This invention is herby placed in the *Public Domain*

Payment Authorization Scheme

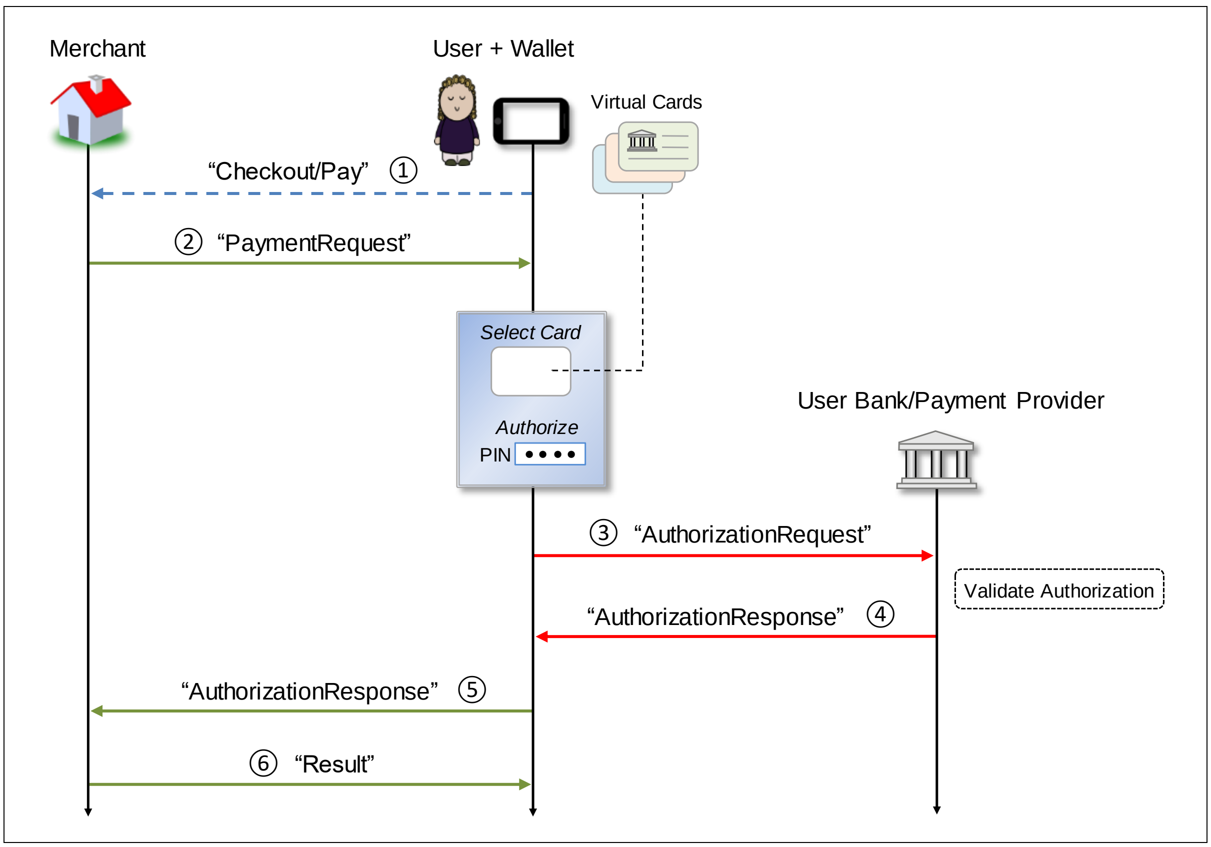
This document outlines a scheme for payment authorizations combining Security, Privacy and Efficient Communication through the use of cryptography.

Entities

A Merchant is an entity requesting a payment from a User. Merchant entities include on-line (Web) shops, local shops, and automated facilities like gas stations. Users are equipped with mobile devices hosting digital Wallets containing Virtual (payment) Cards issued by one or more User Banks.

