Workgroup:	Network Working Group	
Internet-Draft:	draft-solana-charge-00	
Published:	9 September 2026	
Intended Status:	Informational	
Expires:	13 March 2027	
Authors:	L. Galabru	I. Gitter
	Solana Foundation	*Solana Foundation*

Solana Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "solana" payment method within the Payment HTTP Authentication Scheme [I-D.httpauth-payment]. The client constructs and signs a native SOL or SPL token transfer on the Solana blockchain; the server verifies the payment and presents the transaction signature as proof of payment.

Two credential types are supported: `type="transaction"` (default), where the client sends the signed transaction to the server for broadcast, and `type="signature"` (fallback), where the client broadcasts the transaction itself and presents the on-chain transaction signature for server verification.

This document also defines an optional confidential charge profile, in which the transferred amount is encrypted on-chain using the Token-2022 Confidential Transfer extension [CONFIDENTIAL-TRANSFER]. Because a confidential transfer spans multiple transactions, this profile adds a third credential type, `type="bundle"`. The server — the payment recipient — confirms the paid amount by decrypting the amount credited to its own confidential account with its own key, so the amount never appears in cleartext on-chain.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 March 2027.

Copyright Notice

Table of Contents

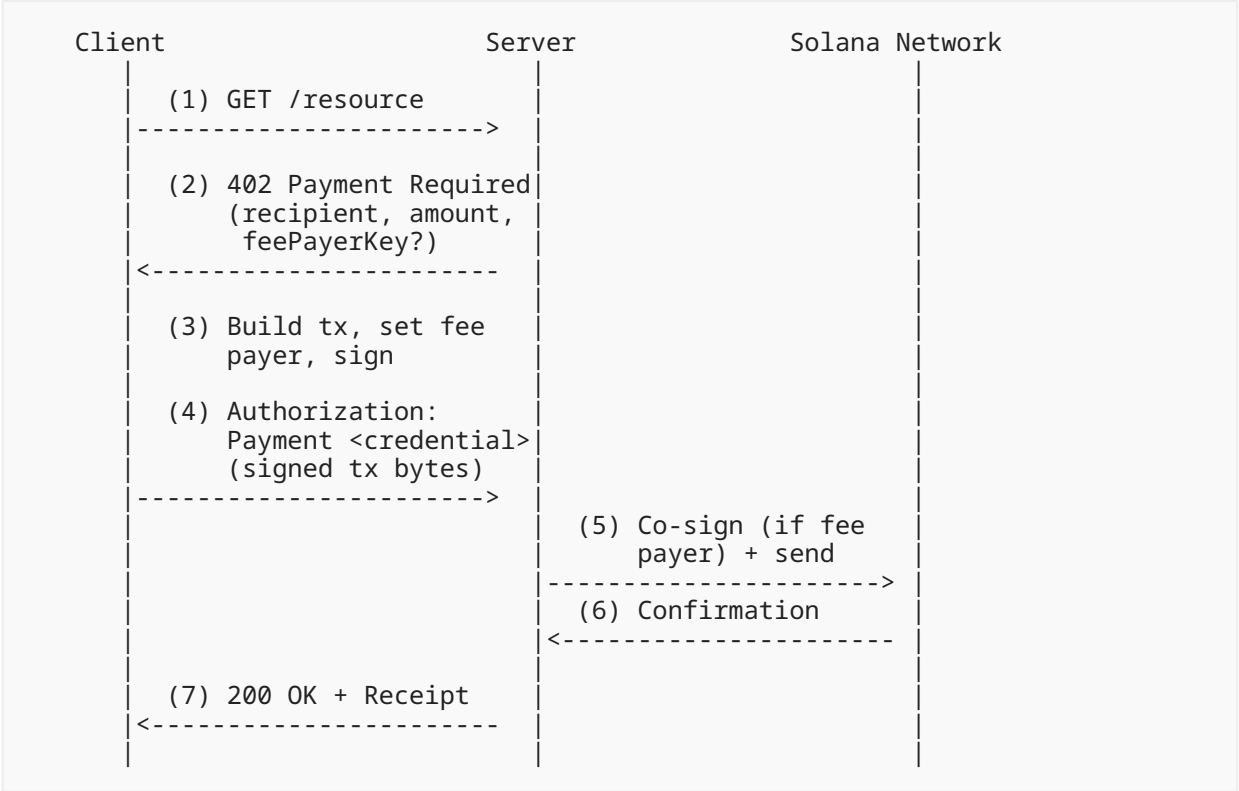
1. Introduction

HTTP Payment Authentication [I-D.httpauth-payment] defines a challenge-response mechanism that gates access to resources behind payments. This document registers the "charge" intent for the "solana" payment method.

