Frictionless Web Payments with Cryptographic Cardholder Authentication

Francisco Corella and Karen Lewison

Revised October 1, 2018

Contents

List of Figures

1 Introduction

The 3-D Secure protocol version 1.0, marketed under different names by different payment networks (Verified by Visa, MasterCard SecureCode, American Express SafeKey, etc.) aims at reducing online credit card fraud by authenticating the cardholder. To that purpose, the merchant's web site redirects the cardholder's browser to the issuing bank, which typically authenticates the cardholder by asking for a static password and/or a one-time password delivered to a registered phone number. 3-D Secure was introduced by Visa in 1999, but

it is still unevenly used in European countries and rarely used in the United States. One reason for the limited deployment of 3-D Secure is the friction caused by requiring users to remember and enter a password and/or retrieve and enter a one-time password. Consumers "hate" 3-D Secure 1.0 [1], and merchants are wary of transaction abandonment. Another reason may be that it facilitates phishing attacks by asking for a password after redirection [2, 3, 4].

3-D Secure 2.0 [5, 6] aims at reducing the friction caused by 3-D Secure 1.0. When 3-D Secure 2.0 is deployed, it will introduce a *frictionless flow* that will eliminate cardholder authentication friction for 95% of transactions deemed to be low risk. But it will do so by *eliminating cardholder authentication altogether* for those transactions. The merchant will send contextual information about the intended transaction to the issuer, including the cardholder's payment history with the merchant. The issuer will use that information, plus its own information about the cardholder and the merchant, to assess the transaction's risk, and will communicate the assessment to the merchant, who will redirect the browser to the issuer for high risk transactions but omit authentication for low risk ones.

This scheme has serious drawbacks. It is privacy invasive for the cardholder. It puts the merchant in a bind, who has to keep customer information for the sake of 3D-Secure while minimizing and protecting such information to comply with privacy regulations. It is complex for the issuer, who has to set up an AI "self-learning" risk assessment system. It requires expensive infrastructure, the contextual information that the merchant sends to the issuer goes through no less than three intermediate servers—a 3DS Server, a Directory Server and an Access Control server. And it provides little or no security benefit for low risk transactions, as the cardholder is not authenticated and the 3-D Secure risk assessment that the issuer performs before the merchant submits the transaction to the payment network is redundant with the risk assessment that it performs later before authorizing or declining the submitted transaction forwarded by the network.

Here we propose a scheme for securing online credit card payments with two-factor authentication of the cardholder without adding friction. The cardholder's checkout experience is the same as in ordinary web payment by credit card, except that the confirmation page where the cardholder reviews the transaction and submits payment is hosted by the issuing bank rather than by the merchant. The cardholder authenticates by proving knowledge of credit card and cardholder data as usual, plus proving possession of a private key kept in browser storage, where it is protected by the same origin policy of the web enforced by the browser. The merchant is given a digital signature on the transaction that provides non-repudiation.

The paper is organized as follows. In Section 2 we introduce terminology to simplify the exposition and avoid ambiguity. In Section 3 we specify the cryptographic payment credential used for cardholder authentication. In Section 4 we describe the infrastructure required to implement the proposed scheme. In Section 5 we specify the credential provisioning protocol. In Section 6 we explain how the issuer can avoid provisioning duplicate payment credentials to the same browser for the same credit card. In Section 7 we specify the cardholder authentication protocol. In Section 8 we describe defenses that can be used against same-origin attacks, malware, and physical capture. In Section 9 we show how to take advantage of a cryptographic module when available, by means of a browser extension.

2 Terminology

We use the following terminology to simplify the exposition and avoid ambiguity:

- We use the term *browser* as a synonym to the more technical term *user agent*. A browser is a particular instance of browser software, such as Chrome, Firefox, Safari, Edge or Internet Explorer, running on a particular machine.

- We refer to a web site as having a *back-end* consisting of server-side software, and a *front-end* consisting of HTML documents downloaded to browsers and JavaScript code embedded in those documents.

- By *downloading a script* we mean responding to an HTTP request with an HTTP response whose body is an HTML document containing an embedded JavaScript script but no displayable content.

