mathit{url}, \bot, m.\texttt{result} \rangle$
70:         let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
71:         stop $\langle \langle m.\texttt{result}, a, \mathit{message} \rangle \rangle, s'$
72:     else if $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists \langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f \rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
              $\hookrightarrow$ such that $m.\texttt{nonce} \equiv \mathit{request}.\texttt{nonce}$ then    $\rightarrow$ Plain HTTP Response
73:         remove $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f \rangle$ from $s'.\texttt{pendingRequests}$
74:         call PROCESSRESPONSE($m, \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f, s'$)
75:     else if $m.1 \equiv \texttt{WS\_MSG} \wedge \exists \langle \mathit{reference}, \mathit{nonce}, \bot, f \rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
              $\hookrightarrow$ such that $m.\texttt{nonce} \equiv \mathit{nonce}$ then    $\rightarrow$ Plain Websocket Message
76:         call DELIVER_TO_DOC($\pi_2(\mathit{reference}), m, s'$)
    ———— Encrypted messages ————
77:     else if $\exists \langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f \rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
              $\hookrightarrow$ such that $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \texttt{HTTPResp}$ then    $\rightarrow$ Encrypted HTTP response
78:         let $m' := \mathsf{dec_s}(m, \mathit{key})$
79:         if $m'.\texttt{nonce} \not\equiv \mathit{request}.\texttt{nonce}$ then
80:             stop
81:         remove $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f \rangle$ from $s'.\texttt{pendingRequests}$
82:         call PROCESSRESPONSE($m', \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f, s'$)
83:     else if $\exists \langle \mathit{reference}, \mathit{nonce}, \mathit{key}, f \rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
              $\hookrightarrow$ such that $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \texttt{WS\_MSG}$ then    $\rightarrow$ Encrypted Websocket Message
84:         let $m' := \mathsf{dec_s}(m, \mathit{key})$
85:         if $m'.\texttt{nonce} \not\equiv \mathit{nonce}$ then
86:             stop
87:         call DELIVER_TO_DOC($\pi_2(\mathit{reference}), m', s'$)
```

```
88: else if $\exists\langle nonce, info\rangle \in^{\langle\rangle} s'.\mathtt{rtcConnections}$
        $\hookrightarrow$ such that $\pi_1(\mathsf{dec_a}(m, info.\mathtt{privkey})) \equiv \mathtt{RTC\_MSG}$ then    $\rightarrow$ WebRTC message
89:     let $m' := \mathsf{dec_a}(m, info.\mathtt{privkey})$
90:     if $m'.\mathsf{nonce} \not\equiv nonce$ then
91:         stop
92:     let $docnonce := info.\mathsf{docnonce}$
93:     call DELIVER_TO_DOC$(docnonce, m', s')$
94: stop
```

Algorithm 3 describes how the function PROCESSEVENT models the browser's event processing. The events CANMAKEPAYMENT, PAYMENTREQUESTEVENT, and PAYMENTRESPONSE are simply processed by transmitting the relevant event to the corresponding party.

PAYMENTHANDLERRESPONSE is more complex since it integrates the information provided by the payment handler through the PaymentHandlerResponse into a PaymentResponse object and submits it to the event set for further processing.

Algorithm 2 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT($\overline{w}$, $\overline{d}$, $s'$)
2:     let $tree$ := Clean($s'$, $s'.\overline{d}$)
3:     let $cookies$ := $\langle\{\langle c.\mathtt{name}, c.\mathtt{content.value}\rangle | c \in^{\langle\rangle} s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$
         $\hookrightarrow$ $\wedge\, c.\mathtt{content.httpOnly} = \bot$
         $\hookrightarrow$ $\wedge\, \left(c.\mathtt{content.secure} \implies \left(s'.\overline{d}.\mathtt{origin.protocol} \equiv \mathtt{S}\right)\right)\}\rangle$
4:     let $tlw \leftarrow s'.\mathtt{windows}$ such that $tlw$ is the top-level window containing $\overline{d}$
5:     let $sessionStorage$ := $s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, tlw.\mathtt{nonce}\rangle\right]$
6:     let $localStorage$ := $s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right]$
7:     let $secrets$ := $s'.\mathtt{secrets}\left[s'.\overline{d}.\mathtt{origin}\right]$
8:     let $R \leftarrow \mathsf{script}^{-1}(s'.\overline{d}.\mathtt{script})$
9:     let $in$ := $\langle tree, s'.\overline{d}.\mathtt{nonce}, s'.\overline{d}.\mathtt{scriptstate}, s'.\overline{d}.\mathtt{scriptinputs}, cookies,$
         $\hookrightarrow$ $localStorage, sessionStorage, s'.\mathtt{ids}, secrets\rangle$
10:    let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         $\hookrightarrow$ $cookies' \leftarrow \mathsf{Cookies}^{\nu}$,
         $\hookrightarrow$ $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         $\hookrightarrow$ $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         $\hookrightarrow$ $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         $\hookrightarrow$ $out^{\lambda}$ := $\langle state', cookies', localStorage', sessionStorage', command\rangle$
         $\hookrightarrow$ such that $(in, out^{\lambda}) \in R$
11:    let $out$ := $out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$
12:    let $s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$
         $\hookrightarrow$ := $\langle \mathsf{CookieMerge}(s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right], cookies')\rangle$
13:    let $s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right]$ := $localStorage'$
14:    let $s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, tlw.\mathtt{nonce}\rangle\right]$ := $sessionStorage'$
15:    let $s'.\overline{d}.\mathtt{scriptstate}$ := $state'$
16:    switch $command$ do
17:        case $\langle\mathtt{HREF}, url, hrefwindow, noreferrer\rangle$
18:            let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $hrefwindow$, $noreferrer$, $s'$)
19:            let $req$ := $\langle\mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, url.\mathtt{parameters}, \langle\rangle\rangle$
20:            if $noreferrer \equiv \top$ then
21:                let $referrerPolicy$ := $\mathtt{noreferrer}$
22:            else
23:                let $referrerPolicy$ := $s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
24:            let $s'$ := CANCELNAV($s'.\overline{w}'.\mathtt{nonce}$, $s'$)
25:            call HTTP_SEND($s'.\overline{w}'.\mathtt{nonce}$, $req$, $url$, $\bot$, $referrer$, $referrerPolicy$, $s'$)
26:        case $\langle\mathtt{IFRAME}, url, window\rangle$
27:            let $\overline{w}'$ := GETWINDOW($\overline{w}$, $window$, $s'$)
28:            let $req$ := $\langle\mathtt{HTTPReq}, \nu_4, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, url.\mathtt{parameters}, \langle\rangle\rangle$
29:            let $referrer$ := $s'.\overline{w}'.\mathtt{activedocument.location}$
30:            let $referrerPolicy$ := $s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
31:            let $w'$ := $\langle\nu_5, \langle\rangle, \bot\rangle$
32:            let $s'.\overline{w}'.\mathtt{activedocument.subwindows}$
                 $\hookrightarrow$ := $s'.\overline{w}'.\mathtt{activedocument.subwindows} +^{\langle\rangle} w'$
33:            call HTTP_SEND($\nu_5$, $req$, $url$, $\bot$, $referrer$, $referrerPolicy$, $s'$)
```

```
34:     case $\langle\mathtt{FORM}, url, method, data, hrefwindow\rangle$
35:         if $method \notin \{\mathtt{GET}, \mathtt{POST}\}$ then [1]
36:             stop
37:         let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $hrefwindow$, $\bot$, $s'$)
38:         if $method = \mathtt{GET}$ then
39:             let $body := \langle\rangle$
40:             let $parameters := data$
41:             let $origin := \bot$
42:         else
43:             let $body := data$
44:             let $parameters := url.\mathtt{parameters}$
45:             let $origin := s'.\overline{d}.\mathtt{origin}$
46:         let $req := \langle\mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, parameters, body\rangle$
47:         let $referrer := s'.\overline{d}.\mathtt{location}$
48:         let $referrerPolicy := s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
49:         let $s'$ := CANCELNAV($s'.\overline{w}'.\mathtt{nonce}, s'$)
50:         call HTTP_SEND($s'.\overline{w}'.\mathtt{nonce}$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
51:     case $\langle\mathtt{SETSCRIPT}, window, script\rangle$
52:         let $\overline{w}'$ := GETWINDOW($\overline{w}, window, s'$)
53:         let $s'.\overline{w}'.\mathtt{activedocument.script} := script$
54:         stop $\langle\rangle$, $s'$
55:     case $\langle\mathtt{SETSCRIPTSTATE}, window, scriptstate\rangle$
56:         let $\overline{w}'$ := GETWINDOW($\overline{w}, window, s'$)
57:         let $s'.\overline{w}'.\mathtt{activedocument.scriptstate} := scriptstate$
58:         stop $\langle\rangle$, $s'$
59:     case $\langle\mathtt{XMLHTTPREQUEST}, url, method, data, xhrreference\rangle$
60:         if $method \in \{\mathtt{CONNECT}, \mathtt{TRACE}, \mathtt{TRACK}\} \wedge xhrreference \notin \{\mathcal{N}, \bot\}$ then
61:             stop
62:         if $url.\mathtt{host} \not\equiv s'.\overline{d}.\mathtt{origin.host}$
          $\hookrightarrow$ $\vee\ url \not\equiv s'.\overline{d}.\mathtt{origin.protocol}$ then
63:             stop
64:         if $method \in \{\mathtt{GET}, \mathtt{HEAD}\}$ then
65:             let $data := \langle\rangle$
66:             let $origin := \bot$
67:         else
68:             let $origin := s'.\overline{d}.\mathtt{origin}$
69:         let $req := \langle\mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, , url.\mathtt{parameters}, data\rangle$
70:         let $referrer := s'.\overline{d}.\mathtt{location}$
71:         let $referrerPolicy := s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$
72:         call HTTP_SEND($\langle s'.\overline{d}.\mathtt{nonce}, xhrreference\rangle$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
73:     case $\langle\mathtt{BACK}, window\rangle$ [2]
74:         let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $window$, $\bot$, $s'$)
75:         NAVBACK($\overline{w}$, $s'$)
76:         stop $\langle\rangle$, $s'$
77:     case $\langle\mathtt{FORWARD}, window\rangle$
78:         let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $window$, $\bot$, $s'$)
79:         NAVFORWARD($\overline{w}$, $s'$)
80:         stop $\langle\rangle$, $s'$
```

```
81:     case $\langle \texttt{CLOSE}, window \rangle$
82:         let $\overline{w}'$ := GETNAVIGABLEWINDOW($\overline{w}$, $window$, $\bot$, $s'$)
83:         remove $s'.\overline{w}'$ from the sequence containing it
84:         stop $\langle\rangle$, $s'$
85:     case $\langle \texttt{POSTMESSAGE}, window, message, origin \rangle$
86:         let $\overline{w}' \leftarrow$ Subwindows($s'$) such that $s'.\overline{w}'.\texttt{nonce} \equiv window$
87:         if $\exists \overline{j} \in \mathbb{N}$ such that $s'.\overline{w}'.\texttt{documents}.\overline{j}.\texttt{active} \equiv \top$
            $\hookrightarrow$ $\wedge(origin \not\equiv \bot \implies s'.\overline{w}'.\texttt{documents}.\overline{j}.\texttt{origin} \equiv origin)$ then
88:             let $s'.\overline{w}'.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
                $\hookrightarrow$ := $s'.\overline{w}'.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
                $\hookrightarrow$ $+^{\langle\rangle}$ $\langle \texttt{POSTMESSAGE}, s'.\overline{w}.\texttt{nonce}, s'.\overline{d}.\texttt{origin}, message \rangle$
89:         if $\exists \overline{j} \in \mathbb{N}$ such that $s'.\texttt{serviceWorkers}.\overline{j}.\texttt{nonce} \equiv window$ then
90:             let $s'.\texttt{serviceWorkers}.\overline{j}.scriptinputs$
                $\hookrightarrow$ := $s'.\texttt{serviceWorkers}.\overline{j}.scriptinputs$
                $\hookrightarrow$ $+^{\langle\rangle}$ $\langle \texttt{POSTMESSAGE}, s'.serviceWorkers.\overline{j}.nonce, \langle\rangle, message \rangle$
91:         stop $\langle\rangle$, $s'$

                         Extension with Payment APIs
92:     case $\langle \texttt{PR\_CREATE}, methodData, details, options \rangle$
93:         if $\neg(methodData \in \text{MethodDatas})$ then
94:             stop $\langle\rangle$, $s'$
95:         let $paymentRequest$ := $\langle \texttt{PAYMENTREQUEST}, \nu_{14}, s'.\overline{d}.\texttt{nonce}, methodData, details, options,$
            $\hookrightarrow$ $\langle\rangle, CR, \bot, \langle\rangle\rangle$
96:         let $s'.\overline{w}.paymentStorage[\nu_{14}] := paymentRequest$   $\rightarrow$ Create Payment Request
97:         let $s'.\overline{w}'.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
            $\hookrightarrow$ := $s'.\overline{w}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
            $\hookrightarrow$ $+^{\langle\rangle}$ $paymentRequest$   $\rightarrow$ Inform script of how to access payment request
98:         stop $\langle\rangle$, $s'$
99:     case $\langle \texttt{PR\_SHOW}, paymentRequestNonce, detailsUpdate \rangle$
100:        let $\overline{pr}'$ := PaymentRequests(s) such that $s'.\overline{pr}'.PRN \equiv paymentRequestNonce$
101:        if $s'.\overline{pr}'.state \neq CR$ then
102:            stop $\langle\rangle$, $s'$
103:        if $s'.\overline{w}.paymentRequestShowing = \top$ then
104:            let $s'.\overline{pr}'.\texttt{state} := CL$
105:            stop $\langle\rangle$, $s'$
106:        let $s'.\overline{pr}'.\texttt{state} := IN$
107:        let $s'.\overline{w}.\texttt{paymentRequestShowing} := \top$
108:        let $handlers := \langle\rangle$
109:        for each $\langle pmi, receiver, paymentIdentifier \rangle \in \overline{pr}'.\texttt{methodData}$ do
110:            $handlers = handlers +^{\langle\rangle}$ GET_PAYMENT_HANDLERS($pmi$, $s'$)
111:        for each handler $\in handlers$ do
112:            $\overline{s}'.\texttt{events} = \overline{s}'.\texttt{events} +^{\langle\rangle} \langle \texttt{CANMAKEPAYMENT}, \texttt{handler.nonce},$
            $\hookrightarrow$ $\langle \overline{w}'.\texttt{activedocument}.location.host, \overline{w}'.\texttt{activedocument}.location.protocol \rangle,$
            $\hookrightarrow$ $\langle \overline{d}'.location.host, \overline{d}'.location.protocol \rangle, methodData \rangle$   $\rightarrow$ Submits data to pay-
ment handlers
113:        let $handler \leftarrow handlers$
114:        if $detailsUpdate \neq \langle\rangle$ then
115:            let $s'.\overline{pr}'.\texttt{updating} := \top$
116:            $\overline{s}'.\texttt{events} = \overline{s}'.\texttt{events} +^{\langle\rangle} \langle \texttt{PR\_UPDATE\_DETAILS}, s'.\overline{pr}'.\texttt{paymentRequestNonce},$
        $\hookrightarrow$ $detailsUpdate \rangle$
```

```
117:         if $handler$.trusted $\neq \top$ then
118:             $\overline{s'}$.events = $\overline{s'}$.events $+^{\langle\rangle}$ $\langle$SUBMITPAYMENT, $s'.\overline{pr'}$.paymentRequestNonce,
             $\hookrightarrow$ $handler$.handlerNonce$\rangle$
119:         stop $\langle\rangle$, $s'$
120:     case $\langle$PR_CANMAKEPAYMENT, $paymentRequestNonce\rangle$
121:         let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
122:         if $s'.\overline{pr'}$.state $\neq CR$ then
123:             stop $\langle\rangle$, $s'$
124:         let $handlers$ := $\langle\rangle$
125:         for each $\langle pmi, receiver, paymentIdentifier\rangle \in \overline{pr'}$.methodData do
126:             $handlers$ =$handlers$ $+^{\langle\rangle}$ GET_PAYMENT_HANDLERS($\overline{w}$, $pmi$, $s'$)
127:         if $handlers \neq \langle\rangle$ then
128:             let $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
             $\hookrightarrow$ := $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
             $\hookrightarrow$ $+^{\langle\rangle}$ $\langle$CANMAKEPAYMENTRESPONSE, $paymentRequestNonce$, $\top\rangle$    $\rightarrow$ Let script know that
     payment handler is available
129:         else
130:             let $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
             $\hookrightarrow$ := $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
             $\hookrightarrow$ $+^{\langle\rangle}$ $\langle$CANMAKEPAYMENTRESPONSE, $paymentRequestNonce$, $\bot\rangle$    $\rightarrow$ Let script know that
     payment handler is not available
131:         stop $\langle\rangle$, $s'$
132:     case $\langle$PR_ABORT, $paymentRequestNonce\rangle$
133:         let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
134:         if $s'.\overline{pr'}$.state $\neq IN$ then
135:             stop $\langle\rangle$, $s'$
136:         if $s'.\overline{pr'}$.responseNonce $\neq \langle\rangle$ then
137:             stop $\langle\rangle$, $s'$
138:         let $s'.\overline{pr'}$.state := $CL$
139:         let $\overline{w}.paymentRequestShowing := \bot$
140:         stop $\langle\rangle$, $s'$
141:     case $\langle$PRES_COMPLETE, $paymentresponse\rangle$
142:         let $\overline{pres'}$ := PaymentObjects(s) such that $s'.\overline{pres'}.paymentresponse \equiv paymentresponse$
143:         if $s'.\overline{pres'}.complete \equiv \top$ then
144:             stop $\langle\rangle$, $s'$
145:         let $s'.\overline{pres'}.complete := \top$
146:         let $s'.\overline{w}.paymentRequestShowing := \bot$
147:     case $\langle$PRES_RETRY, $paymentresponse, errorFields\rangle$
148:         let $\overline{pres'}$ := PaymentObjects(s) such that $s'.\overline{pres'}.paymentresponse \equiv paymentresponse$
149:         if $s'.\overline{pres'}.complete \equiv \top$ then
150:             stop $\langle\rangle$, $s'$
151:         let $pr$ := $s'$.paymentStorage[$s'.\overline{pres'}$.paymentRequestNonce]
152:         let $pr.state := IN$
153:         let $handlers$ := $\langle\rangle$
154:         for each $\langle pmi, data\rangle \in \overline{pr'}$.methodData do
155:             $handlers$ =$handlers$ $+^{\langle\rangle}$ GET_PAYMENT_HANDLERS($pmi$, $s'$)
```

```
156:            let $\mathit{handler} := \mathit{handlers}$ such that $\mathit{handler}.\mathit{handlerNonce} \equiv s'.\overline{pres'}.\mathit{handlerNonce}$
                    $\hookrightarrow$ if possible, otherwise stop $\langle\rangle, s'$
157:            if $\mathit{handler}.\mathtt{trusted} \neq \top$ then
158:                $\overline{s'}.\mathtt{events} = \overline{s'}.\mathtt{events} +^{\langle\rangle} \langle\mathtt{SUBMITPAYMENT}, \mathit{pr}.\mathtt{paymentRequestNonce},$
                    $\hookrightarrow$ $\mathit{handler}.\mathtt{handlerNonce}\rangle$
159:            stop $\langle\rangle, s'$
160:        case $\langle\mathtt{PR\_GET}, \mathit{paymentnonce}\rangle$   $\rightarrow$ Get Payment Request or Response Object
161:            let $\overline{pr'} := \mathtt{PaymentObjects}(\mathtt{s})$ such that $\pi_2(s'.\overline{pr'}) \equiv \mathit{paymentnonce}$
162:            let $s'.\overline{w}.\mathtt{documents}.\overline{j}.\mathtt{scriptinputs}$
                    $\hookrightarrow$ $:= s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{scriptinputs} +^{\langle\rangle} s'.\overline{pr'}$   $\rightarrow$ Let script get status of payment
    request
163:        case $\langle\mathtt{PR\_UPDATE\_DETAILS}, \mathit{paymentRequestNonce}, \mathit{details}\rangle$   $\rightarrow$ According to spec, updateDetails
    can only occure as a reaction to a PaymentRequestUpdateEvent. This is simplified in the model.
164:            let $\overline{pr'} := \mathtt{PaymentObjects}(\mathtt{s})$ such that $s'.\overline{pr'}.\mathit{PRN} \equiv \mathit{PRN}$
165:            if $s'.\overline{pr'}.\mathtt{state} \neq IN$ then
166:                stop $\langle\rangle, s'$
167:            let $s'.\overline{pr'}.\mathtt{updating} := \top$
168:            $\overline{s'}.\mathtt{events} = \overline{s'}.\mathtt{events} +^{\langle\rangle} \langle\mathtt{PR\_UPDATE\_DETAILS}, s'.\overline{pr'}.\mathtt{paymentRequestNonce}, \mathit{details}\rangle$
169:        case else
170:            stop
```