Solana is a high-throughput blockchain with sub-second finality and low transaction fees [SOLANA-DOCS]. This specification supports payments in both native SOL and SPL tokens (including Token-2022 [SPL-TOKEN-2022]), making it suitable for micropayment use cases where fast confirmation and low overhead are important.

1.1. Pull Mode (Default)

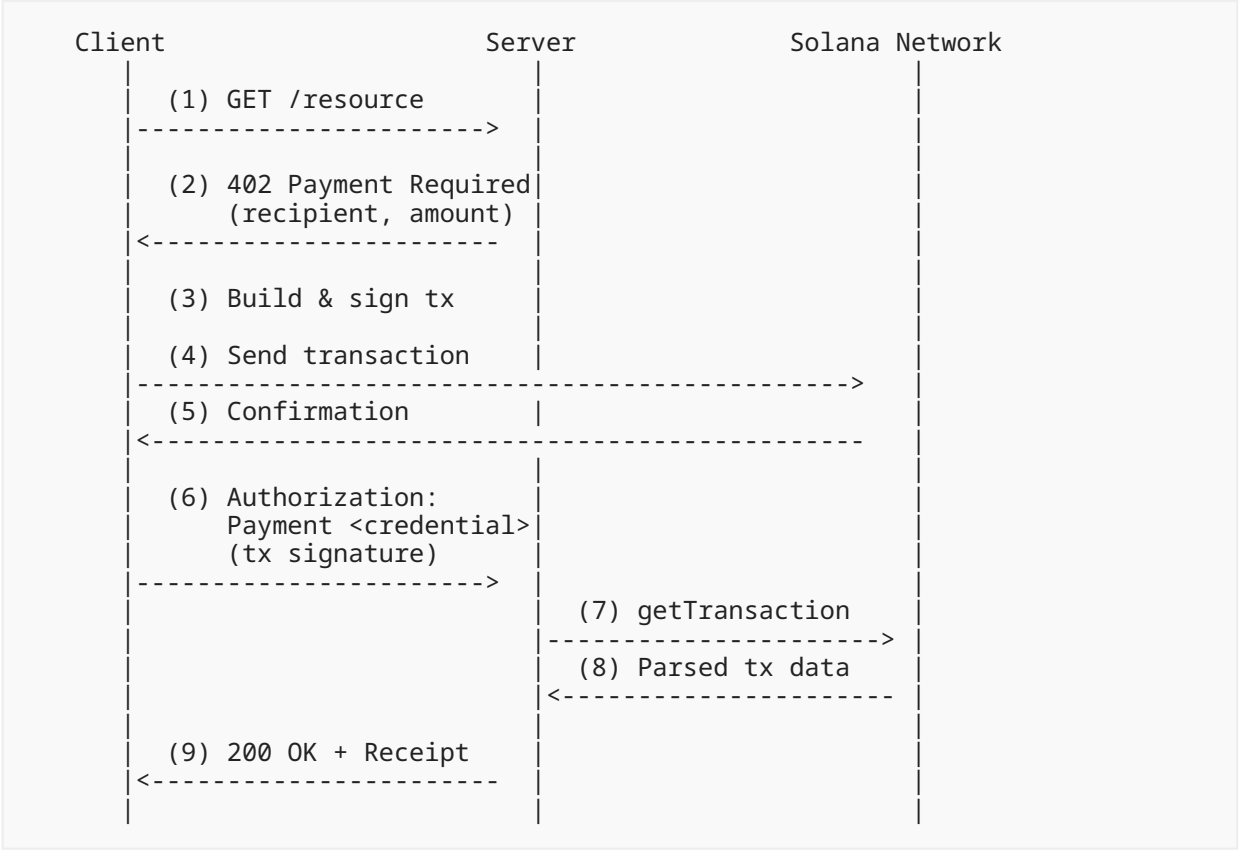
The default flow, called "pull mode", uses `type="transaction"` credentials. The client signs the transaction and the server "pulls" it for broadcast to the Solana network:



In this model the server controls transaction broadcast, enabling fee sponsorship (Section 9) and server-side retry logic. When `feePayer` is `true`, the challenge includes `feePayerKey` so the client sets the server as fee payer. The server co-signs with its fee payer key before broadcasting.

1.2. Push Mode (Fallback)

The fallback flow, called "push mode", uses `type="signature"` credentials. The client "pushes" the transaction to the network itself and presents the confirmed signature. The client broadcasts the transaction itself and presents the confirmed transaction signature:



This flow is useful when the client cannot or does not wish to delegate broadcast to the server. The server verifies the payment by fetching and inspecting the on-chain transaction via RPC.