Current Solutions

On the Web it is common to direct user authorizations to a payment provider without any intermediaries like shown in the state diagram below:

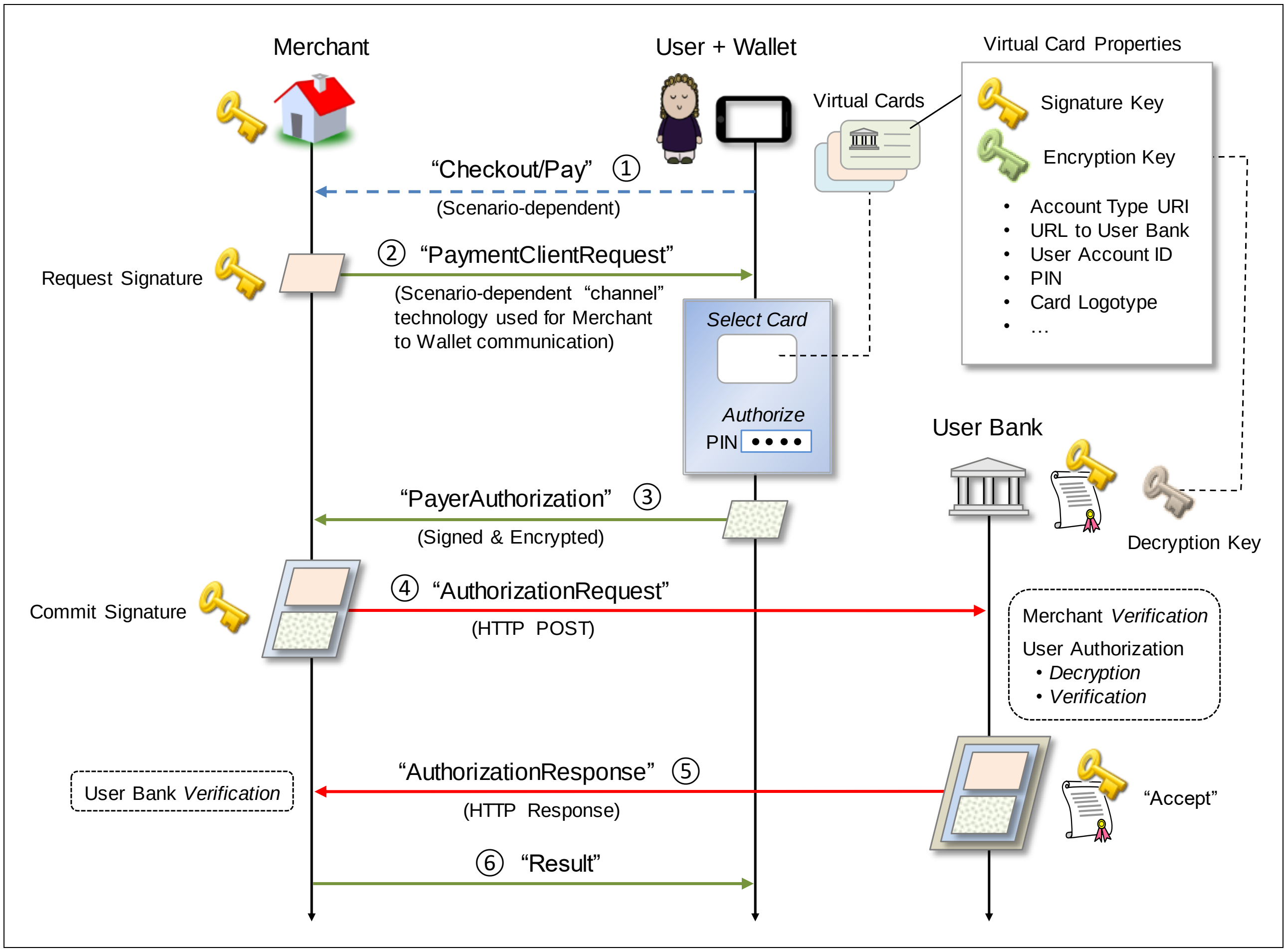


Although indeed working, there are also certain drawbacks with this approach like:

1. Requiring Wallets maintaining *two independent channels* during payment operations which greatly complicate state handling and user interfaces
2. Requiring mobile devices having *Internet connectivity*, something which cannot always be guaranteed to be available when paying in a local shop. Apple Pay (for example) does *not* impose such requirements
3. Introducing weakness by making Wallets responsible for handing over the result of a payment authorization to the Merchant. See step #4 and #5. *If this process fails, the User Bank and Merchant will have different opinions about the status of a successful payment operation*
4. Merchants get no opportunity discovering features and/or detecting potential interoperability issues with the selected User Bank *before* a transaction is committed

Enhanced Solution

To cope with these issues, the described solution utilizes a *single channel* between the Wallet and the outer world. The channel itself may use different technologies depending on scenario. On the mobile Web (using a browser in the mobile device), the channel would typically be some kind of Web to "App" interface like W2NB, while a local shop would rather use NFC and BLE. The state diagram below shows the enhanced scheme:



The reason for adding encryption to the plot is because *user payment data should not be given to Merchants* (they only need to know if a requested payment operation has been performed or not).

An observant reader probably wonders what happens if step #4 or #5 fails due to network errors. This is of course quite possible but here the server-to-server based approach offers additional advantages:

- Server-to-server connections are likely to be *faster and more reliable* than mobile networks
- Adding retransmissions and idempotent operation would be fairly simple, making the system *more fault tolerant*

Step #6 may fail which though in most cases would not lead to a disaster since the User presumably have a "session" with the Merchant which can be resumed in case there is a network glitch.

Detailed Operation

The following pages describe each step in the payment authorization scheme using a hypothetical payment system which though borrows heavily from a somewhat more sophisticated real-world implementation known as Saturn. Although this specification utilizes JSON, the concept is virtually independent of format. Also see Secure Authorization Objects.

1. Checkout/Pay

This operation is scenario-dependent and is not a part of the authorization scheme except for the end-result which is a freshly Merchant-signed **`PaymentClientRequest`** message used in the proceeding step.

2. PaymentClientRequest

The **PaymentClientRequest** message is delivered to the Wallet using the scenario-dependent channel which also launches the Wallet application:


```
{
  "@context": "https://example.com/paymentstd",
  "@qualifier": "PaymentClientRequest",
  "acceptedAccountTypes": ["https://supercard.com","https://bankdirect.net","https://unusualcard.org"],
  "paymentRequest": {
    "payee": {
      "commonName": "Demo Merchant",
      "id": "86344"
    },
    "amount": "599.00",
    "currency": "USD",
    "referenceId": "#1000000",
    "timeStamp": "2016-10-22T06:01:36Z",
    "signature": {
      "algorithm": "ES256",
      "publicKey": {
        "type": "EC",
        "curve": "P-256",
        "x": "rZ344aiTaOATmLBOdfYThvnQu_zyB1aJZrbbbks2P9I",
        "y": "lKOvfJdgN8WqEbXMDYPRSMsPicm0Tk10pmer9LxvxLg"
      },
      "value": "_O4Ta4idtMcAHcRnjyEHkOOkb2 … afRQkUjsnp2LY8wcOn7m4b8OSDA"
    }
  }
}
```


The **@context** and **@qualifier** properties are used throughout this specification as a way to identify the actual message (object) type.

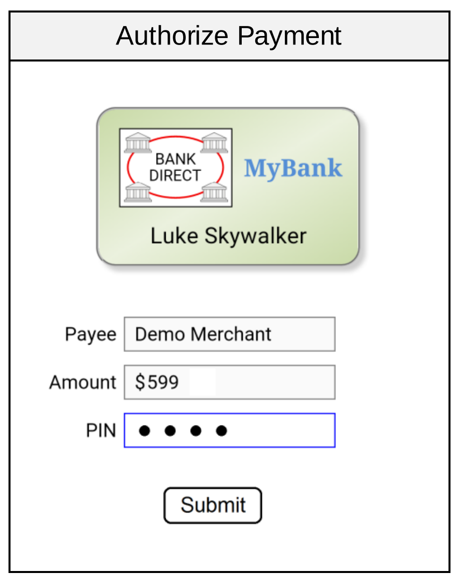
The **acceptedAccountTypes** array holds a list of payment methods expressed as URIs that the Merchant understands.

The **payee** object holds the identity of the Merchant.

The **referenceId** property holds the Merchant's identification of the particular payment request.