Algorithm 3 Web Browser Model: Process an event.

```
1:  function PROCESSEVENT($e$,$\overline{w}$, $s'$)
2:      switch $e$ do
    ——— PR show - preflight request whether payment can be made ———
3:          case $\langle$CANMAKEPAYMENT, handlerNonce, $topOrigin$, $paymentRequestOrigin$, $methodData\rangle$
4:              let $\overline{sw'}$ := $\mathcal{P}(\mathbb{N})$ such that $s'.\overline{sw'}.nonce \equiv handlerNonce$
5:              let $s'.\overline{sw'}.scriptinputs$ := $s'.\overline{sw'}.scriptinputs +^{\langle\rangle} e$   → Pass canmakepayment request event to worker
6:              stop $\langle\rangle$, $s'$   → Result is modeled as being independent of reply of script

    ——— User accepts the payment request algorithm ———
7:          case $\langle$SUBMITPAYMENT, $paymentRequestNonce$, $handlerNonce\rangle$
8:              let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
9:              if $s'.\overline{pr'}$.updating = $\top$ then
10:                 stop $\langle\rangle$, $s'$
11:             if $s'.\overline{pr'}.state \neq IN$ then
12:                 stop $\langle\rangle$, $s'$
13:             let $handlers$ := $\langle\rangle$
14:             for each $\langle pmi, receiver, paymentIdentifier\rangle \in \overline{pr'}$.methodData do
15:                 $handlers$ =$handlers$ $+^{\langle\rangle}$ GET_PAYMENT_HANDLERS($pmi$, $s'$)
16:             let $handler$ := $\underline{handlers}$ such that $handler.nonce \equiv handlerNonce$
17:             let $total$ := $s'.\overline{pr'}$.details.total
18:             let $modifiers$ := $s'.\overline{pr'}$.details.modifiers   → Abstraction: pass all modifiers
19:             let $requestBillingAddress \leftarrow \{\bot, \top\}$
20:             let $instrument \leftarrow handler$.instruments
21:             let $instrumentKey$ := $instrument$.instrumentKey
22:             let $methodData$ := union of terms $i \in \mathbb{N} : \pi_i(s'.\overline{pr'}.methodData)$ for which
                    ↪ $\pi_i(s'.\overline{pr'}.methodData)$.supportedMethods $\equiv instrument$.enabledMethods
23:             let $pre$ := $\langle$PAYMENTREQUESTEVENT, $\nu_{16}$, paymentRequestNonce, $handler$.nonce, $methodData$,
                    ↪ $total$, $modifiers$, $instrumentKey$, $requestBillingAddress\rangle$
24:             let $s'.\overline{w}.paymentStorage[\nu_{16}]$:= $pre$   → Make event available for later use
25:             $\overline{s'}$.events = $\overline{s'}$.events $+^{\langle\rangle}$ $pre$   → User selects a payment handler at a later time
26:             let $paymentIntent$ := $s'.\overline{pr'}$
27:             let $paymentIntent$.methodData := $methodData$
28:             $\overline{s'}$.paymentIntents[$paymentRequestNonce$] = $paymentIntent$
                    ↪ $+^{\langle\rangle}\overline{s'}$.paymentIntents[$paymentRequestNonce$]
29:             stop $\langle\rangle$, $s'$

    ——— Payment Handler is selected and processes request ———
30:         case $\langle$PAYMENTREQUESTEVENT, paymentRequestEvent, paymentRequestNonce, handlerNonce,
                ↪ $methodData$, $total$, $modifiers$, $instrumentKey$, $requestBillingAdress\rangle$
31:             let $\overline{sw'}$ := $\mathcal{P}(\mathbb{N})$ such that $s'.\overline{sw'}.nonce \equiv handlerNonce$
32:             let $s'.\overline{sw'}.scriptinputs$ := $s'.\overline{sw'}.scriptinputs +^{\langle\rangle} e$   → Pass payment request event to worker
33:             stop $\langle\rangle$, $s'$
```

```
——— Payment handler's response is merged with relevant data ———
case $\langle$PAYMENTHANDLERRESPONSE, paymentRequestEvent, paymentRequestNonce, handlerNonce,
    $\hookrightarrow$ $methodName, details\rangle$
    $\rightarrow$ See: Respond to PaymentRequest Algorithm [19]
    let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
    let $\overline{pre'}$ := $s'.paymentStorage[paymentRequestEvent]$
    if $methodName \notin^{\langle\rangle} \overline{pre'}$.methodData then
        stop $\langle\rangle$, $s'$
    $\rightarrow$ See: User accepts the Payment Request algorithm (adapted) [23]
    if $\overline{pr'}.updating = \top$ then
        stop $\langle\rangle$, $s'$
    if $\overline{pr'}.state \neq IN$ then
        stop $\langle\rangle$, $s'$
    let $handler \leftarrow s'$.serviceWorkers such that $s'.serviceWorkers.nonce \equiv handlerNonce$
    let $shippingAddress := \langle\rangle$
    if $\overline{pr'}$.options.requestShippingAddress = $\top$ then
        $shippingAdress = \nu_{17}$
    let $shippingOption := \langle\rangle$
    if $\overline{pr'}$.options.requestShipping = $\top$ then
        $shippingOption = \overline{pr'}$.shippingOption
    let $payerInfo := \langle\rangle$
    if $\overline{pr'}$.options.requestPayerInfo = $\top$ then
        $payerInfo = \nu_{15}$    $\rightarrow$ Any payer specific info (phone, name, email)
    let $responceNonce := \nu_{16}$
    if $s'.\overline{pr'}$.responseNonce $\neq \langle\rangle$ then
        $responceNonce = s'.\overline{pr'}$.responseNonce    $\rightarrow$ Use old response if retry
    let $response$ := $\langle$PAYMENTRESPONSE, $responseNonce$, paymentRequestNonce, $handlerNonce$,
    $\hookrightarrow$ $methodName, details, shippingAddress, shippingOption, payerInfo, \bot\rangle$
    let $s'.paymentStorage[response$.paymentResponse$]$ := $response$
    $\overline{s'}$.events = $\overline{s'}$.events $+^{\langle\rangle}$ $response$    $\rightarrow$ Create PAYMENTRESPONSE Event
    stop $\langle\rangle$, $s'$
——— Payment Response is submitted to script in user agent ———
case $\langle$PAYMENTRESPONSE, paymentRequestNonce, $methodName, details, shippingAddress,$
    $\hookrightarrow$ $shippingOption, payerName, payerEmail, payerPhone\rangle$
    let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
    DELIVER_TO_DOC($s'.\overline{d}$.nonce, $e$, $s'$)
    stop $\langle\rangle$, $s'$

——— PR update details ———
case $\langle$PR_UPDATE_DETAILS, paymentRequestNonce, $details\rangle$
    let $\overline{pr'}$ := PaymentObjects(s) such that $s'.\overline{pr'}.PRN \equiv PRN$
    let $s'.\overline{pr'}$.details := $details$
    let $s'.\overline{pr'}$.updating := $\bot$
    stop $\langle\rangle$, $s'$
```