1.3. Relationship to the Charge Intent

This document inherits the shared request semantics of the "charge" intent from [I-D.payment-intent-charge]. It defines only the Solana-specific `methodDetails`, `payload`, and verification procedures for the "solana" payment method.

2. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

Transaction Signature

A base58-encoded [BASE58] unique identifier for a Solana transaction, produced by the first signer. Serves as both the transaction identifier and proof of payment in this specification.

SPL Token A fungible token on Solana conforming to the SPL Token program [SPL-TOKEN] or the Token-2022 program [SPL-TOKEN-2022].

Associated Token Account (ATA) A deterministically derived token account for a given owner and mint, per the Associated Token Program. The address is a Program Derived Address (PDA) seeded by the owner's public key, the token mint, and the token program ID.

Lamports The smallest unit of native SOL. 1 SOL = 1,000,000,000 lamports.

Base Units The smallest transferable unit of an SPL token, determined by the token's decimal precision. For example, USDC uses 6 decimals, so 1 USDC = 1,000,000 base units.

Fee Payer An account that pays Solana transaction fees. When the server acts as fee payer, it adds its signature to the transaction before broadcasting, covering the transaction fee on behalf of the client.

Pull Mode The default settlement flow where the client signs the transaction and the server broadcasts it (`type="transaction"`). The server "pulls" the signed transaction from the credential. Enables fee sponsorship and server-side retry logic.

Push Mode The fallback settlement flow where the client broadcasts the transaction itself and presents the confirmed signature (`type="signature"`). The client "pushes" the transaction to the network directly. Cannot be used with fee sponsorship.

Confidential Transfer A Token-2022 transfer in which the transferred amount is encrypted on-chain using the Confidential Transfer extension [CONFIDENTIAL-TRANSFER]. The amount is hidden from public observers; only the sender, the receiver, and a designated auditor can recover it.

Confidential Token Account A Token-2022 associated token account that has been configured for confidential transfers via the `ConfigureAccount` operation, holding an encrypted available balance, an encrypted pending balance, and the owner's ElGamal public key. On mints that do not auto-approve, the account **MUST** also be approved by the mint's confidential-transfer authority before it can send or receive.

ElGamal Public Key A twisted-ElGamal public key bound to a confidential token account. Transfer amounts are encrypted under the sender's, the receiver's, and (when configured) the auditor's ElGamal public keys.

Auditor A party optionally designated by the mint's `ConfidentialTransferMint` configuration whose ElGamal public key is included in every confidential transfer, letting the holder of the corresponding secret decrypt transferred amounts. The auditor is the mint **issuer's** compliance facility; the charge server does NOT act as the auditor and plays no auditor role in this specification.

Pending Balance The portion of a confidential account's balance that has received incoming confidential transfers but is not yet spendable. The account owner converts pending balance into spendable available balance with the `ApplyPendingBalance` operation; no other party can perform this step.

Proof Context State Account A short-lived account into which the ZK ElGamal Proof Program [ZK-ELGAMAL-PROOF] records a verified proof so that a later instruction in the same or a subsequent transaction can reference it. Because confidential-transfer proofs are too large to share a single transaction with the transfer itself, they are verified into context state accounts first and closed afterward to reclaim rent.

4. Intent Identifier

The intent identifier for this specification is "charge". It **MUST** be lowercase.

5. Intent: "charge"

The "charge" intent represents a one-time payment gating access to a resource. The client builds and signs a Solana transfer transaction, then either sends the signed transaction bytes to the server for broadcast (`type="transaction"`) or broadcasts the transaction itself and sends the on-chain signature (`type="signature"`). The server verifies the transfer details and returns a receipt.

6. Encoding Conventions

All JSON [RFC8259] objects carried in auth-params or HTTP headers in this specification **MUST** be serialized using the JSON Canonicalization Scheme (JCS) [RFC8785] before encoding. JCS produces a deterministic byte sequence, which is required for any digest or signature operations defined by the base spec [I-D.httpauth-payment].

The resulting bytes **MUST** then be encoded using base64url [RFC4648] Section 5 without padding characters (`=`). Implementations **MUST NOT** append `=` padding when encoding, and **MUST** accept input with or without padding when decoding.

This encoding convention applies to: the `request` auth-param in `WWW-Authenticate`, the credential token in `Authorization`, and the receipt token in `Payment-Receipt`.