- We use the term *DOM form* to refer to a Document Object Model form element created by JavaScript code rather than by an HTML form tag.

- *A POST redirection* is an HTTP response that downloads a script that creates a DOM form and submits it as an HTTP POST request. We refer to the HTTP POST request as *the request that results from the POST redirection*.

- We say that a downloaded script or other JavaScript code *contains some data* if that data is present in the JavaScript code encoded as one or more simple or composite JavaScript literals. For example, JavaScript code contains the credit card number 4111111111111111 if it contains an assignment such as:

```
var ccnum = "4111111111111111";
```

- A *leading script* is a script embedded in a HTML page that runs before the page is rendered by the browser. Such a script may preempt the display of the page by navigating to a different page before the page is rendered.

- In the literature, the term "Web API" is used ambiguously to refer to a server-side API, exposed by a Web server and used by sending HTTP requests to the server, or to a client-side API, exposed by browsers and accessed via function calls made by JavaScript code embedded in web pages. Here, we use the terms *HTTP API* to refer to the former, and *JavaScript API* to refer to the latter.

- In the literature, the term "endpoint" is used with several meanings. Here we use it as a more abstract term than "URL" to refer to the destination of HTTP requests sent to a web server to trigger a specific functionality. An endpoint has a URL, and saying that a request is sent to an endpoint is synonymous with saying that it is sent to the URL of the endpoint.

- In the literature, the term "scheme" is used with several meanings. Here we reserve the term to refer to the proposed cardholder authentication scheme and use alternative terminology for other meanings, e.g. we refer to a "digital-signature cryptosystem" instead of a "digital-signature scheme" and to the "protocol portion" of a URL instead of its "scheme".

- We say that a value generated at random with high entropy is *universally unique* to refer to the fact that there is a negligible probability that it will coincide with a preexisting value in the same or a different context. For example, we rely on universal uniqueness to assert that a random key ID generated in one browser will be different with overwhelming probability from a random key ID generated in a different browser.

3 Cryptographic payment credential

In the proposed scheme, web payments with a credit card are secured by means of *cryptographic payment credentials*, each provisioned to a browser used by the cardholder. A cryptographic payment credential consists of a private key and a short-term *credit card certificate*. The credit card certificate is signed by the issuing bank and contains the public key associated with private key, the credit card number, and certificate metadata including a short validity period.

A significant security feature of the proposed scheme is that the private key is stored in the cardholder's browser while the credit card certificate is kept by the issuing bank and supplied at transaction time. This makes it possible to use the credential on an ordinary browser without recourse to secure hardware and without exposing the credit card number to an attacker who gains physical possession of the machine that hosts the browser.

A cardholder who makes web payments with the same credit card on different browsers (hosted on the same or different machines) uses a different cryptographic payment credential for that card on each browser.

A cardholder who makes web payments on the same browser with different credit cards, issued by the same bank or by different banks, uses a different cryptographic payment credential for each card on that browser.

4 Infrastructure

The infrastructure required to implement the proposed scheme consists of a public database of issuing banks that support the proposed scheme (the *scheme database*). Each issuer is identified in the database by its issuer identification number (IIN), which consists of the first six digits of the credit card number. (Eight digits will be used in the future.) The database maps the IIN of each issuer to two data items used by the scheme:

1. The URL of a cardholder authentication endpoint, to which the merchant redirects the cardholder's browser to authenticate the cardholder and obtain a signature on the transaction.

2. The public key associated with the private key that the issuer uses to sign credit card certificates.

Merchants can download the scheme database or use an HTTP API to query the database. The database, downloads of the database, and queries to the HTTP API must be integrity-protected but need not be confidentiality-protected.

5 Credential provisioning protocol

The credential provisioning protocol comprises the following steps:

1. To obtain a cryptographic payment credential for a credit card to be used in a particular browser, the cardholder logs in with that browser to the web site of the card issuer and navigates to a credential provisioning page for that card, where a button is available for requesting the credential. A warning near the button tells the cardholder not to click the button on a public machine. Clicking the button causes a form to be submitted as an HTTP POST request. Since the cardholder has logged in, the browser sends a session cookie containing a login session ID along with the request. The form includes a hidden input containing a reference to a *credit card account record* stored in a database of the issuer's web site (the *issuer's database*), and a token for prevention of cross-site request forgery (CSRF) attacks. The *CSRF token* is a secret bound to the login session, such as a random high-entropy string stored in the login session record or an HMAC signature on the session ID computed with a symmetric key known only to the issuer.