A.2.2 Service Workers/Payment Handlers

Algorithm 4 with its function `RUNWORKER` models how service workers are executed. The algorithm is very similar to Algorithm 2 with its function `RUNSCRIPT`. The biggest difference lies within the non-existence of a relevant document and the available command set.

The six commands that are available are `PAYMENTHANDLERRESPONSE`, `SET_PAYMENTMANAGER`, `GET_PAYMENTMANAGER`, `XMLHTTPREQUEST`, `POSTMESSAGE`, and `OPEN_WINDOW`.

Their basic functionalities are the following:

`PAYMENTHANDLERRESPONSE` is called by a script to submit a PaymentHandlerResponse to the payment issuer. By doing so, it signals the completion of its processing of the PaymentRequestEvent.

`SET_PAYMENTMANAGER` A service worker can determine which payment instruments it does want to support. This command is used to perform such updates.

`GET_PAYMENTMANAGER` A service worker can get a list of its supported payment instruments. This command offers such functionality.

`XMLHTTPREQUEST` as in `RUNSCRIPT`.

`POSTMESSAGE` posts a message to a window or another service worker via post message.

`OPEN_WINDOW` allows a payment handler to open a window for further user interaction.

Algorithm 5 defines a helper function `GET_PAYMENT_HANDLERS` that returns all relevant payment handlers for a given payment method identifier.

To model a payment handler, Algorithm 6 was designed. It models a payment handler that upon receiving a PaymentRequestEvent opens a window through which it obtains an authentication token. With this token it can issue a payment and craft a payment handler response.

The function `DELIVER_TO_DOC` in Algorithm 7 is extended to be able to deliver messages as well to service workers.

Documents of the same scope as to which a service worker is registered have a reference to these service workers. This is provided through extending Algorithm 8 by a few lines and using the new Algorithm 9 with its function `GET_SWS`. It returns a set of relevant service workers for a given URL.

Algorithm 4 Web Browser Model: Execute a Service Worker.

```
1: function RUNWORKER($\overline{sw}$, $s'$)
2:     let $swOrigin := \langle s'.\overline{sw}.\mathtt{scope.host}, s'.\overline{sw}.\mathtt{scope.protocol}\rangle$
3:     let $cookies := \langle\{\langle c.\mathtt{name}, c.\mathtt{content.value}\rangle | c \in^{\langle\rangle} s'.\mathtt{cookies}\,[s'.\overline{sw}.\mathtt{scope.host}]$
         ↪ $\wedge\, c.\mathtt{content.httpOnly} = \bot$
         ↪ $\wedge\, (c.\mathtt{content.secure} \implies (s'.\overline{sw}.\mathtt{scope.protocol} \equiv \mathtt{S}))\}\rangle$
4:     let $localStorage := s'.\mathtt{localStorage}\,[swOrigin]$
5:     let $secrets := s'.\mathtt{secrets}\,[swOrigin]$
6:     let $R \leftarrow \mathsf{script}^{-1}(s'.\overline{sw}.\mathtt{script})$
7:     let $in := \langle s'.\overline{sw}.\mathtt{nonce}, s'.\overline{sw}.\mathtt{scriptstate}, s'.\overline{sw}.\mathtt{scriptinputs}, cookies,$
         ↪ $localStorage, s'.\mathtt{ids}, secrets, swOrigin\rangle$
8:     let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         ↪ $cookies' \leftarrow \mathsf{Cookies}^{\nu}$,
         ↪ $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         ↪ $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
         ↪ $out^{\lambda} := \langle state', cookies', localStorage', command\rangle$
         ↪ such that $(in, out^{\lambda}) \in R$
9:     let $out := out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$
10:    let $s'.\mathtt{cookies}\,[\overline{sw}.\mathtt{scope.host}]$
         ↪ $:= \langle \mathsf{CookieMerge}(s'.\mathtt{cookies}\,[\overline{sw}.\mathtt{scope.host}], cookies')\rangle$
11:    let $s'.\mathtt{localStorage}\,[swOrigin] := localStorage'$
12:    let $\overline{sw}.\mathtt{scriptstate} := state'$
13:    switch $command$ do
14:        case $\langle \mathtt{PAYMENTHANDLERRESPONSE}, \mathtt{paymentRequestNonce}, \mathtt{handlerNonce}, methodName, details\rangle$
15:            let $\overline{pre'} \leftarrow \mathbb{N}$ such that $\pi_1(\pi_{\overline{pre'}}(s'.\overline{sw}.scriptinputs)) = \mathtt{PAYMENTREQUESTEVENT}$
                 ↪ $\wedge s'.\overline{sw}.scriptinputs.\mathtt{paymentRequestNonce} \equiv paymentRequestNonce$
                 ↪ if possible, otherwise stop $\langle\rangle, s'$
16:            $\overline{s'}.\mathtt{events} = \overline{s'}.\mathtt{events} +^{\langle\rangle} \langle \mathtt{PAYMENTHANDLERRESPONSE}, \mathtt{paymentRequestNonce},$
                 ↪ $\mathtt{handlerNonce}, methodName, details\rangle$   → Create PAYMENTHANDLERRESPONSE Event
17:            stop $\langle\rangle$, $s'$
18:        case $\langle \mathtt{SET\_PAYMENTMANAGER}, newValue\rangle$
19:            let $s'.\overline{sw}.paymentManager := newValue$
20:            stop $\langle\rangle$, $s'$
21:        case $\langle \mathtt{GET\_PAYMENTMANAGER}\rangle$
22:            let $s'.\overline{sw}.scriptinputs := s'.\overline{sw}.paymentManager$
23:            stop $\langle\rangle$, $s'$
24:        case $\langle \mathtt{XMLHTTPREQUEST}, url, method, data, xhrreference\rangle$   → XmlHttpRequests as in scripts but for workers
25:            if $method \in \{\mathtt{CONNECT}, \mathtt{TRACE}, \mathtt{TRACK}\} \wedge xhrreference \notin \{\mathcal{N}, \bot\}$ then
26:                stop
27:            if $url.\mathtt{host} \not\equiv s'.\overline{sw}.\mathtt{scope.host}$
                 ↪ $\vee\, url \not\equiv s'.\overline{sw}.\mathtt{scope.protocol}$ then
28:                stop
29:            if $method \in \{\mathtt{GET}, \mathtt{HEAD}\}$ then
30:                let $data := \langle\rangle$
31:                let $origin := \bot$
32:            else
33:                let $origin := \langle s'.\overline{sw}.\mathtt{scope.protocol}, s'.\overline{sw}.\mathtt{scope.host}\rangle$
34:            let $req := \langle \mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, , url.\mathtt{parameters}, data\rangle$
35:            let $referrer := s'.\overline{sw}.\mathtt{scope}$
36:            let $referrerPolicy := \mathtt{noreferer}$
37:            call HTTP_SEND($\langle s'.\overline{sw}.\mathtt{nonce}, xhrreference\rangle$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
```

```
38:         case $\langle \texttt{POSTMESSAGE}, window, message, origin\rangle$
39:             let $\overline{w}' \leftarrow$ Subwindows$(s')$ such that $s'.\overline{w}'$.nonce $\equiv$ window
40:             if $\exists \overline{j} \in \mathbb{N}$ such that $s'.\overline{w'}$.documents.$\overline{j}$.active $\equiv \top$
                ↪ $\wedge(origin \not\equiv \bot \implies s'.\overline{w'}$.documents.$\overline{j}$.origin $\equiv origin)$ then
41:                 let $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
                    ↪ := $s'.\overline{w'}$.documents.$\overline{j}$.scriptinputs
                    ↪ $+^{\langle\rangle}$ $\langle \texttt{POSTMESSAGE}, s'.\overline{w}$.nonce$, s'.\overline{d}$.origin$, message\rangle$
42:             if $\exists \overline{j} \in \mathbb{N}$ such that $s'$.paymentStorage.$\overline{j}$.handlerNonce $\equiv$ window then
43:                 let $s'$.serviceWorkers.$\overline{j}.scriptinputs$
                    ↪ := $s'$.serviceWorkers.$\overline{j}.scriptinputs$
                    ↪ $+^{\langle\rangle}$ $\langle \texttt{POSTMESSAGE}, s'.serviceWorkers.\overline{j}.handlerNonce, \langle\rangle, message\rangle$
44:             stop $\langle\rangle$, $s'$
45:         case $\langle \texttt{OPEN\_WINDOW}, paymentRequestEvent, windownonce, url\rangle$ $\rightarrow$ Allow payment handler to open window
46:             if $\pi_1(s'$.paymentStorage$[paymentRequestEvent]) = \texttt{PAYMENTREQUESTEVENT}$ then
47:                 let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
48:                 let $s'$.windows := $s'$.windows $+^{\langle\rangle}$ $w'$
49:                 let req := $\langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, url$.host$, url$.path$, url$.parameters$, \langle\rangle, \langle\rangle\rangle$
50:                 call HTTP_SEND($\langle \texttt{REQ}, windownonce\rangle$, req, url, $\bot$, $\bot$, $\bot$, $s'$)
```

Algorithm 5 Web Browser Model: Process script interaction with Payment APIs

```
1: function GET_PAYMENT_HANDLERS($pmi$, $s'$)
2:     let $handlers := \{\}$
3:     for each handler $\in s'.serviceWorkers$ do
4:         for each instrument $\in$ handler.paymentManager.instruments do
5:             if instrument.$method = pmi$ then
6:                 let $handlers := handlers \cup \{$handler$\}$
       return $handlers$
```