The **signature** object holds the Merchant's *Request Signature* (see state-diagram) in JCS format.

The images below show how the **PaymentClientRequest** object could be handled in a Wallet user interface:



Note the use of the Virtual Card property *Card Logotype* which enables banks to *personalize cards during enrollment*.

After the user have selected card and authorized the payment by a PIN or biometric (like touching a fingerprint reader), the Wallet software performs the operations specified in this section.

2.1 Hash Payment Requests

Run a hash function (here using the SHA256 method) over the **paymentRequest** object using the normalization method specified in JCS (excluding the handling of the **value** property).

2.2 Create Signed User Authorization Object

Create a signed user authorization object containing relevant data including the hash calculated in the previous step:


```
{
  "requestHash": {
    "algorithm": "S256",
    "value": "eYqbGYkHfAsOUTJiuqfU98Rou_mfn0etWUkvDVOF_Fw"
  },
  "domainName":  "demomerchant.com",
  "account": {
    "type": "https://bankdirect.net",
    "id": "8645-7800239403"
  },
  "encryptionParameters": {
    "algorithm": "A128CBC-HS256",
    "key": "Ivq5sSrtNNpOvN9t9_pRCfc6dqT3IuVg6H2h9NlHULs"
  },
  "responseToChallenge": [...],
  "timeStamp": "2016-10-22T08:02:18+02:00",
  "signature": {
    "algorithm": "ES256",
    "publicKey": {
      "type": "EC",
      "curve": "P-256",
      "x": "vlYxD4dtFJOp1_8_QUcieWCW-4KrLMmFL2rpkY1bQDs",
      "y": "fxEF70yJenP3SPHM9hv-EnvhG6nXr3_S-fDqoj-F6yM"
    },
    "value": "TDKWQb9idTyPXgpOgIxXeogt ... lhC5_dG3uU6MPmqjQLc7jju4f0Q"
  }
}
```


The **requestHash** object holds the hash of the **paymentRequest** object.

The **domainName** property holds the host (DNS) name of the invoking Merchant server.

The **account** object holds the *Account Type URI* and *User Account ID* of the selected Virtual Card (see state diagram).

The **encryptionParameters** object holds a random symmetric key which is generated for each Wallet invocation. For actual usage turn to the section Private Messaging.

The *optional* **responseToChallenge** property holds an array with response data to support Risk Based Authentication.

The **signature** object holds a JCS-formatted ECDSA signature based on the *Signature Key* of the selected Virtual Card (see state diagram).

Continued on the next page…

2.3 Encrypt User Authorization Data

Since the user authorization object contains potentially sensitive customer specific data (the *User Account ID* and *Signature Key*), it is encrypted using the *Encryption Key* of the selected Virtual Card. This data and some other related properties are then put in a newly created **`PayerAuthorization`** object:

```
{
  "@context": "https://example.com/paymentstd",
  "@qualifier": "PayerAuthorization",
  "providerUrl": "https://payproc.mybank.com/authorize",
  "accountType": "https://bankdirect.net",
  "encryptedAuthorization": {
    "algorithm": "A128CBC-HS256",
    "encryptedKey": {
      "algorithm": "ECDH-ES",
      "publicKey": {
        "type": "EC",
        "curve": "P-256",
        "x": "TfCrhFwZRU_ea7lUWwRi3HkuyT2yF9IxN5xKh2khjlk",
        "y": "nZFwxLP0TvFXD2xPKzRTIGevgLjpiMw2BP86hszj5x4"
      },
      "ephemeralKey": {
        "type": "EC",
        "curve": "P-256",
        "x": "D7zSvy3mbS4WbB2qgKwchLRwQFir5T_p09HpnAi_RqA",
        "y": "gkwNJ2o6BtUASkmp1DO4UvllsQL5zAzvVEHB7t0CqX0"
      }
    },
    "iv": "zyConPq8uA7GFjaTkta-qA",
    "tag": "S8zUQ3tioyYPzbtNBO6Ftw",
    "cipherText": "GLkd4uHnjqL_EX9tssDNLzsZj … s7u2ezBOibNoQN5V3cl2ieB-hLHj4XppJI"
  }
}
```

The **`providerUrl`** property holds a *URL to the User Bank* of the selected Virtual Card. Also see Authority Objects.