2. In response to the request, after validating the session cookie and the CSRF token, the issuer downloads a script containing the credit card number and the CSRF token. The script checks if a payment credential has already been provisioned to the same browser for the same credit card, as described below in Section 6. If so, it redirects to an error page by submitting a DOM form. Otherwise the script generates a digital-signature key pair and a random *key ID* to be used as a reference to the private key.

   The key pair pertains to an asymmetric signature cryptosystem such as DSA, ECDSA or RSASSA-PSS. If a cryptosystem with domain parameters is used, such as DSA or ECDSA, it is assumed for simplicity that the domain parameters are included in the script as JavaScript literals and incorporated into the public key; alternatively, a single well-known set of domain parameters could be used for the scheme, or each issuer could have a set of domain parameters published in the scheme database.

   The key ID is generated at random with high-entropy and is therefore universally unique. The script stores the private key as a property of an object, which is itself stored in an in-browser *private-key database* accessible through the IndexedDB JavaScript API [7, 8], and can be retrieved using the key ID as a database retrieval key. The private-key database may already exist and contain private keys of payment credentials for other credit cards issued by the same issuer; otherwise it is created. Access to the private-key database is restricted by the same origin policy to scripts that originate from the issuer. Other private-key databases may be used by other issuers on the same browser.

3. The script creates a DOM form comprising the credit card number, the public key, the key ID, the CSRF token, and a signature on the other form inputs computed with the private key. The script submits the form as an HTTP POST request, which plays the role of a certificate-signing request (CSR). The browser sends the session cookie along with the request.

4. After validating the session cookie and the CSRF token, the issuer verifies the signature using the public key and creates a short-term credit card certificate binding the public key to the credit card number. Being short-term, the certificate will soon become invalid; but it will be replaced as needed by an uptodate certificate for the same public key as described in Section 7. The issuer stores the credit card certificate and the key ID in a *certificate record* of the issuer's database, together with the user-agent string found in the certificate-signing HTTP request and a reference to the credit card account record.

A credit card is said to be *enabled for cryptographic cardholder authentication (CCHA) on a particular browser* (abbreviated as *CCHA-enabled for the browser*) if that browser has been provisioned with a cryptographic payment credential for the card, in which case a certificate record that references the credit card account record has been created for the browser. Notice that, although the certificate record is specific to the browser, the only information about the browser in that record is the user-agent string, which identifies the browser software but does not uniquely identify the browser.

A credit card is said to be *CCHA-enabled* if it is CCHA-enabled for one or more browsers, which can be determined by checking if there are one or more certificate records that reference the credit card account record.

6 Avoiding duplicate provisioning

A cardholder may use multiple browsers on multiple machines to make online payments with different credit cards. If so, he or she may forget that a payment credit credential for a particular credit card has already been provisioned to a particular browser, and submit a duplicate provisioning request.

When a provisioning request is made from a browser, the issuer cannot tell that the request is a duplicate simply by checking if there is already a certificate record for the browser that refers to the credit card account record in the issuer's database, since a certificate record does not uniquely identify a browser. To check for duplication, the script downloaded at step 2 of the provisioning protocol containing the credit card number and the CSRF token also contains the key IDs found in all the certificate records that refer to the credit card record. The script queries the private key database through the IndexedDB API with each of the key IDs, and detects duplication if one of the queries retrieves an object containing a payment-credential private key.

7 Cardholder authentication protocol

The cardholder authentication protocol involves the issuer, the merchant, the cardholder, and the cardholder's browser, as illustrated in Figure 1.

The cardholder need not be logged in to the issuer on the browser to participate in the protocol.

Tasks assigned to the issuer by the protocol are best performed by software integrated into the web site of the issuer that has access to the issuer's database. Tasks assigned to the merchant may be performed by the merchant's own software or by shopping-cart software incorporated into the merchant's web site, or they may be outsourced to another party.