Algorithm 6 Relation of *script_default_payment_handler*

```
Input: $\langle handlerNonce, scriptstate, scriptinputs, cookies, localStorage, secrets, swOrigin \rangle$ → Script that models the behaviour of a payment handler. Reacts to payment request event and communicates to a given endpoint of the payment service. Crafts a payment handler response.
 1: let $switch \leftarrow \{\mathtt{request}, \mathtt{pay}, \mathtt{craft}, \mathtt{xmlhttp}, \mathtt{setmgr}, \mathtt{getmgr}\}$
 2: if $switch \equiv \mathtt{request}$ then    → Process payment request event.
 3:     let $\overline{pre'} \leftarrow \mathbb{N}$ such that $\pi_1(\pi_{\overline{pre'}}(scriptinputs)) = \mathtt{PAYMENTREQUESTEVENT}$ if possible, otherwise
          ↪ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$    → Choose a random payment request event to request token for
 4:     let $url \leftarrow \langle \mathtt{URL}, S, swOrigin.\mathtt{host}, \mathtt{/index}, \langle\rangle, \langle\rangle\rangle$
 5:     let $command := \langle \mathtt{OPEN\_WINDOW}, \pi_{\overline{pre'}}(scriptinputs), \nu_{18}, url\rangle$    → Open window for user authentication
 6:     stop $\langle s, cookies, localStorage, command\rangle$
 7: else if $switch \equiv \mathtt{pay}$ then    → Craft payment handler response
 8:     let $\overline{pre'} \leftarrow \mathbb{N}$ such that $\pi_1(\pi_{\overline{pre'}}(scriptinputs)) = \mathtt{PAYMENTREQUESTEVENT}$ if possible, otherwise
          ↪ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$    → Reply to a random payment request event
 9:     let $url \leftarrow \langle \mathtt{URL}, \mathtt{S}, swOrigin.\mathtt{host}, \mathtt{/pay}, \langle\rangle, \langle\rangle\rangle$
10:     let $total := scriptinputs.\overline{pre'}.\mathtt{total}$
11:     let $tokenMessage$ such that $\pi_1(token) = \mathtt{POSTMESSAGE} \wedge token \in scriptinputs$ if possible; otherwise
          ↪ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$
12:     let $token := \pi_3(tokenMessage)$
13:     let $methodData := \pi_1(scriptinputs.\overline{pre'}.\mathtt{methodData})$    → Use first methodData definition even if several may be given
14:     let $receiver := methodData.\mathtt{receiver}$
15:     let $paymentIdentifier := methodData.\mathtt{paymentIdentifier}$
16:     let $paymentrequestnonce := scriptinputs.\overline{pre'}.\mathtt{paymentRequestNonce}$
17:     let $command := \langle \mathtt{XMLHTTPREQUEST}, url', \mathtt{POST},$
          ↪ $\langle token, receiver, total, paymentrequestnonce, paymentIdentifier\rangle, \perp\rangle$
18:     stop
19: else if $switch \equiv \mathtt{craft}$ then    → Craft payment handler response
20:     let $\overline{pre'} \leftarrow \mathbb{N}$ such that $\pi_1(\pi_{\overline{pre'}}(scriptinputs)) = \mathtt{PAYMENTREQUESTEVENT}$    → Reply to a random payment request event
21:     let $methodName \leftarrow \mathcal{T}_{\mathcal{N}}$
22:     let $details \leftarrow \mathcal{T}_{\mathcal{N}}$
23:     let $command := \langle \mathtt{PAYMENTHANDLERRESPONSE}, \overline{pre'}.\mathtt{paymentRequestNonce}, \mathtt{handlerNonce},$
              ↪ $methodName, details\rangle$
24:     stop $\langle s, cookies, localStorage, command\rangle$
25: else if $switch \equiv \mathtt{setmgr}$ then    → Set Payment Manager
26:     let $newValue \leftarrow \mathcal{T}_{\mathcal{N}}$
27:     let $command := \langle \mathtt{SET\_PAYMENTMANAGER}, newValue\rangle$
28:     stop $\langle s, cookies, localStorage, command\rangle$
29: else if $switch \equiv \mathtt{getmgr}$ then    → Get Payment Manager
30:     let $command := \langle \mathtt{GET\_PAYMENTMANAGER}\rangle$
31:     stop $\langle s, cookies, localStorage, command\rangle$
32: else if $switch \equiv \mathtt{xmlhttp}$ then    → Perform XMLHTTPRequest
33:     let $protocol \leftarrow \{\mathtt{P}, \mathtt{S}\}$
34:     let $host \leftarrow \mathsf{Doms}$
35:     let $path \leftarrow \mathbb{S}$
36:     let $fragment \leftarrow \mathbb{S}$
37:     let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
38:     let $url := \langle \mathtt{URL}, protocol, host, path, parameters, fragment\rangle$
39:     let $command := \langle \mathtt{XMLHTTPREQUEST}, url, \langle\rangle, \langle\rangle\rangle$
40:     stop $\langle s, cookies, localStorage, command\rangle$
```

Algorithm 7 Web Browser Model: Deliver a message to the script in a document or a service worker.

```
1: function DELIVER_TO_DOC($nonce$, $data$, $s'$)
2:     if $\exists x \in \mathcal{P}(\mathbb{N}) : s'.\overline{sw'}.nonce \equiv nonce$ then
3:         let $\overline{sw} := \mathcal{P}(\mathbb{N})$ such that $s'.\overline{sw'}.nonce \equiv nonce$
4:         let $s'.\overline{sw}.scriptinputs := s'.\overline{sw}.scriptinputs +^{\langle\rangle} data$
5:     else
6:         let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$, $\overline{d}$ such that $s'.\overline{d}.\mathtt{nonce} \equiv docnonce \wedge s'.\overline{d} = s'.\overline{w}.\mathtt{activedocument}$
           $\hookrightarrow$ if possible; otherwise stop
7:         let $s'.\overline{d}.\mathtt{scriptinputs} := s'.\overline{d}.\mathtt{scriptinputs} +^{\langle\rangle} data$
```

Algorithm 8 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, $key$, $f$, $s'$)
   ——— Process headers in response ———
2:     if Set-Cookie $\in$ $response$.headers then
3:         for each $c \in^{\langle\rangle}$ $response$.headers[Set-Cookie], $c \in$ Cookies do
4:             let $s'$.cookies[$request$.host] := AddCookie($s'$.cookies[$request$.host], $c$)
5:     if Strict-Transport-Security $\in$ $response$.headers $\wedge$ $requestUrl$.protocol $\equiv$ S then
6:         let $s'$.sts := $s'$.sts $+^{\langle\rangle}$ $request$.host
7:     if Referer $\in$ $request$.headers then
8:         let $referrer$ := $request$.headers[Referer]
9:     else
10:        let $referrer$ := $\bot$
11:    if Location $\in$ $response$.headers $\wedge$ $response$.status $\in \{303, 307\}$ then
12:        let $url$ := $response$.headers[Location]
13:        if $url$.fragment $\equiv \bot$ then
14:            let $url$.fragment := $requestUrl$.fragment
15:        let $method'$ := $request$.method
16:        let $body'$ := $request$.body
17:        if $response$.status $\equiv 303 \wedge$ $request$.method $\not\in$ {GET, HEAD} then
18:            let $method'$ := GET
19:            let $body'$ := $\langle\rangle$
20:        if Origin $\in$ $request$.headers $\wedge$ $method'$ $\equiv$ POST then
21:            let $origin$ := $\diamond$
22:        else
23:            let $origin$ := $\bot$
24:        if $\pi_1(reference) \not\equiv$ XHR then    → Do not redirect XHRs.
25:            let $req$ := $\langle$HTTPReq, $\nu_6$, $method'$, $url$.host, $url$.path, $url$.parameters, $\langle\rangle$, $body'\rangle$
26:            let $referrerPolicy$ := $response$.headers[ReferrerPolicy]
27:            call HTTP_SEND($reference$, $req$, $url$, $origin$, $referrer$, $referrerPolicy$, $s'$)
   ——— Deliver/process data in response ———
28:    switch $\pi_1(reference)$ do
29:        case REQ    → normal response
30:            let $\overline{w}$ ← Subwindows($s'$) such that $s'.\overline{w}$.nonce $\equiv \pi_2(reference)$
                ↪ if possible; otherwise stop
31:            if $response$.body $\not\sim \langle *, * \rangle$ then
32:                stop {}, $s'$
33:            let $script$ := $\pi_1$($response$.body)
34:            let $scriptinputs$ := $\pi_2$($response$.body) $+^{\langle\rangle}$ GET_SWS($requestUrl$, $s'$)    → Payment Handler Extension
35:            let $d$ := $\langle\nu_7$, $requestUrl$, $response$.headers, $referrer$, $script$, $\langle\rangle$, $scriptinputs$, $\langle\rangle$, $\top\rangle$
36:            if $s'.\overline{w}$.documents $\equiv \langle\rangle$ then
37:                let $s'.\overline{w}$.documents := $\langle d\rangle$
38:            else
39:                let $\overline{i}$ ← $\mathbb{N}$ such that $s'.\overline{w}$.documents.$\overline{i}$.active $\equiv \top$
40:                let $s'.\overline{w}$.documents.$\overline{i}$.active := $\bot$
41:                remove $s'.\overline{w}$.documents.$(\overline{i}+1)$ and all following documents from $s'.\overline{w}$.documents
42:                let $s'.\overline{w}$.documents := $s'.\overline{w}$.documents $+^{\langle\rangle}$ $d$
43:            stop {}, $s'$
```

```
44:         case XHR   → process XHR response
45:             let $headers := response.\mathtt{headers} - \mathtt{Set\text{-}Cookie}$
46:             let $m := \langle \mathtt{XMLHTTPREQUEST}, headers, response.\mathtt{body}, \pi_3 reference \rangle$
47:             call DELIVER_TO_DOC$(\pi_2(reference), m, s')$
48:             stop $\{\}, s'$
49:         case WS   → process WebSocket response
50:             if $response.\mathtt{status} \not\equiv \mathtt{101} \vee response.\mathtt{headers}[\mathtt{Upgrade}] \not\equiv \mathtt{websocket}$ then
51:                 stop
52:             let $wsconn := \langle reference, request.\mathtt{nonce}, key, f \rangle$
53:             let $s'.\mathtt{wsConnections} := s'.\mathtt{wsConnections} +^{\langle\rangle} wsconn$
```

Algorithm 9 Web Browser Model: Get relevant service workers for URL

```
1: function GET_SWS$(url, s')$
2:     let $handlers := \{\}$
3:     for each $\mathtt{handler} \in s'.serviceWorkers$ do
4:         if $\mathtt{handler.scope.domain} = url.\mathtt{domain} \wedge \mathtt{handler.scope.protocol} = url.\mathtt{protocol}$ then
5:             let $handlers := handlers \cup \{\mathtt{handler}\}$
       return $handlers$
```

A.3 Payment Provider Server

Definition A.3.1
The set of possible states Z^{pp} of a payment provider server pp is the set of terms of the form:

$$\langle DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, ids, \boldsymbol{transactions}, \boldsymbol{tokens}\rangle$$

Where DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, and ids defined as in the Web Infrastructure Model [8].

The remaining sub terms have the following form:

- *transactions* $\in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$ *stores all transactions that have been submitted and accepted by the payment provider server.*
- *tokens* $\in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$ *stores all auth tokens that have been authorized by the payment provider through basic authorization.*

An initial state s_0^{pp} of pp is a state of pp with s_0^{pp}`.pendingDNS` $\equiv \langle\rangle$, s_0^{pp}`.pendingRequests` $\equiv \langle\rangle$, s_0^{pp}`.corrupt` $\equiv \bot$, s_0^{pp}`.keyMapping` *being the same as the key mapping for browsers [9],* s_0^{pp}`.tlskeys` $\equiv$ `tlskeys`^{pp}, s_0^{pp}`.transactions` $\equiv \langle\rangle$, *and* s_0^{pp}`.tokens` $\equiv \langle\rangle$.

Placeholder	Usage
ν_1	Algorithm 10, placeholder for auth token used by service worker

Table A.2 List of placeholders used in payment provider server algorithms.

The model of the payment provider server is based on the template for HTTPS servers offered in the WIM [8].

Table A.2 shows the placeholder that was added to the generic HTTPS model offered in [8]. It is used to create authentication tokens with which a payment can be authenticated.

The template is modeled in Algorithm 10 with a basic interface offering the ability to process HTTPS requests of three kinds.

/`index` serves an entry index page with the `script_client_index` script.

/`authenticate` expects the clients credentials as input to respond with an authentication token that can later be used to pay requests.

/`pay` is used to process a transaction given a receiver, a total and an authentication token.

In the algorithm, transactions are stored within a dictionary. This allows for an easy implementation of the retry mechanism, since earlier seen requests of the same request id are simply overwritten.

The `script_client_index` script is defined in Algorithm 11. It can only do two different things. First, it can obtain an authorization token by use of a `XMLHTTPREQUEST`. Secondly, it can submit such a token with a post request to the service worker waiting for it.

Algorithm 10 Relation of an PP R^i: Processing HTTPS Requests.