The **`accountType`** property holds a copy of the *Account Type ID* of the selected Virtual Card.

The **`encryptedAuthorization`** object holds a JEF structure where **`cipherText`** holds an encrypted version of the previously generated signed authorization object (after being serialized into a UTF-8 encoded byte array). The **`publicKey`** object holds the *Encryption Key* of the selected Virtual Card which is used for the ECDH operation which is the core of the encryption process.

After the preceding step has been performed, the Wallet transfers the **`PayerAuthorization`** message to the Merchant through the scenario-dependent channel.

Continued on the next page…

3 PayerAuthorization

Assuming that the received object appears to be correct (the authorization data cannot be checked for correctness since it is encrypted), the Merchant creates an **`AuthorizationRequest`** object which also embeds parts of the **`PayerAuthorization`** object. Finally, the Merchant adds a *Counter Signature* resulting in an object like the following:

```
{
   "@context": "https://example.com/paymentstd",
   "@qualifier": "AuthorizationRequest",
   "accountType": "https://bankdirect.net",
   "paymentRequest": {

        As provided by PaymentClientRequest…
   },
   "encryptedAuthorization": {

        As provided by the Wallet response…
   },
   "clientIpAddress": "224.165.21.50",
   "timeStamp": "2016-10-22T06:02:20Z",
   "signature": {
      "algorithm": "ES256",
      "publicKey": {
         "type": "EC",
         "curve": "P-256",
         "x": "rZ344aiTaOATmLBOdfYThvnQu_zyB1aJZrbbbks2P9I",
         "y": "lKOvfJdgN8WqEbXMDYPRSMsPicm0Tk10pmer9LxvxLg"
      },
      "value": "G4ct46eTx-GgF2qrSnHKRR9f9Ajd … ju85d56gSON2M3I20-u6sfcejw"
   }
}
```

The **`accountType`** property holds a copy of the same property received in the **`PayerAuthorization`** object.

Note that the **`publicKey`** *must be identical* to the public key featured in **`PaymentClientRequest`**.

To aid Risk Based Authentication the **`clientIpAddress`** (if applicable for the scenario) is supplied by the Merchant.

Next the **`AuthorizationRequest`** object is sent to the URL (Web address) specified in the **`providerUrl`** property of the **`PayerAuthorization`** object, typically using an HTTP POST operation over TLS (aka HTTPS).

Also see Authority Objects.

4 AuthorizationRequest

When the User Bank targeted by the **`providerUrl`** has received the **`AuthorizationRequest`** message, it performs the operations specified in this section.

4.1 Merchant Verification

In addition to verifying that the signature is technically correct, the authenticity if the Merchant must be checked. Exactly how this is done is outside of this specification but the data used is the **`publicKey`** and the **`payee`** data of the embedded **`paymentRequest`**. Also see Authority Objects.

4.2 User Authorization

Assuming the verification of the Merchant succeeded, the user authorization is checked for correctness and authenticity. Any errors in formats or signatures must be rejected.

The first step involves decryption of the **`encryptedAuthorization`** object which is done using a private key matching the supplied **`publicKey`**. The private key must (of course) only be known by the User Bank. The result of this operation should be identical to the object in step 2.2.

Next the authenticity of the User (payer) as expressed in the decrypted object is checked against a register holding the **`id`** and **`publicKey`** of all customers.

Finally the **requestHash** is compared against the calculated hash value of the received **paymentRequest** object.

4.3 Payment Operation

Assuming the previous step succeed the next step is to perform the actual payment operation including checking that the customer has enough funds to execute the request.

However, *how and where to send the money the Merchant requested is outside of this specification since it only deals with the authorization process*.

The Merchant's receive account could for example be discovered as a part of the Merchant authenticity verification process or be directly included in the **AuthorizationRequest**.

Existing payment networks would typically be used to perform the transfer itself.