The protocol comprises the following steps, illustrated in Figure 1 with failure handling and other details omitted for clarity:

1. In the checkout page, the merchant asks the cardholder to enter credit card and cardholder data as usual. The credit card data includes the credit card number, the expiration date, and the security code. The cardholder data may include, at the merchant's discretion, the cardholder's name, address, phone number, etc. Once the cardholder has filled out all data required by the merchant, JavaScript code embedded in the checkout page extracts the IIN from credit card number and looks it up using the HTTP API of the scheme database, accessed via the XMLHttpRequest JavaScript API [9] or the Fetch [10] JavaScript API, to find out if the issuing bank supports the scheme. If so the JavaScript code enables a button that the cardholder can click for continuing the checkout process with cardholder authentication. It may also disable another button for continuing the process without authentication, or leave both buttons enabled.

2. If the cardholder clicks the button for continuing the checkout process with cardholder authentication, the back-end of the merchant's web site extracts the IIN from the credit card number and looks it up in the scheme database, using either a local copy or the HTTP API of the database, to obtain the URL of the cardholder authentication endpoint of the issuer. Then the merchant performs a POST redirection to the cardholder authentication endpoint, conveying the credit card and cardholder data, an encoded description of the transaction that includes a timestamp and a merchant-defined transaction ID, and a callback URL where the merchant expects to receive the result of the cardholder authentication; the callback URL is an https URL. We shall refer to the HTTP POST request to the cardholder authentication endpoint that results from the POST redirection as the *cardholder authentication request*.

3. The issuer searches its database for a credit card account record containing the card number, and checks if the expiration date, security code and cardholder data agree with the record. The issuer also checks if there are any certificate records that reference the credit account record, indicating that the credit card is CCHA-enabled. If no account record can be found for the credit card number, or the expiration date, security code or cardholder data do not agree with the record, or the card is not CCHA-enabled, the issuer performs a POST redirection to the callback URL, notifying the merchant that cardholder authentication has failed.