```
1: function PROCESS_HTTPS_REQUEST($m$, $k$, $a$, $f$, $s'$)
2:     if $m$.path $\equiv$ /index then
3:         let $headers$ := $\langle\langle$ReferrerPolicy, origin$\rangle\rangle$
4:         let $m'$ := $\mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $headers$, $\langle$script_client_index, $\langle\rangle\rangle\rangle, k)$
5:     else if $m$.path $\equiv$ /authenticate then
6:         let $identity$ := $m$.body[id]
7:         let $password$ := $m$.body[secret]
8:         if $password \neq$ secretOfID($identity$) then
9:             stop $\langle\rangle$, $s'$
10:        let $token$ := $\langle$TOKEN, $\nu_1\rangle$
11:        let $s'$.tokens[$identity$] := $s'$.tokens[$identity$] $+^{\langle\rangle}$ $token$
12:        let $m'$ := $\mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $\langle\rangle$, $token\rangle$, $k)$
13:        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
14:    else if $m$.path $\equiv$ /pay $\wedge$ $m$.method $\equiv$ POST then
15:        let $token$ := $m$.body[token]
16:        let $sender \leftarrow ids$ such that $token \in s'$.tokens[$sender$]
         $\hookrightarrow$ if possible; otherwise stop
17:        let $receiver$ := $m$.body[receiver]
18:        let $total$ := $m$.body[total]
19:        let $paymentrequestnonce$ := $m$.body[paymentrequestnonce]
20:        let $paymentIdentifier$ := $m$.body[paymentIdentifier]
21:        let $s'$.transactions[$\langle paymentIdentifier\rangle$] := $\langle sender, receiver, total, pid, paymentrequestnonce\rangle$
     $\rightarrow$ Seamlessly integrate retry updates
22:        let $m'$ := $\mathsf{enc_s}(\langle$HTTPResp, $m$.nonce, 200, $\langle\rangle$, $\langle\rangle\rangle$, $k)$
23:        stop $\langle\langle f, a, m'\rangle\rangle$, $s'$
```

Algorithm 11 Relation of *script_client_index*.

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
1: let $switch \leftarrow \{$auth, postToken$\}$
2: if $switch \equiv$ auth then
3:     let $url$ := GETURL($tree$, $docnonce$)
4:     let $id \leftarrow ids$
5:     let $username$ := $\pi_1(id)$
6:     let $domain$ := $\pi_2(id)$
7:     let $interactive \leftarrow \{\bot, \top\}$
8:     let $url'$ := $\langle$URL, S, $url$.host, /authenticate, $\langle\rangle$, $\langle\rangle\rangle$
9:     let $secret$ such that $secret$ = secretOfID($id$) $\wedge$ $secret \in secrets$ if possible; otherwise
         $\hookrightarrow$ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$
10:    let $command$ := $\langle$XMLHTTPREQUEST, $url'$, POST, $\langle id, secret\rangle$, $\bot\rangle$
11:    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
12: else
13:    let $token$ such that $\pi_1(token)$ = TOKEN $\wedge$ $token \in scriptinputs$ if possible; otherwise
         $\hookrightarrow$ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$
14:    let $swNonce$ such that $\pi_1(swNonce)$ = SWNONCE $\wedge$ $swNonce \in scriptinputs$ if possible; otherwise
         $\hookrightarrow$ stop $\langle s, cookies, localStorage, sessionStorage, \langle\rangle\rangle$
15:    let $command$ := $\langle$POSTMESSAGE, $swNonce$, $token$, $\langle\rangle\rangle$
16:    stop $\langle scriptstate, cookies, localStorage, sessionStorage, command\rangle$
```

A.4 Merchant Server

Definition A.4.1
The set of possible states Z^{ms} of a merchant server ms is the set of terms of the form:

$$\langle DNSaddress, pendingDNS, pendingRequests, tlskeys, keyMapping, corrupt\rangle$$

Where DNSaddress, pendingDNS, pendingRequests, tlskeys, keyMapping, and corrupt defined as in the Web Infrastructure Model [8].

The definition of the server state is therefore exactly the same as in the WIM.

An initial state s_0^{ms} of ms is a state of as with s_0^{ms}.`pendingDNS` $\equiv \langle\rangle$, s_0^{ms}.`pendingRequests` $\equiv \langle\rangle$, s_0^{ms}.`corrupt` $\equiv \bot$, s_0^{ms}.`keyMapping` *being the same as the keymapping for browsers [9], and* s_0^{ms}.`tlskeys` $\equiv$ `tlskeys`^{as}.

The model of the payment provider server is based on the template for HTTPS servers offered in the WIM [8].

The template is implemented in Algorithm 12 with a basic interface offering the ability to process HTTPS requests of only a single kind.

`/index` serves an entry index page with the `script_arbitrary_merchant` script.

Algorithm 12 Relation of a merchant server R^i: Processing HTTPS Requests.

```
1: function PROCESS_HTTPS_REQUEST($m$, $k$, $a$, $f$, $s'$)
2:     if $m$.path ≡ /index then
3:         let headers := ⟨⟨ReferrerPolicy, origin⟩⟩
4:         let $m'$ := $\mathsf{enc_s}$(⟨HTTPResp, $m$.nonce, 200, headers, ⟨script_arbitrary_merchant, ⟨⟩⟩⟩, $k$)
5:     stop ⟨⟩, $s'$
```

`script_arbitrary_merchant` is not defined through code but through the following textual definition. `script_arbitrary_merchant` non-deterministically chooses a command to output for each iteration of the RUNSCRIPT algorithm (Algorithm 2). By modeling it like this, the bahavior of a malicious and an honest merchant does not have to be differentiated.

A.5 Web Payment APIs model with attackers

The formal model of the Web Payment APIs model is based on the definition of a web system given in Definition 27 of the WIM [8].

Definition A.5.1
A web system $\mathcal{WPAPI} = (\mathcal{W}, \mathcal{S}, \texttt{script}, E_0)$ is tuple called a Web Payment APIs model with attackers. It is defined as follows:

- *$\mathcal{W}$ denotes a system described by a set of Dolev-Yao processes. It is partitioned into the sets* Hon *and* Net. Net *includes a network attacker process.* Hon *consists out of a finite set of web browsers B, a finite set of web servers for the merchants C and a finite set of payment provider servers PP with* Hon := $B \cup C \cup PP$.

  As in [9], DNS servers are not modeled directly since they are subsumed by the network attacker.

- *$\mathcal{S}$ contains the scripts of the model with a mapping to their string representation. They are shown in table A.3.*

- *E^0 is defined as in the WIM[8] as an (infinite) sequence of events, containing an infinite number of events of the form $\langle a.a.\texttt{TRIGGER}\rangle$ for every $a \in \cup_{p \in \mathcal{W}} I^p$.*

$s \in \mathcal{S}$	script(s)
script_default_payment_handler	script_default_payment_handler
script_client_index	script_client_index

Table A.3 List of scripts with their mapping to a string represenation

A.6 Mentionable simplifications and exclusions

In the here presented model of the Web Payment APIs, several aspects differ from the exact specification in the APIs. This section is intended to give a short overview over the aspects that differ and the reasoning for why these aspects were modeled differently.

During the processing of a payment request, there exists a set of situations in which a payment request can be updated by the use of the updateWith() method of a PaymentMethodChangeEvent. In the here presented model, a payment request can be updated independently of the existence of a PaymentMethodChangeEvent. This decision simplifies the resulting method significantly while not weakening the resulting proof. In a realistic scenario, one can expect that if an attacker would want to perform such an update, that the attacker would be able to influence the user to creating such an event by e.g., entering a different receiver address.

Furthermore, the `onpaymentmethodchange` interface of the payment request object, and the `changePaymentMethod()` interface of a PaymentRequestEvent are not modelled, since their general use-case are unclear to the author. Additionally, they do not communicate new information and happen after the relevant submission of the PaymentRequestEvent (Payment Handler). Such arbitrary communication can be implemented through different and existing interfaces as well.

The interface `onshippingaddresschanged` and the `redactList` feature of addresses are not modeled as well. This aspect is not modelled since it is a direct threat to a customer's privacy. The approach of using a redact list is very basic and still leaks a lot of sensitive data. Instead of modelling it, a thorough discussion is offered. If it had been added to the model, the issue would have been as visible as without modeling it.

The feature set of service workers that was modeled is not exhaustive. Features such as the installation process and their functionality as network proxies are not depicted. Offering an exhaustive model of the service worker API would have gone beyond the scope of this work. The installation process is complex and issues within it would not have been linked to the Web Payment APIs but to the service worker API. Therefore, we assume in this work that the service workers are already installed at the beginning of a run. This might be a oversimplification that hides existing issues within the APIs. Furthermore, only the necessary features of service workers, that directly affect the Payment Handler API were modelled.

Additionally, we did exclude modelling the feature of merchant validation in this thesis. The feature of merchant validation was introduced very recently and we interpret this feature and its specification as unstable.

B Definitions

The following definitions extend the original WIM[8] and are referenced throughout the extended model and its analysis. Besides completing the formal definition of the earlier presented extension of the WIM, this chapter serves as a reference, explaining which information is stored in which entities such as events and objects.

Definition B.0.1 (Payment Request)
A Payment Request is a term:

$$\langle \mathsf{PAYMENTREQUEST}, \mathit{paymentRequestNonce}, \mathit{documentnonce}, \mathit{methodData}, \mathit{details}, \mathit{options},$$

$$\mathit{shippingOption}, \mathit{state}, \mathit{updating}, \mathit{responseNonce}\rangle$$

*Where paymentRequestNonce $\in \mathcal{N}$, documentnonce $\in \mathcal{N}$, methodData $\in \mathcal{T}_{\mathcal{N}}$, details $\in \mathcal{T}_{\mathcal{N}}$, options $\in \mathcal{T}_{\mathcal{N}}$, shippingOption $\in \mathcal{T}_{\mathcal{N}}$, state $\in \{CR, IN, CL\}$ (corresponding to the states: **Cr**eated, **In**teractive and **Cl**osed), updating $\in \{\top, \bot\}$, and responseNonce $\in \mathcal{N}$.*

Definition B.0.2 (Payment Request Event)
A Payment Request Event is a term:

$$\langle \mathsf{PAYMENTREQUESTEVENT}, \mathsf{paymentRequestNonce}, \mathsf{handlerNonce}, \mathit{methodData}, \mathit{total},$$

$$\mathit{modifiers}, \mathit{instrumentKey}, \mathit{requestBillingAdress}\rangle$$

Where paymentRequestNonce $\in \mathcal{N}$, handlerNonce $\in \mathcal{N}$, methodData $\in \mathcal{T}_{\mathcal{N}}$, total $\in \mathcal{T}_{\mathcal{N}}$, modifiers $\in \mathcal{T}_{\mathcal{N}}$, instrumentKey $\in \mathcal{T}_{\mathcal{N}}$, and requestBillingAdress $\in \{\bot, \top\}$.

Definition B.0.3 (Payment Handler Response)
A Payment Handler Response is a term:

$$\langle \mathsf{PAYMENTHANDLERRESPONSE}, \mathsf{paymentRequestNonce}, \mathsf{handlerNonce}, \mathit{methodName}, \mathit{details}\rangle$$

Where $\mathsf{paymentRequestNonce} \in \mathcal{N}$, $\mathsf{handlerNonce} \in \mathcal{N}$, *methodName $\in \mathcal{T}_{\mathcal{N}}$, and details $\in \mathcal{T}_{\mathcal{N}}$.*

Definition B.0.4 (Payment Response)
A Payment Response is a term:

$$\langle \mathsf{PAYMENTRESPONSE}, \mathsf{paymentResponse}, \mathsf{paymentRequestNonce}, \mathit{handlerNonce}, \mathit{methodName}, \mathit{details},$$

$$\mathit{shippingAddress}, \mathit{shippingOption}, \mathit{payerInfo}, \mathit{complete}\rangle$$

Where paymentResponse $\in \mathcal{N}$, paymentRequestNonce $\in \mathcal{N}$, handlerNonce $\in \mathcal{N}$, methodName $\in \mathcal{T}_{\mathcal{N}}$, details $\in \mathcal{T}_{\mathcal{N}}$, shippingAddress $\in \mathcal{T}_{\mathcal{N}}$, shippingOption $\in \mathcal{T}_{\mathcal{N}}$, payerInfo $\in \mathcal{T}_{\mathcal{N}}$, and complete $\in \{\bot, \top\}$.

Definition B.0.5 (Service Worker Registration)
A Service Worker Registration is defined through a term of the form:

$$\langle nonce, scope, script, scriptinputs, scriptstate, paymentManager, trusted\rangle$$

where $nonce \in \mathcal{N}$, $scope \in URLs$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $paymentManager \in PaymentManagers$, and $trusted \in \{\bot, \top\}$.

The set of all Service Worker Registrations is called ServiceWorkerRegistrations.

Definition B.0.6 (Payment Method Identifier)
A Payment Method Identifier is a URL for that holds true:

$$protocol = S$$

PaymentMethodIdentifiers is the set of all possible Payment Method Identifiers.

Definition B.0.7 (Payment Manager)
A Payment Manager is a term consisting of a series of PaymentInstruments. The set of all Payment Managers is denoted by PaymentManagers

Definition B.0.8 (Payment Instrument)
A Payment Instrument is a term

$$\langle \mathit{instrumentKey}, \mathit{enabledMethods}\rangle$$

where instrumentKey $\in \mathbb{S}$ and enabledMethods $\in PaymentMethodIdentifiers$.

Definition B.0.9 (Payment Method Data)
A payment method data term is a term consisting of a sequence of terms

$$\langle x_1, x_2, \dots\rangle$$

where each x_i is a term $\langle \mathit{pmi}, \mathit{receiver}, \mathit{paymentIdentifier}\rangle$ with pmi $\in$ PaymentMethodIdentifiers, receiver $\in \mathcal{T}_{\mathcal{N}}$, and paymentIdentifier $\in \mathbb{S}$. The set of all possible payment method data terms is referenced by MethodDatas.

C Security Properties

As introduced in Section 3.3, the three major security goals of the Web Payment APIs are session integrity, privacy and payment integrity. In this work, we excluded the privacy component for the security analysis and focused our efforts on the integrity properties. The following subsections introduce the properties in both an intuitive and a formal manner.

C.1 Session Integrity

Intuitively, session integrity describes the property that a payment can only be performed if the sender of the transaction did intend to perform this transaction. No other malicious party can trigger transactions in the name of the sender.

Put more formally, the following definition expresses this authentication property.