If the steps above succeeded, the User Bank creates an **AuthorizationResponse** object which minimally would appear like this:


```
{
  "@context": "https://example.com/paymentstd",
  "@qualifier": "AuthorizationResponse",
  "@embedded": {
    "@context": "https://example.com/paymentstd",
    "@qualifier": "AuthorizationRequest",
                        .
         Exact copy of the AuthorizationRequest  object…
                        .
  },
  "accountReference": "***********9403",
  "referenceId": "#75643",
  "timeStamp": "2016-10-22T06:02:21Z",
  "signature": {
    "algorithm": "ES256",
    "signerCertificate": {
      "issuer": "CN=Payment Network Sub CA3,C=EU",
      "serialNumber": "1461174553809",
      "subject": "CN=mybank.com,2.5.4.5=#130434353031,C=FR"
    },
    "certificatePath": [
      "MIIBtTCCAVmgAwIBAgIGAVQ0 … 9Ly9t7A-jMuGl3FwxFeOawwmz1bM6",
      "MIIDcjCCAVqgAwIBAgIBAzANB … W9x5ZxVhvpP_We_5TddhlTUMNPvw"
    ],
    "value": "cdRqFlzVEou5Zj-EqWGCCLtxY … JkEBD4fFOqVnU9dstv_P2BoHQ"
  }
}
```


A noteworthy feature of the **AuthorizationResponse** object is that *it embeds the result of all previous steps of the authorization* which has applications both for auditing and debugging. This scheme also eliminates the need to reference external transaction objects, potentially simplifying system design. In this specification a property named **@embedded** is used for indicating the embedding of a complete message object.

The **accountReference** property holds a shortened version of the customer account to be used on Merchant receipts.

The **referenceId** property holds User Bank's identification of the particular authorization response.

Note that **signature** is JCS-formatted as in the previous examples but this time using X.509 certificates rather than public keys. That is, *participating banks are supposed to belong to a common PKI*.

The next step is simply returning the **AuthorizationResponse** object as the response to the HTTP POST in step 3.

5 AuthorizationResponse

After receiving the **AuthorizationResponse** object the Merchant verifies that the embedded **AuthorizationRequest** matches the one which was sent in step 3. If this succeeds the signature is verified and being checked for belonging to a for the merchant known payment network. Also see Authority Objects.

Although not shown here, the **`AuthorizationResponse`** should contain Merchant account information as well in the case it was not provided in the **`AuthorizationRequest`**.

If all tests succeed the Merchant fulfills the purchase, and returns a confirmation receipt to the user.

6 Receipt

The exact format and procedure for delivering a confirmation receipt to the user is scenario-dependent. On the Web, the Wallet would be closed and a success Web page would be shown to the user while a purchase in a local shop would preferably provide an indication in the Wallet itself.

Private Messaging and Risk Based Authentication

This section describes how the **`encryptionParameters`** in step 2.2 are to be used.

Occasionally users request high-value payments or exhibit “unusual” patterns like changing IP address zone from one country to another, non-neighboring country. In these circumstances banks typically want to “challenge” the user by for example answering specific questions, inputting a code received over SMS, or even asking for GPS location data.

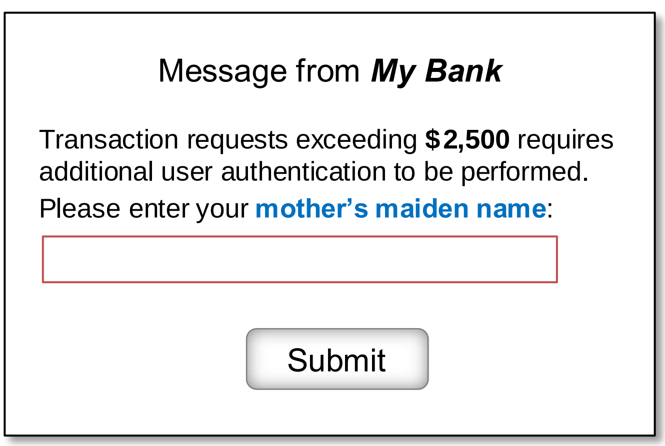
The described authorization scheme can support that by instead of returning an **`AuthorizationResponse`**, ignore the **`AuthorizationRequest`** and rather return a message like below:

```
{
  "@context": "https://example.com/paymentstd",
  "@qualifier": "ProviderUserResponse",
  "encryptedMessage": {
    "algorithm": "A128CBC-HS256",
    "iv": "cht7SYItQF8LO3QBg2bbbA",
    "tag": "33orw76LP7YibQqKPmKURA",
    "cipherText": "9PyK-rlhb41oBXVSdYa0Ats9 ... RBxdxqdGVbLf07Qw8Tr2LblxNYPUEc"
  }
}
```