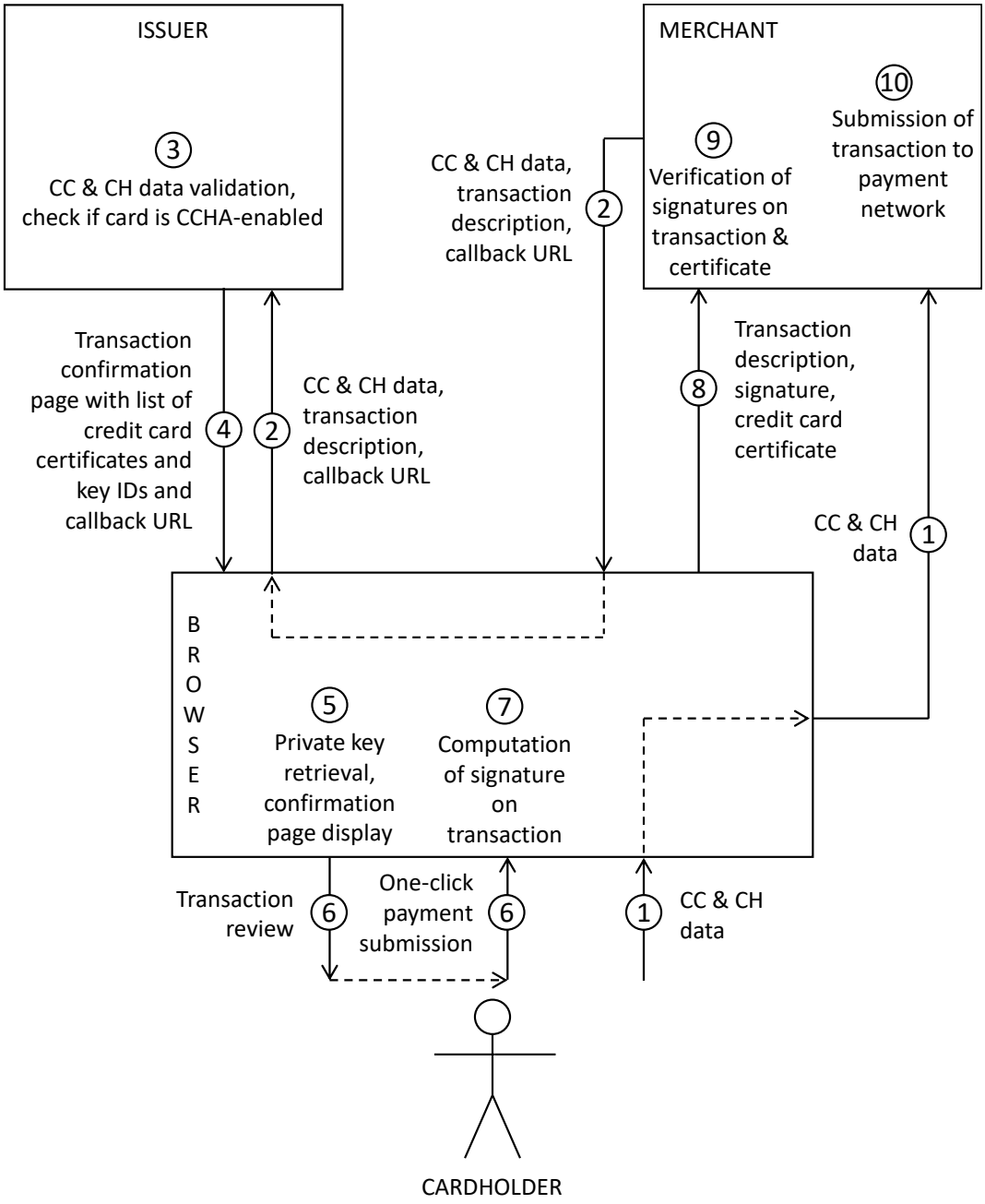

Figure 1: Cardholder authentication protocol. CC = credit card, CH = cardholder.

4. Otherwise the issuer compares the user-agent string of the current browser, found in the user-agent HTTP header of the carholder authentication request, to the user-agent strings found in the certificate records that reference the credit card account record. If the user-agent string in a certificate record refers to the same version or an older version of the same browser software as the user-agent in the cardholder authentication request, then that certificate record may have been created for the current browser. If no such certificate records are found, the issuer performs a POST redirection to the callback URL, notifying the merchant that cardholder authentication has failed. If one or more such records are found, the issuer brings any expired certificates in those records up to date, by reissuing them with an expiration time that lies a short time in the future; then it assembles a list containing the credit card certificates and key IDs found in those records, each certificate being paired with the corresponding key ID.

   Then the issuer decodes the description of the transaction and downloads an issuer-hosted confirmation page that includes the decoded description of the transaction to be displayed to the cardholder, as well as JavaScript code containing the encoded description to be signed, the callback URL, and the list of certificates and key IDs.

5. The confirmation page has a leading script that tries to retrieve a private key from the private key database by means of the IndexedDB API, using each of the key IDs in the list of certificates and key IDs. Since key IDs are universally unique and duplicate provisioning is avoided as described above in Section 6, at most one of the key IDs can retrieve a private key. If no private key is retrieved, the confirmation page is not displayed; instead, the leading script constructs a DOM form containing the transaction ID and submits it to the callback URL as an HTTP POST request, notifying the merchant that cardholder authentication has failed. If a key ID retrieves a private key, the private key and the credit card certificate paired with the key ID in the list are assigned to JavaScript global variables, and the confirmation page is displayed.

6. The confirmation page presents the decoded description of the transaction to the cardholder and identifies the merchant by means of the domain name in the callback URL. A script in the page may use a JavaScript implementation of the client side of the TLS protocol to establish a TLS connection to the callback URL for the purpose of retrieving the TLS certificate of the merchant; if the certificate is an extended-valuation (EV) certificate, information in the certificate may be used to further identify the merchant.

   The confirmation page contains a form with a visible submission button and three hidden inputs. The first hidden input contains the encoded description of the transaction. The second and third hidden inputs are to be filled in.

   The cardholder reviews the transaction and merchant identification in the confirmation page, and clicks on the submission button to submit the payment.