Definition C.1.1 (Session Integrity Property)
We say that a Web Payment APIs system model $\mathcal{WPAPI}^n$ fulfills Session Integrity iff for every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$.transactions and the honest browser $b = \mathtt{ownerOfID}(\mathtt{t.sender})$ it holds true:

$$t.\mathtt{paymentRequestNonce} \in S(b).\mathtt{paymentIntents}$$

C.2 Payment Integrity

Intuitively, the financial transactions submitted to a payment provider server should only be transactions that the user intended to be processed. The sender, receiver and total should not be different to what the user did intend to pay. We describe this by a security property that we call payment integrity. This name is chosen since the messages received by the payment provider server should not be different to the users intent.

To express the user's intent, a field *paymentIntents* was introduced to the browser's state, that keeps track of the status of a paymentrequest when the user submits it.

Because of pending events that were not processed yet, there can not be an exact one-to-one mapping between paymentIntents and transactions in every step.

We therefore formulated two properties that slightly relax the assumption of a one-to-one mapping. By Intended Payments we formulate the property that if a transaction exists on an honest payment provider server, there must exist exactly one browser that has a payment intent with the same paymentRequestNonce, total and receiver. Since this property would still allow for multiple transactions for a single intent, we introduce a second property. By Uniqueness of Payments we formulate the property that if there exists a paymentIntent in a browser, there exists at most one transaction that is founded on this payment intent (identified by the payment request nonce). Furthermore, said transaction, if it exists, has to match the corresponding sender, total and receiver of the payment intent.

The final payment integrity property is described through the combination of both properties.

The following definitions express this intuitive description formally.

Definition C.2.1 (Intended Payments)
We say that a Web Payment APIs system model $\mathcal{WPAPI}^n$ fulfills Intended Payments iff for every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$.transactions it holds true:

There exists a browser b in B, with $\texttt{ownerOfID}(\texttt{t.sender}) = b$*, such that:*

- $\exists i \in \mathbb{N} : \pi_i(S(b).\texttt{paymentIntents}[t.\texttt{PRN}]).\texttt{total} = t.\texttt{total}$
- $\exists i \in \mathbb{N} : \pi_i(S(b).\texttt{paymentIntents}[t.\texttt{PRN}]).\texttt{methodData.receiver} = t.\texttt{receiver}$

Definition C.2.2 (Uniqueness of Payments)
We say that a Web Payment APIs system model $\mathcal{WPAPI}^n$ fulfills uniqueness of payments iff for every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in S(b)$.$\texttt{paymentIntents}$ *it holds true:*

$$\left| \bigcup_{\substack{pp \in PP: \\ S(pp).\texttt{isCorrupted} = \bot}} \{t \in S(pp).\texttt{transactions} \mid prn = t.\texttt{paymentRequestNonce}\} \right| \leq 1$$

If such a t existst, there exists a $pi \in his$ such that:

- $b = \texttt{ownerOfID}(\texttt{t.sender})$
- $t.\texttt{total} = pi.\texttt{total}$
- $t.\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$

Definition C.2.3 (Payment Integrity Property)
We say that a Web Payment APIs system model $\mathcal{WPAPI}^n$ fulfills payment integrity iff the system both fulfills the Intended Payments property and the Uniqueness of Payments property.

D Proofs

D.1 General Properties

Lemma 1 (Credentials do not leak). For every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, for every honest browsers $b \in B$, every $id \in b.\texttt{ids}$ with $pp = \texttt{governor}(id)$ and pp honest, it holds true:

$$\forall p \in \mathcal{W} \backslash \{b, pp\} : \texttt{secretOfID}(id) \notin d_{\varnothing}(S(p))$$

Proof. Let $s := \texttt{secretOfID}(id)$. Without loss of generality, we choose a concrete id, s, b, and pp with above mentioned properties.

We show that s can not be obtained by any other party $p \in \mathcal{W} \backslash \{b, pp\}$. Let's assume that the opposite is the case. There exists a $p \in \mathcal{W} \backslash \{b, pp\}$ that can derive s:

$$\exists p \in \mathcal{W} \backslash \{b, pp\} : \texttt{secretOfID}(id) \in d_{\varnothing}(S(p))$$

In S^0 only b and pp can derive s by definition. Therefore, p must have obtained s through an action of either b or pp. This action could either be a corruption of b or pp or a regular message of an honest b or pp.

Since both b and pp are honest by assumption, the attacker can not obtain s through corruption of neither b nor pp.

pp only uses the credentials in Line 8 of Algorithm 10. In this line, the credentials are only compared to the submitted credentials of the user. At all other paths, even if s is part of the request, it can not leak to the attacker. Since the credentials do not become part of a message, pp can not be the instance that makes s derivable for p.

Let's therefore consider b. By definition of the browser, only scripts and serviceworkers from the origin $\langle id.\texttt{domain}, S \rangle$ can obtain s. The only scripts / service workers that are served through the origin corresponding to pp are *script_client_index* (Algorithm 11) and *script_default_payment_handler* (Algorithm 6).

Case 1: *script_client_index*: *script_client_index* references s in Line 9. The secret is transmitted through an HTTPS connection to the host (instance) that served the page. Since this page has to have had the same origin as the secret, this can only be pp. pp did already know the credentials and does not leak them as shown above. The only remaining option is leaking the credentials through observation of the message and inferring the content. As Lemma 8 of the WIM [7] shows, this is not possible for HTTPS messages.

Case 2: *script_default_payment_handler*: *script_default_payment_handler* does not reference the credentials at all.

Therefore, neither b nor pp do leak s to any other p. Which proofs the Lemma.

■

Lemma 2 (Authorization tokens do not leak). For every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $token \in S(pp).\texttt{tokens}$ with id such that $S(pp).\texttt{tokens}[id] = token$, $b \in B : b = \texttt{ownerOfID}(id)$ and b honest, it holds true:

$$\forall p \in \mathcal{W} \backslash \{b, pp\} : token \notin d_{\varnothing}(S(p))$$

Proof. All tokens of pp are nonces created in Line 10 of Algorithm 10. Since pp is honest and $token$ is a nonce of pp, an attacker can only obtain $token$ through an action of pp that allows other parties to derive $token$.

Since pp is honest, an attacker can not derive $token$ through corruption of pp. Therefore, we have to consider actions of pp that allow other instances to derive $token$.

As shown in Lemma 8 of the WIM [8], the used encrypted HTTPS messages do not leak $token$ to the attacker on observation.

There are only three cases in which pp creates messages: `/index`, `/authenticate`, and `/pay`. Neither `/index` nor `/pay` return a token and therefore are irrelevant. The remaining `/authenticate` does only create and return a token on submission of credentials.

As shown in Lemma 1, the attacker can not derive the credentials.

It is left to show that an owner of credentials $b \in B$ does not allow an attacker to derive $token$.

As mentioned above, a token is only issued as a response to a call to `/authenticate`. This call has to include valid credentials for a token to be issued. As explained in Lemma 1, the only source for said credentials are the *script_client_index*.

As mentioned in the proof of Lemma 1, to be able to access the corresponding secret of an ID, *script_client_index* has to be served by a corresponding origin $\langle id.\texttt{domain}, S\rangle$.

The only outgoing information leaving *script_client_index* can be found in Line 15. There, the message is posted to the serviceWorker that is available in the $scriptinputs$. This serviceWorker is added to the input in Line 34 of Algorithm 8 by GET_SWS (Algorithm 9). GET_SWS only returns service workers that are of the same scope as $id.\texttt{domain}$ which by the initial state definition of the browser state are considered trusted.

A corresponding service worker therefore has to be served by a $pp = \texttt{governor}(id)$.

As mentioned in the definition of the browser's initial state, service workers that are served through an honest payment provider are by definition trustworthy as well. This property was introduced to be in accordance with the payment method manifest [21]. Additionally, without this property an attacker would directly be able to serve arbitrary service worker and payment handler scripts through the domain of the payment method provider.

This, therefore, has to be *script_default_payment_handler*, since no other payment handler script is served by pp.

script_default_payment_handler accesses its $scriptinputs$ in several lines, but in all of them the first field of the term is checked for the right value of the term. Since this first field is set by the browser which is honest by assumption, only Line 11 is relevant.

The resulting outgoing message is sent via HTTPS message to the host of the service worker which has to be the owner of id.domain which is pp.

Therefore, $token$ can not leak.

■

Lemma 3 (No two PaymentRequests have same PaymentRequestNonce). For every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest $b \in B$, any payment request nonce *paymentRequestNonce* $\in \mathcal{N}$ it exists at most one $\overline{pr'} \in \mathsf{PaymentObjects}(S(b))$ such that:

$$\pi_1(S(b).\overline{pr'}) \equiv \mathsf{PAYMENTREQUEST} \wedge S(b).\overline{pr'}.\mathit{paymentRequestNonce} \equiv \mathit{paymentRequestNonce}$$

Proof. Since b is honest by assumption, we only have to consider a non-corrupted behaving browser b. Note, that the *paymentRequestNonce* of a payment request is never changed in the model after creation. This models the behaviour of the corresponding JavaScript object.

Furthermore, the first field of a term stored within the *paymentStorage* is never modified within the model. Therefore, we only have to consider the creation of objects in the *paymentStorage* that have the required first value of PAYMENTREQUEST. Within the model, such a paymentRequest is only generated in Line 96 of Algorithm 2. Each created payment request is initialized with a new and random *paymentRequestNonce*. Therefore, there can not be two payment requests sharing the same nonce.

■

Lemma 4 (PaymentRequestNonce in intent and transaction has to originate from browser). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, for every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in S(b).\mathsf{paymentIntents}$ and any transaction t of an honest payment provider pp it holds true:

$$(t.\mathsf{paymentRequestNonce} = prn) \Rightarrow (b = \mathsf{ownerOfID}(t.\mathit{sender}))$$

Proof. Let prn, b, t and pp be as defined in the lemma.

This proof is based on the additional assumption, that a payment request nonce does not leak through a malicious merchant's website. The payment request nonce is used within the model to reference the real-life JavaScript PaymentRequest object. Additionally, it was artificially introduced into the model to formulate a causal mapping between a transaction and a payment intent. This mapping allows for a comprehensive formalization of the security properties. In a real life setting, a JavaScript object reference can not leak to a party outside the browser. Therefore, this additional assumption does not introduce a loss of generality for the real-life setting.

The proof of the lemma is based on two properties. Firstly, we show that prn is only derivable by b and pp and does not leak to a malicious party. Afterwards, we show that t can only be added through the party that is the owner of the ID $t.\mathtt{sender}$. Since the creator of t must have had knowledge of prn and be the owner of the ID $t.\mathtt{sender}$, we can conclude that $(t.\mathtt{paymentRequestNonce} = prn) \Rightarrow (b = \mathtt{ownerOfID}(t.sender))$.

Property 1: prn is only derivable by b and pp Payment intents are only added to the browser state in Line 28 of Algorithm 3. The corresponding event must have been created either in Line 118 or Line 158 of Algorithm 2. Both directly obtain their payment request nonce through references to $\mathtt{PaymentRequests}(\mathrm{B}(s))$ which by definition originates from Line 96 of Algorithm 2. Therefore, the relevant payment request nonce is a fresh nonce whose initial derivability lies within b itself.

Within an honest browser knowledge of the payment request nonce is shared with the merchant's site script (Line 97, Algorithm 2) and the service worker to whom the payment request was submitted (Line 32, Algorithm 3). By above explained assumption, we assume that the merchant does not leak the payment request nonce with a party outside of the browser. A service worker only obtains the payment request nonce after a payment request was submitted, which only happens in case of trusted service workers (Line 118 and Line 158 of Algorithm 2).

A non-malicious payment handler that acts according to Algorithm 6, only shares the obtained payment request nonce in Line 17 with $swOrigin.\mathtt{host}$. In case of a trusted payment handler, this must represent an honest pp. Since HTTPS communication is used, the content does not leak to an outside party. As visible in Algorithm 10, an honest pp does not leak the payment request nonce as well.

Therefore, only b and pp can derive the payment request nonce.

Property 2: The creator of t must be $\mathtt{ownerOfID}(t.\mathtt{sender})$ Through Lemma 1 we know that in case of an honest $\bar{b}$ and an honest $\bar{pp}$ the credentials of an ID do not leak to an outside party. Furthermore, in case of an honest $\bar{pp}$ we need an authorization token $token$ to create the transaction t (Line 21, Algorithm 10). Through Lemma 2, we know that $token$ does not leak to a party that is not $\bar{b}$ or $\bar{pp}$. Therefore, the token $token$ must have been used by either $\bar{b}$ or $\bar{pp}$ to create the transaction. Since both are honest and an honest $\bar{pp}$ does not issue requests to its /pay endpoint, $\bar{b}$ must be the issuer. Since $t.\mathtt{sender}$ is directly associated to the token $token$ (Line 16, Algorithm 10), we can conclude that $\bar{b} = \mathtt{ownerOfID}(t.\mathtt{sender})$.

Since the issuer of t, has to be able to derive the credentials of $t.\mathtt{sender}$ and the payment request nonce, said issuer can only be b or pp. Since only b can create the corresponding payment intent, it therefore has to be b.