`ProviderUserResponse` messages must be transferred “as is” to Wallets by Merchants as an alternative to step 5 and 6 processing. Since such messages pass through Merchants, they are encrypted by the User Bank using the encryption key supplied in the user authorization object.

For details on the encryption scheme used here see JEF.

After the Wallet has received a **`ProviderUserResponse`** message, it renders it:



When the user has supplied the requested information and hit “Submit” the authorization process is restarted from step 2.2 with the difference that the user response is added to the user authorization object as well.

`ProviderUserResponse` may also return information-only messages to the user like that there is not enough money on the account.

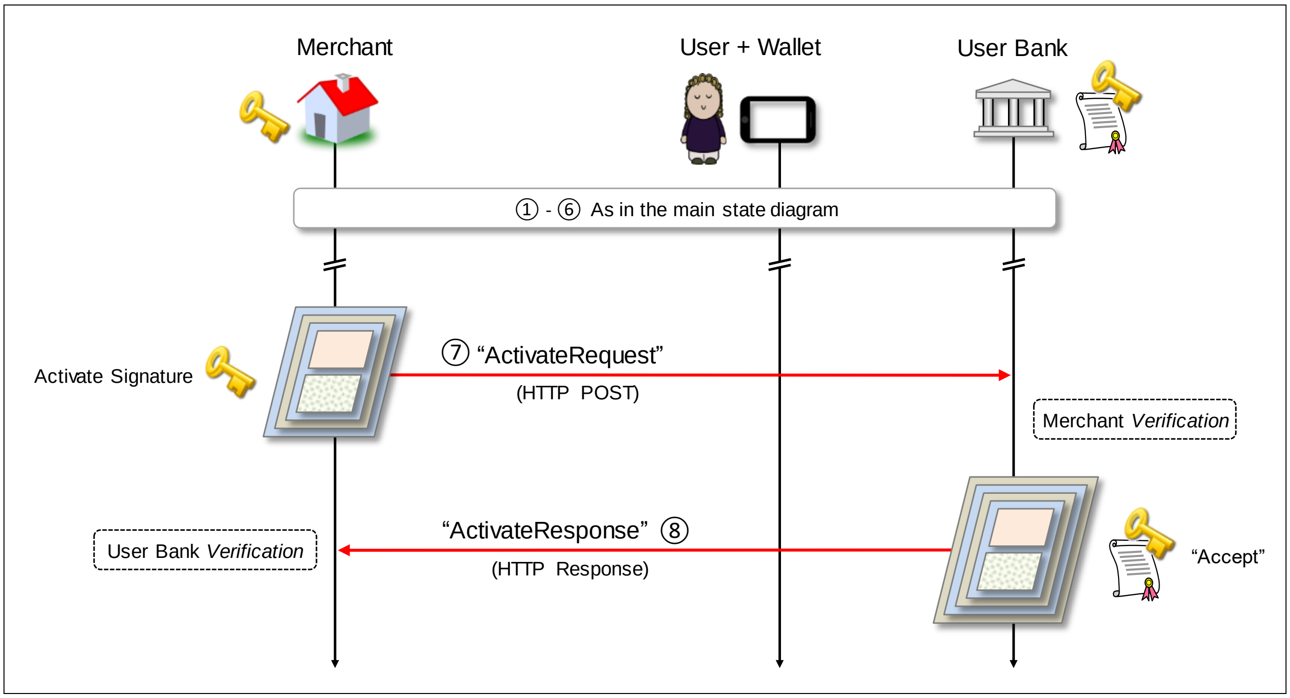
The exact format of the actual (decrypted) messages is not specified here, but the system which this document is derived from uses a subset of HTML5 for text while JSON elements are used for describing actions, labels, and input formats.