7. The click on the submission button launches a presubmission script that handles the `onsubmit` event of the form. The presubmission script computes a signature on the encoded transaction description using the private key retrieved in step 5 and assigned to a global variable. Then it fills in the second hidden input of the DOM form with

the signature, and the third hidden input with the credit card certificate assigned to a global variable in step 5.

8. The presubmission script submits the DOM form to the callback URL of the merchant as an HTTP POST request.

9. The merchant uses the public key in the credit card certificate to verify the signature on the encoded description of the transaction, and the public key associated with the issuer's IIN in the scheme database to verify the signature in the credit card certificate.

10. If verification of the signatures is successful the merchant retains the signed description of the transaction to be used later if necessary for dispute resolution. Then it submits the transaction to the payment network for processing as usual.

8 Defenses against same-origin attacks, malware, and physical capture

The browser restricts access to the private key component of a cryptographic payment credential to JavaScript code whose origin is the web site of the issuing bank. The *web origin* of JavaScript code [11, 12] is defined as a triple consisting of the protocol, hostname and TCP port portions of the URL used to download the HTML document where the JavaScript code is embedded. (In an https URL the protocol portion is "https", the TCP port is 443, and the hostname is the DNS domain name of the server that serves the HTML document in an HTTP response message.)

JavaScript that originates from the issuing bank should be deemed trustworthy, but it should be considered best practice to use available defenses against the possibility that it may be malicious. Malicious same-origin code might be the result of a cross-site scripting (XSS) vulnerability or an attack by a bank insider.

One defense againt a same-origin attack is to reduce the attack surface by using a subdomain of the DNS domain of the issuer for a portion of the issuer's web site dedicated to provisioning the cryptographic payment credential and authenticating the cardholder. Then only XSS vulnerabilities in that portion of the site can be exploited for same-origin attacks, and only insiders with access to that portion of the site can plant malicious same-origin JavaScript code.

Another defense against a same-origin attack is to generate the key pair using the Web Cryptography API [13], which produces `CryptoKey` objects containing the public and private keys. The private key can then be made *unextractable* from its CryptoKey object. This does not prevent malicious same-origin code from using the private key, but prevents such code from exfiltrating the key from the cardholder's machine so that it can be used elsewhere with more convenience for the attacker. A security downside of this defense, however, is that implementation of the Web Cryptography API varies from browser to browser and cannot be audited.

Another defense against exfiltration of the private key by same origin code is to store it in a *cryptographic module* implemented in a secure operating system (OS) different from the "rich" OS where the browser is running. The secure OS can be hosted, e.g., in a Trusted

Execution Environment (TEE), a Trusted Platform Module (TPM) or a USB dongle. The cryptographic module can be made accessible to scripts running in the browser by means of a browser extension that enforces the same-origin policy and exposes a JavaScript API that allows JavaScript code to use the private key but not to extract it. More details are provided below in Section 9.

A cryptographic module can prevent exfiltration not only by same-origin JavaScript code, but also by malware that runs in the rich OS but does not have privileged access to the secure OS. However it cannot prevent in-place use of the private key by such malware.

If the secure OS is hosted in hardware with some form of tamper resistance, the cryptographic module also provides some protection again physical exfiltration by an attacker who gains physical possession of the hardware. Tamper resistance is often featured by a TPM and sometimes by a TEE; and a USB dongle can be manufactured to provide tamper-resistance. Notice, however, that resistance to physical tampering is relative to the level of skill and dedication of the attacker. A presentation at BlackHat 2010 [14] showed how an attacker working alone with modest resources but high dedication was able to read the memory of a chip with a high-grade tamper-resistance certification [15].

Whether or not it is tamper-resistant, a USB dongle pluggable into a laptop provides protection against an attacker who steals the laptop merely by the fact that the dongle may not be plugged in when the laptop is stolen. The same is true, of course, for any kind of removable module, e.g. for a smart card accessed by means of a card reader.

A cryptographic module in a TEE or a TPM can be used with any browser installed on the machine where the module is located, and a cryptographic module in a USB dongle or other removable hardware can be used on multiple browsers installed in multiple devices. Storing the private key in a cryptographic module has the practical advantage that it can be used on multiple browsers without having to provision a separate cryptographic payment credential to each of the browsers.