■

D.2 Session Integrity

Theorem 1 (Session Integrity). For every run ρ of any $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$.*transactions* and for b = `ownerOfID`(`t.sender`) being honest, it holds true:

$$t.\texttt{paymentRequestNonce} \in S(b).\texttt{paymentIntents}$$

Proof. For this proof we trace the transaction t back to its corresponding payment intent.

Since we consider only an honest pp, a transaction t is only added to $S(pp)$.`transactions` in case of a call to `/pay` (Line 21, Algorithm 10).

This call must have been authorized by a token tok stored in $S(pp)$.`tokens`. As shown in Lemma 2, a network attacker can not obtain such a token from pp without knowledge of the identity and secret of a victim.

Only an instance that can derive the credentials of an identity can issue and obtain a token. The credentials (Lemma 1) and the tokens (Lemma 2) do not leak to anybody except the honest browser b and the honest payment provider server pp. Since an honest pp does not issue calls to its own `/pay` enpoint, only b can have made the call to `/pay` that did trigger the creation of transaction t.

Let's consider the last authorized call to `/pay` of b that did lead to the creation/overwrite of t in $S(pp)$.`transactions`. Without loss of generality, let id be the identity that was used to authenticate the corresponding $token$ of the payment. Furthermore, let $pre := t.\texttt{paymentRequestNonce}$.

As mentioned in Lemma 1 and since b is not corrupted, the only script that uses `secretOfID`(id) is *script_client_index*.

In general, the relevant call to `/pay` could have had several origins, such as an URL navigation, a `FORM` command of a script, a `IFRAME` command of a script, a `XMLHTTPREQUEST` command of a script or a service worker and so forth.

But since the call has to be authorized by a token $token$ and has to be a `POST` call, only *script_default_payment_handler* can be the source of the call.

script_client_index, can not issue a `POST` call and only shares tokens with a *script_default_payment_handler* served of the same domain as *script_client_index*. As argued in Lemma 2, the token is only used in Line 11 of Algorithm 6.

Which is followed by the relevant `XMLHTTPREQUEST` command, that calls `/pay`.

As Line 8 shows, an event term with its first field being `PAYMENTREQUESTEVENT` is needed in the script inputs of the payment handler script for this line to be executed.

Therefore, a term with its first field being `PAYMENTREQUESTEVENT` must have been added to the corresponding scriptinputs of the service worker.

Notice that the submitted *paymentrequestnonce* is the one associated with said `PAYMENTREQUESTEVENT` event term.

The only place where this can happen to an honest b, is Line 32 of Algorithm 3. In this line the event is passed to the service worker in an unmodified manner.

This event only triggers if PROCESSEVENT was called with a corresponding event. For such an event to be added to $S(b)$.events, Line 25 has to be executed.

If this line was executed, Line 28 has executed as well, which sets the paymentIntent of $S(b)$.paymentIntents$[\mathit{paymentRequestNonce}] = S(b).\overline{pr'}$.

Since it is exactly this *paymentRequestNonce* that is passed in the event it follows, that *paymentRequestNonce* is the PaymentRequestNonce that is submitted in pp and stored in t.

This proves the theorem.

■

D.3 Intended Payments

Before proofing the fulfillment of the intended payments property, we offer a short informal overview of the proof. In a first step, the property is decomposed into three proofs. The first proof shows that if a transaction exists on an honest payment provider server, that exactly one browser may exist with a fitting paymentIntend in its paymentIntents (Theorem 1). Building uppon the first proof, the second proof shows that for said pair of the transaction and the paymentIntent, the totals have to be identical (Lemma 5). The third lemma shows that for said pair of the transaction and the paymentIntent, the receivers have to be identical (Lemma 6).

Lemma 5 (Equality of totals of paymentIntent and transaction). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$*.transactions* it holds true:

There exists a browser b in B, with ownerOfID(t.sender) = b, such that:

$$\exists i \in \mathbb{N} : \pi_i(S(b).\mathtt{paymentIntents}[t.\mathtt{paymentRequestNonce}]).\mathtt{total} = t.\mathtt{total}$$

Proof. As shown in Theorem 1, we know that

$$t.\mathtt{paymentRequestNonce} \in S(b).\mathtt{paymentIntents}$$

holds true for exactly one b = ownerOfID(t.sender).

Analogously to Theorem 1, pp itself can not be the source of a deviation in the total, since it is not corrupted, communicated through HTTPS and directly stores the $total$ value submitted to it when /pay is called.

As a remainder, we only have to show that b fulfills this property.

This lemma continuous from where Theorem 1 ended and shows that t.total must equal one payment that the user intended to execute.

The $total$ submitted to pp originates directly from Line 10 of Algorithm 6.

We use a similar argument as in Theorem 1:

As Line 8 shows, an event term with its first field being `PAYMENTREQUESTEVENT` is needed in the script inputs of the payment handler script for this line to be executed. Therefore, a term with its first field being `PAYMENTREQUESTEVENT` must have been added to the corresponding scriptinputs of the service worker.

Notice that the submitted *paymentrequestnonce* is the one associated with said `PAYMENTREQUESTEVENT` event term.

The only place where this can happen to an honest b, is Line 32 of Algorithm 3. In this line the event is passed to the service worker in an unmodified manner.

This event only triggers if PROCESSEVENT was called with a corresponding event. For such an event to be added to $S(b)$`.events`, Line 25 has to be executed.

Which leads to the conclusion, that the submitted total has to be the one of Line 17 of Algorithm 3. Therefore the value of $total$ can not be changed once the `SUBMITPAYMENT` event was executed and furthermore originates directly from the corresponding value of the PaymentRequest at that time.

Since this is the total that is added to $paymentIntents$, as well as the one sent to pp, this proves the lemma.

■

Lemma 6 (Equality of receivers of paymentIntent and transaction). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$*.transactions* it holds true:

There exists a browser b in B, with `ownerOfID(t.sender) = b`, such that:

$\exists i \in \mathbb{N} : \pi_1(\pi_i(S(b).\texttt{paymentIntents}[t.\texttt{paymentRequestNonce}]).\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}$

Proof. The proof of this lemma is analogous to the proof of Lemma 5.

As with the totals, a diverting $receiver$ could only originate from b.

The $receiver$ submitted to pp originates directly from Line 14 of Algorithm 6.

We use a similar argument as in Theorem 1:

As Line 8 shows, an event term with its first field being `PAYMENTREQUESTEVENT` is needed in the script inputs of the payment handler script for this line to be executed. Therefore, a term with its first field being `PAYMENTREQUESTEVENT` must have been added to the corresponding scriptinputs of the service worker.

Notice that the submitted *paymentrequestnonce* is the one associated with said `PAYMENTREQUESTEVENT` event term.

The only place where this can happen to an honest b, is Line 32 of Algorithm 3. In this line the event is passed to the service worker in an unmodified manner.

This event only triggers if PROCESSEVENT was called with a corresponding event. For such an event to be added to $S(b)$`.events`, Line 25 has to be executed.

Which leads to the conclusion, that the submitted methodData set has to be the one of Line 22 of Algorithm 3. Therefore, the value of $methodData$ can not be changed once the `SUBMITPAYMENT` event was executed and furthermore originates directly from the corresponding value of the PaymentRequest at that time.

As argued in Section 4.2, we introduced the assumption that payment handlers can identify which *methodData* to use unambiguously. Since in the model both `paymentIntent` and paymentHandler refer to the first *methodData* and its receiver value.

Accordingly, both choose the same *receiver* as the *methodData* sets that they use are identical copies that can not be changed.

■

Theorem 2 (Intended Payments). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, for every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest payment provider server $pp \in PP$, every $t \in S(pp)$*.transactions* it holds true:

There exists a browser b in B, with $\texttt{ownerOfID}(\texttt{t.sender}) = \texttt{b}$, such that:

- $\exists i \in \mathbb{N} : \pi_i(S(b).\texttt{paymentIntents}[t.\texttt{PRN}]).\texttt{total} = t.\texttt{total}$
- $\exists i \in \mathbb{N} : \pi_i(S(b).\texttt{paymentIntents}[t.\texttt{PRN}]).\texttt{methodData.receiver} = t.\texttt{receiver}$

Proof. Let pp, t and b defined as in theorem's description. As shown in Theorem 1, b exists and $b = \texttt{ownerOfID}(\texttt{t.sender})$ holds true. Furthermore, we know that $t.\texttt{paymentRequestNonce} \in S(b).\texttt{paymentIntents}$. Through Lemma 5 we know that within said $S(b).\texttt{paymentIntents}[t.\texttt{paymentRequestNonce}]$ there exists $i \in \mathbb{N}$ such that:

$$\pi_i(S(b).\texttt{paymentIntents}[t.\texttt{paymentRequestNonce}]).\texttt{total} = t.\texttt{total}$$

Lastly, Lemma 6 guarantees that there exists $i \in \mathbb{N}$ such that:

$$\pi_i(S(b).\texttt{paymentIntents}[t.\texttt{paymentRequestNonce}]).\texttt{methodData.receiver} = t.\texttt{receiver}$$

The combination of these three lemmas proves the theorem.

■

D.4 Uniqueness of Payments

We split the proof of uniqueness of payments into four major lemmas: The first lemma shows that for a given *paymentIntent*, at most one transaction t exists at an honest pp that shares the same paymentRequestNonce. Following, the remaining lemmas each show that given the existence of such a t, the properties $b = \texttt{ownerOfID}(\texttt{t.sender})$, $t.\texttt{total} =$ *paymentIntent*$.\texttt{total}$ and $t.\texttt{receiver} =$ *paymentIntent*$.\texttt{methodData.receiver}$ hold true.

Lemma 7 (At most one honest transaction for each paymentIntent). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in S(b).\texttt{paymentIntents}$ it holds true:

$$\left| \bigcup_{\substack{pp \in PP: \\ S(pp).\texttt{isCorrupted}=\bot}} \{t \in S(pp).\texttt{transactions} \mid prn = t.\texttt{paymentRequestNonce}\} \right| \leq 1$$

Proof. We proof this property by contradiction. Let's assume that

$$\left| \bigcup_{\substack{pp \in PP: \\ S(pp).\texttt{isCorrupted}=\bot}} \{t \in S(pp).\texttt{transactions} \mid prn = t.\texttt{paymentRequestNonce}\} \right| > 1$$

From which directly would follow that there are one or more payment provider servers that store a set of at least two transactions t_1 and t_2 in their $transactions$ with:

$$t_1.\texttt{paymentRequestNonce} = t_2.\texttt{paymentRequestNonce} = prn$$

Let pp_1 be the payment provider server associated to t_1 and pp_2 be the payment provider server of t_2. As shown in Lemma 1, the only way how t_1 and t_2 could have been added to pp_1 and pp_2 is through authorized calls to /pay and with a valid token. Since both pp_1 and pp_2 are honest by assumption, do only use HTTPS communication and because of Lemma 1 and Lemma 2, the issuer of these calls must have been $b_1 = \texttt{ownerOfID}(\texttt{t}_1.\texttt{sender})$ or $b_2 = \texttt{ownerOfID}(\texttt{t}_2.\texttt{sender})$.

Since $t_1.\texttt{paymentRequestNonce} = t_2.\texttt{paymentRequestNonce} = prn$, we can conclude through Lemma 4 that since $prn \in S(b).\texttt{paymentIntents}$, $\texttt{ownerOfID}(t_1.\texttt{sender}) = b = \texttt{ownerOfID}(t_2.\texttt{sender})$.

Therefore, $b = b_1 = b_2$.

Let's consider the last authorized call to /pay of b that did lead to the creation/overwrite of t_1.

Without loss of generality, let id be the identity that was used to authenticate the corresponding $token$ of the payment. Furthermore, let $prn := t.\texttt{paymentRequestNonce}$.

As mentioned in Lemma 1 and since b is not corrupted, the only script that uses $\texttt{secretOfID}(id)$ is *script_client_index*.

In general, the relevant call to /pay could have had several origins, such as an URL navigation, a FORM command of a script, a IFRAME command of a script, a XMLHTTPREQUEST command of a script or a service worker and so forth.

But since the call has to be authenticated by a token $token$ and has to be a POST call, only *script_default_payment_handler* can be the source of the call.

script_client_index, the source of $token$ for b, can not issue a POST call and only shares tokens with a *script_default_payment_handler* served of the same domain as *script_client_index*. As argued in Lemma 2, the token can only be accessed in Line 11 of Algorithm 6.

Which is followed by the relevant XMLHTTPREQUEST command that calls /pay.

As Line 8 shows, an event term with its first field being PAYMENTREQUESTEVENT is needed in the script inputs of the payment handler script for this line to be executed.

Therefore, a term with its first field being PAYMENTREQUESTEVENT must have been added to the corresponding scriptinputs of the service worker.

Notice that the submitted *paymentrequestnonce* is the one associated with said PAYMENTREQUESTEVENT event term.

The only place where this can happen to an honest b, is Line 32 of Algorithm 3. In this line, the event is passed to the service worker in an unmodified manner.

This event only triggers if PROCESSEVENT was called with a corresponding event. For such an event to be added to $S(b)$.events, Line 25 has to be executed.

If this line was executed, Line 28 has executed as well, which appends the paymentIntent of $S(b).\overline{pr'}$ to $S(b)$.paymentIntents[*paymentRequestNonce*].