Secure Authorization Objects

If **`AuthorizationRequest`** objects are rather used to *reserve* funds, the resulting **`AuthorizationResponse`** objects may be stored by the Merchant without too much security concerns since they would in such a setting need yet another authentic counter signature by the Merchant in order to be "activated". This should be compared with storing credit card data which if stolen can be used by anybody.

Such a scheme could support *hotel bookings* and *automated gas stations needing preauthorization*, as well as *reoccurring payments*.

Below is an enhanced state diagram illustrating delayed activation:



Depending on the type of the transaction the **`ActivateRequest`** message may contain an **`amount`** which differs from (overrides) the original **`amount`** specified in the **`paymentRequest`** object in step 2.

Authority Objects

This specification only deals with the core payment authorization mechanism. By applying the techniques described in AUTH a more flexible and scalable operation can be achieved.

An obvious candidate is replacing the *URL to User Bank* in Virtual Cards (see state diagram) with a URL pointing to a *User Bank Authority Object* which can provide more upfront data to Merchants including underlying payment system support of the targeted User Bank.

The same applies to step 3 where the **`AuthorizationRequest`** object could be extended with an **`authorityUrl`** property pointing to a *Merchant Authority Object* making it easier for User Banks verifying the authenticity of Merchants.

Security Considerations

Since a valid transaction request is bound to both a Merchant and a User account, it appears that the described system effectively thwarts large scale attacks.

Risk Based Authentication and account limits should further reduce the ability to perform high-value attacks.

Out of Scope

How Virtual (payment) Cards are provisioned by User Banks and stored in Wallets *is not a part of this specification*. The latter is also valid for the provisioning of User Bank X.509 certificates as well as for Merchant signature keys.

Invention Summary

This invention disclosure describes the following sub inventions:

1. How virtual payment cards equipped with static encryption keys can be used for simplifying the communication between a digital wallet holding such cards and the outer world by encrypting sensitive user authorization data so that it can safely pass through untrusted parties like merchants. *There appears to be potential reducing security certifications (with respect to payments) for merchants to almost nothing including the need for specific payment terminals.*
2. How decentralized operation can be achieved by URLs stored in a virtual cards which point to the issuing bank, in contrast to deploying central registries where card numbers are mapped to banks. Also see Authority Objects.
3. How the encryption scheme referred to in claim #1 can also be used to securely transport an encryption key generated by the wallet to the user's bank in order to create a means for the bank sending private messages and/or perform risk based authentication *through the payment channel* rather than through a specific ditto.
4. How counter signing and message embedding can *simplify processing*, *auditing*, and *debugging* by eliminating external references to previous steps in a transaction.
5. How a secure authorization object can be created and at a later stage be activated by a counter signature.

References

Name	Description	URL
JCS	JSON Cleartext Signature	https://cyberphone.github.io/doc/security/jcs.html
JEF	JSON Encryption Format	https://cyberphone.github.io/doc/security/jef.html
JSON	JavaScript Object Notation	https://tools.ietf.org/rfc/rfc7159.txt
NFC	Near Field Communication	https://nfc-forum.org/
BLE	Bluetooth Low Energy	https://www.bluetooth.com/
SHA256	Secure Hash Algorithm with 256-bit result	https://tools.ietf.org/rfc/rfc6234.txt
ECDSA	Elliptic Curve Digital Signature Algorithm	https://tools.ietf.org/rfc/rfc5480.txt
ECDH	Elliptic Curve Diffie-Hellman algorithm	https://tools.ietf.org/rfc/rfc5480.txt
X.509	Digital Certificates and Support	https://tools.ietf.org/rfc/rfc5280.txt
AUTH	Authority Objects	https://cyberphone.github.io/doc/defensive-publications/authority-objects.pdf
W2NB	Web2Native Bridge	https://github.com/cyberphone/web2native-bridge
Saturn	Saturn Payment Authorization System	https://cyberphone.github.io/doc/saturn/

Permanent document URL: https://cyberphone.github.io/doc/defensive-publications/payment-authorization-scheme.pdf

Edited by: anders.rundgren.net@gmail.com