9 How to use a cryptographic module when available

Supporting the use of a cryptographic module by means of a browser extension requires modifying the basic scheme described above to add the following functionality.

The script of step 2 of the provisioning protocol checks for the presence of the cryptographic module API exposed by the browser extension. If the API exists, the script uses it to request the creation of the key pair and obtain the key ID. In this case, the key ID is not used to retrieve the private key from the module: the private key never leaves the module. It is used instead to request the computation of the signature on the transaction description within the module using the private key. It is also used to retrieve the public key from the module, where it is stored together with the private key, so that it can be submitted to the issuer back-end in the CSR at step 3.

The browser extension enforces the same origin policy of the web, making the key pair usable only by scripts that have the same origin as the script that requested its creation.

The API call to generate the key pair takes as an argument the URL of the cardholder authentication endpoint. The browser extension verifies that the URL has the same web origin as the script that makes the API call, then stores an association between the key

ID and the URL. It uses the association to add an HTTP header containing the key ID to every request that targets the URL, just as an ordinary browser without any extension adds an HTTP header containing a cookie to every request that targets a URL within the scope of the cookie. Multiple key IDs may be associated with the same URL, if the same issuer provisions payment credentials for different credit cards to the same browser. In that case the value of the HTTP header is a list containing all those key IDs.

Steps 3 and 4 of the provisioning protocol are not modified, and result, as in the basic scheme, in the creation of a certificate record that references the credit card account record and contains the key ID and the credit card certificate.

At step 2 of the authentication protocol, the browser extension adds the HTTP header with all the key IDs associated with the URL of the cardholder authentication endpoint to the cardholder authentication request. At step 4 the issuer checks if one of those key IDs is present in a certificate record that references the credit card account record. If so the list of certificates and key IDs assembled by the issuer consists of that key ID paired with the certificate found in that certificate record. At step 5 the leading script detects the presence of the the cryptographic module API; then, instead of retrieving a private key using the IndexedDB API and assigning it to a global variable, it assigns the key ID itself to a global variable. At step 7 the presubmission script uses the key ID to request the computation of the signature on the transaction within the module, using the private key but without the private key leaving the module.

References

[1] Miles Brignall. MasterCard and Visa to simplify hated verification systems. The Guardian, 13 Nov 2014. https://www.theguardian.com/money/2014/nov/13/mastercard-visa-kill-off-verification-systems.

[2] Ben Laurie. More Banking Stupidity: Phished by Visa. March 28, 2009. https://www.links.org/?p=591.

[3] Jonathan Baker-Bates. Phishing with 3-D Secure. 13 August 2009. https://webtorque.org/phishing-with-3-d-secure/.

[4] Steven J. Murdoch and Ross Anderson. Verified by Visa and MasterCard SecureCode: or, How Not to Design Authentication, 2010.

[5] VISA. New and improved 3-D Secure. https://www.visaeurope.com/media/pdf/visa-infographic.pdf.

[6] EMVCo. EMV 3DS 2.0 Webcast. https://www.youtube.com/watch?v=Recwd2YOTsk&feature=youtu.be.

[7] Mozilla. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[8] W3C. Indexed Database API. https://www.w3.org/TR/IndexedDB/.

[9] Mozilla. XMLHttpRequest.
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[10] Mozilla. Fetch API.
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[11] A. Barth. The Web Origin Concept. IETF RFC 6454.
https://tools.ietf.org/html/rfc6454.

[12] Mozilla. Origin.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin.

[13] Mozilla Developer Network. Web Crypto API.
https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API.

[14] Christopher Tarnovsky. Hacking The Smartcard Chip (Blackhat 2010). Video broken into eight 10-minute segments. http://www.securitytube.net/video/945.

[15] Infineon. World's Most Stringent Security Tests Confirm Infineon's Security Competence in Smart Card ICs. http://www.smartcardalliance.org/articles/2006/01/04/worlds-most-stringent-security-tests-confirm-infineons-security-competence-in-smart-card-ics.