Note that the *paymentRequestNonce* which is stored in the PAYMENTREQUESTEVENT is the one directly passed to the SUBMITPAYMENT event. Since t_1.paymentRequestNonce = t_2.paymentRequestNonce = prn, we can conclude that to satisfy the assumption of two transactions with the same payment request nonce, there is a need for two submit payment events that reference the same payment request nonce.

To summarize the proof to this point: Assumed that there would be multiple transactions that fulfill said property, then there must have been multiple SUBMITPAYMENT events that triggered the creation of these transactions.

To continue the proof, a modification to the model that diverts from the reference implementation of Google Chrome [12], had to be performed. Otherwise, the vulnerability explained in Section 4.1 would be possible.

SUBMITPAYMENT events can only be created in two places of the model: In line 118 and Line 158 of Algorithm 2.

As shown in Lemma 3, there can only exist one payment request object for a given *paymentRequestNonce*. The calls to PR_SHOW and/or PRES_RETRY therefore have to have happened on the same payment request object in the *paymentStorage*.

Let's consider the source of the second SUBMITPAYMENT event. As mentioned above, this must either have been PR_SHOW or PRES_RETRY.

Case 1: PR_SHOW PR_SHOW can not have been the source of the second SUBMITPAYMENT event. To be able to reach Line 118, $s'.\overline{pr'}.state$ must be CR (**Cr**eated). This is only the case for an unmodified $s'.\overline{pr'}.state$. All other overwrites of $s'.\overline{pr'}.state$ set the value either to IN or CL. Since PR_SHOW only creates the SUBMITPAYMENT event after the state was set to IN, it can not have been called in advance. Furthermore, PRES_RETRY can not have created a SUBMITPAYMENT event in advance as well. The reason for that is that it needs a PaymentResponse object with the fitting payment request nonce. PAYMENTRESPONSE payment objects are only created in Line 58 of Algorithm 3. This line is only executed when processing an PAYMENTHANDLERRESPONSE event (Line 34). A PAYMENTHANDLERRESPONSE is only created in Line 16 of Algorithm 4, if a

`PAYMENTREQUEST` event with a corresponding *paymentRequestNonce* exists in its scriptinputs (Line 15). The only situation in which such a `PAYMENTREQUEST` event is added to its scriptinputs is in Line 32 of Algorithm 3.

As argued earlier, these events must originate from a `SUBMITPAYMENT` event that can only be created through `PR_SHOW` or `PRES_RETRY`. If `PRES_RETRY` had been the source, this `PRES_RETRY` would have to have a `SUBMITPAYMENT` source itself which inductively at some point must have lead to a `PR_SHOW`. `PR_SHOW` can not have been called in advance since it would have set the $S(b).\overline{pr'}.state$ to IN. Therefore, `PR_SHOW` can not be the source of the second `SUBMITPAYMENT` event.

Case 2: `PRES_RETRY` As argued in case 1, `PRES_RETRY` mus be the source of such a second `SUBMITPAYMENT` event. And indeed, as shown in case 1, there can be infinite chains of `PRES_RETRY` commands, each triggering a `SUBMITPAYMENT` event. But, `SUBMITPAYMENT` events triggered by a `PRES_RETRY` commands, can not lead to seperate transactions.

First notice that the payment provider server that receives the resulting transaction has to be the same as earlier when using `PRES_RETRY`. The selected handler is the one that did trigger the PaymentResponse object which is used for this command. As argued above, an honest payment handler sends its `/pay` requests always to the payment provider associated through its domain/scope. Therefore, both requests must have been sent to the same payment provider server.

Furthermore, the submitted paymentIdentifier has to be the same. Since the paymentIdentifier part of the *methodData* is only set on creation and can not be changed afterwards. Therefore, it has to be the same as well.

The sender does not have to be equal, but it has to originate from a $token$ associated to an *id* for which $b = \mathsf{ownerOfID}(id)$ as argued above.

This aspect proves Lemma 8 as well.

Since for an honest payment provider this equality can only lead to the creation of a single transaction, this is a direct contradiction which proves that there can not have been two transactions in the very first place.

■

Lemma 8 (Authorized transactions for honest paymentIntent). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $pi \in \pi_i(S(b).\mathit{paymentIntents})$ it holds true:

If there exists an honest pp with a transaction $t \in S(pp).\mathsf{transactions}$ such that $pi.\mathit{paymentRequestNonce} = t.\mathsf{paymentRequestNonce}$, then:

$$b = \mathsf{ownerOfID}(\mathsf{t.sender})$$

Proof. This property was shown in the proof of Lemma 7. The basic argumentation lies within the fact that the creation of transactions has to be authorized by credentials and tokens, which do not leak (Lemma 1 and Lemma 2). For the precise proof refer to the proof of Lemma 7.

■

Lemma 9 (Equal transactions's total for honest paymentIntent). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in S(b)$.`paymentIntents` it holds true:

If there exists an honest pp with a transaction $t \in S(pp)$.`transactions` such that $prn = t$.`paymentRequestNonce`, then there existst $pi \in his$ such that:

$$t.\texttt{total} = pi.\texttt{total}$$

Proof. By assumption, we know that a transaction t exists for which $prn = t$.`paymentRequestNonce`. As shown earlier (Lemma 7), such transactions can only be created through a browser b for which b = `ownerOfID(t.sender)`. As shown in Lemma 8 such a transaction must have originated from processing a `SUBMITPAYMENT` event.

The *total* passed to the `PAYMENTREQUESTEVENT`, which is passed unmodified to the payment provider is determined in Line 17 of Algorithm 3. Exactly the same value is appended to the `paymentIntents` in Line 28.

Since paymentIntents are only appended and can not be modified, this is a $pi \in S(b)$.`paymentIntents`$[prn]$ for which t.`total` = pi.`total`

This proves the lemma.

■

Lemma 10 (Equal transactions's receiver for honest paymentIntent). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in S(b)$.`paymentIntents` it holds true:

If there exists an honest pp with a transaction $t \in S(pp)$.`transactions` such that *pi.paymentRequestNonce* = t.`paymentRequestNonce`, then there existst $pi \in his$ such that:

$$t.\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$$

Proof. By assumption, we know that a transaction t exists for which $prn = t$.`paymentRequestNonce`. As shown earlier (Lemma 7), such transactions can only be created through a browser b for which b = `ownerOfID(t.sender)`. As shown in Lemma 8 such a transaction must have originated from processing a `SUBMITPAYMENT` event.

The *methodData* passed to the `PAYMENTREQUESTEVENT`, is passed unmodified to the payment handler. Its value is calculated in Line 22.

The payment handler selects the first paymentMethod of the *methodData* submitted through the `PAYMENTREQUESTEVENT` event in Line 13 of Algorithm 6. Afterwards it submits the associated receiver to the payment provider (Line 14). As argued in (Lemma 7), this is the only sequence of events, that could have lead to the creation of t.

Exactly the same value for the *methodData* field is set for the payment intent (Line 27) that is appended to the `paymentIntents` in Line 28.

Since paymentIntents are only appended and can not be modified, this is a $pi \in \mathrm{S}(b).\mathtt{paymentIntents}[prn]$ for which $t.\mathtt{receiver} = \pi_1(pi.\mathtt{methodData}).\mathit{receiver}$

This proves the lemma.

■

Theorem 3 (Uniqueness of Payments). For every Web Payment APIs system model $\mathcal{WPAPI}^n$, every run ρ of $\mathcal{WPAPI}^n$, every configuration (S, E, N) in ρ, every honest browser $b \in B$, every $(prn, his) \in \mathrm{S}(b).\mathtt{paymentIntents}$ it holds true:

$$\left| \bigcup_{\substack{pp \in PP:\\ \mathrm{S}(pp).\mathtt{isCorrupted}=\bot}} \{t \in \mathrm{S}(pp).\mathtt{transactions} \mid prn = t.\mathtt{paymentRequestNonce}\} \right| \leq 1$$

If such a t exists, there exists a $pi \in his$ such that:

- $b = \mathtt{ownerOfID}(\mathtt{t.sender})$
- $t.\mathtt{total} = pi.\mathtt{total}$
- $t.\mathtt{receiver} = \pi_1(pi.\mathtt{methodData}).\mathtt{receiver}$

Proof. As shown in Lemma 7, there exists at most one such t. Lemma 8 shows that $\mathtt{ownerOfID}(\mathtt{t.sender}) = b$. Lemma 9 shows that $t.\mathtt{total} = pi.\mathtt{total}$. Finally, Lemma 10 shows that $t.\mathtt{receiver} = \pi_1(pi.\mathtt{methodData}).\mathtt{receiver}$.

■

Bibliography

[1] M. Abadi, C. Fournet. “Mobile Values, New Names, and Secure Communication”. In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 104–115. ISSN: 0362-1340. DOI: 10.1145/373243.360213. URL: https://doi.org/10.1145/373243.360213 (cit. on p. 25).

[2] Apple Inc. *Apple Pay*. URL: https://www.apple.com/apple-pay/ (cit. on pp. 11, 13, 14, 21).

[3] Apple Inc. *Providing Merchant Validation*. URL: https://developer.apple.com/documentation/apple_pay_on_the_web/apple_pay_js_api/providing_merchant_validation (cit. on pp. 21, 22).

[4] Apple Inc. *Safari*. URL: https://www.apple.com/safari/ (cit. on pp. 14, 15).

[5] K. Czesak. *Shaping the Future of Payments in the Browser*. Sept. 2018. URL: https://shopifyengineering.myshopify.com/blogs/engineering/shaping-the-future-of-payments-in-the-browser (cit. on p. 13).

[6] Z. Ðurić, O. Marić, D. Gašević. “Internet payment system: A new payment system for internet transactions”. In: *Journal of Universal Computer Science* 13.4 (2007), pp. 479–503 (cit. on p. 30).

[7] D. Fett, R. Küsters, 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)*. Aug. 2017, pp. 189–202. DOI: 10.1109/CSF.2017.20 (cit. on pp. 25, 73).

[8] D. Fett. “An expressive formal model of the web infrastructure”. In: (2018) (cit. on pp. 11, 25–27, 29, 42, 44, 45, 63, 65, 66, 69, 74).

[9] D. Fett, P. Hosseyni, R. Kuesters. *An Extensive Formal Security Analysis of the OpenID Financial-grade API*. 2019. arXiv: 1901.11520 [cs.CR] (cit. on pp. 63, 65, 66).

[10] GitHub. *Prevent double spending through retry · Issue 882 · w3c/payment-request*. URL: https://github.com/w3c/payment-request/issues/882 (cit. on p. 31).

[11] Google Inc. *Google Pay A better way to pay by Google*. URL: https://pay.google.com/intl/en_us/about/ (cit. on pp. 11, 13).

[12] Google Ireland Limited. *Google Chrome - The New Chrome and Most Secure Web Browser*. URL: https://www.google.com/intl/en_us/chrome/ (cit. on pp. 20, 82).

[13] MDN web docs. *PaymentResponse.retry()*. URL: https://developer.mozilla.org/en-US/docs/Web/API/PaymentResponse/retry (cit. on p. 31).

[14] Network Working Group. *A Universally Unique IDentifier (UUID) URN Namespace*. July 2005. URL: https://tools.ietf.org/html/rfc4122 (cit. on p. 19).

[15] PayPal. *Send Money, Pay Online or Set Up a Merchant Account*. URL: https://www.paypal.com/us/home (cit. on pp. 11, 13).

[16] T. C. Projects. *Chromium*. URL: https://www.chromium.org/Home (cit. on p. 31).

[17] O. Rachamim. *How many online stores are there in the world?* https://www.digitalcommerce360.com/2014/12/04/how-many-online-stores-are-there-world/. Dec. 2017 (cit. on p. 11).

[18] R. Solomakhin. *Payment Request Test*. URL: https://rsolomakhin.github.io/ (cit. on p. 20).

[19] W3C. *Payment Handler API*. Oct. 2019. URL: https://www.w3.org/TR/payment-handler/ (cit. on pp. 11, 15, 22, 31, 34, 55).

[20] W3C. *Payment Method Identifiers*. Sept. 2019. URL: https://www.w3.org/TR/payment-method-id/ (cit. on pp. 11, 15).

[21] W3C. *Payment Method Manifest*. Dec. 2017. URL: https://www.w3.org/TR/payment-method-manifest/ (cit. on pp. 11, 15, 44, 74).

[22] W3C. *Payment Method: Basic Card*. Jan. 2020. URL: https://www.w3.org/TR/payment-method-basic-card/ (cit. on pp. 15, 34).

[23] W3C. *Payment Request API*. Dec. 2019. URL: https://www.w3.org/TR/payment-request/ (cit. on pp. 11, 15, 31, 34, 36, 55).

[24] W3C. *Service Workers 1*. Nov. 2019. URL: https://www.w3.org/TR/service-workers/ (cit. on pp. 28, 29).

[25] W3C. *Web Payments Working Group*. https://www.w3.org/Payments/WG/ (cit. on p. 11).

All links were last followed on March 5, 2020.

Declaration