7. Request Schema

7.1. Shared Fields

The `request` auth-param of the `WWW-Authenticate: Payment` header contains a JCS-serialized, base64url-encoded JSON object (see Section 6). The following shared fields are included in that object:

amount **REQUIRED**. The payment amount in base units, encoded as a decimal string. For native SOL, the amount is in lamports. For SPL tokens, the amount is in the token's smallest unit (e.g., for USDC with 6 decimals, "1000000" represents 1 USDC). The value **MUST** be a positive integer that fits in a 64-bit unsigned integer (max 18,446,744,073,709,551,615).

currency **REQUIRED**. For native SOL, **MUST** be the lowercase string `"sol"`. For SPL tokens, **MUST** be the base58-encoded mint address (e.g., `"EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v"` for USDC). The mint address uniquely identifies the token and is used by the client to construct the transfer instruction. **MUST NOT** exceed 128 characters.

description **OPTIONAL**. A human-readable memo describing the resource or service being paid for. **MUST NOT** exceed 256 characters.

recipient **REQUIRED**. The base58-encoded public key of the account receiving the payment. For native SOL transfers, this is the destination account. For SPL token transfers, this is the owner of the destination associated token account, not the ATA address itself.

externalId **OPTIONAL**. Merchant's reference (e.g., order ID, invoice number), per [I-D.payment-intent-charge]. May be used for reconciliation or idempotency. **MUST NOT** exceed 566 bytes (Solana Memo Program limit). When present, clients **SHOULD** include this value as a Memo Program instruction in the transaction, making it visible on-chain for auditing and reconciliation. Servers **MAY** verify the memo matches the `externalId` from the challenge.

7.2. Method Details

The following fields are nested under `methodDetails` in the request JSON:

network **OPTIONAL**. Identifies which Solana cluster the payment should be made on. **MUST** be one of "mainnet", "devnet", or "localnet". Defaults to "mainnet" if omitted. Clients **MUST** reject challenges whose network does not match their configured cluster.

decimals Conditionally **REQUIRED**. The number of decimal places for the token (0–9). **MUST** be present when `currency` is a mint address; **MUST** be absent when `currency` is `"sol"`. Used by the client to construct a `TransferChecked` instruction.

tokenProgram **OPTIONAL**. The base58-encoded program ID of the token program governing the token. **MUST** be either the Token Program (`TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`) or the Token-2022 Program (`TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb`). If omitted, clients **MUST** determine the correct token program by fetching the mint account from the network and inspecting its owner program. If that lookup fails, returns an unexpected owner, or cannot be verified, clients **MUST** reject the challenge rather than falling back to the Token Program. Servers **SHOULD** include this field as a hint to avoid the extra RPC lookup. **MUST NOT** be present when `currency` is `"sol"`.

feePayer **OPTIONAL**. A boolean indicating whether the server will pay transaction fees on behalf of the client. Defaults to `false` if omitted. When `true`, the `feePayerKey` field **MUST** also be present. See Section 9.

feePayerKey Conditionally **REQUIRED**. The base58-encoded public key of the server's fee payer account. **MUST** be present when `feePayer` is `true`; **MUST** be absent when `feePayer` is `false` or omitted. The client uses this key as the transaction fee payer when constructing the transaction.

splits **OPTIONAL**. An array of at most 8 additional payment splits. Each entry is a JSON object with the following fields:

- `recipient` (**REQUIRED**): Base58-encoded public key of the split recipient.
- `amount` (**REQUIRED**): Amount in the same base units and asset as the primary `amount`.
- `memo` (**OPTIONAL**): Human-readable label for this split (e.g., "platform fee", "referral"). **MUST NOT** exceed 566 bytes (Solana Memo Program limit).
- `ataCreationRequired` (**OPTIONAL**): Boolean. Defaults to `false`. When `true`, the client **MUST** include an idempotent Associated Token Account creation instruction for this split recipient's ATA before the split transfer. This field **MUST NOT** be `true` unless `currency` is an SPL token mint address. In fee-sponsored pull mode (`feePayer: true`), this field is the only authorization for the server fee payer to fund split-recipient ATA creation. See Section 9.5.

When present, the client **MUST** include a transfer instruction for each split in addition to the primary transfer to `recipient`. All splits use the same asset as the primary payment (native SOL or the token from `currency`).

The top-level `amount` is the total the client pays. The sum of all split amounts **MUST NOT** exceed `amount`. The primary `recipient` receives `amount` minus the sum of all split amounts; this remainder **MUST** be greater than zero. Servers **MUST** reject challenges where splits consume the entire amount. Servers **MUST** verify each split transfer on-chain during credential verification. If the same recipient appears more than once in `splits`, each occurrence is a distinct payment leg and **MUST** be verified separately; servers **MUST NOT** implicitly aggregate such entries.

This mechanism is a Solana-specific extension to the base `charge` intent. It can be used for fee payer cost recovery, platform fees, revenue sharing, or referral commissions.

recentBlockhash **OPTIONAL**. A base58-encoded recent blockhash for the client to use when constructing the transaction. When provided, clients **SHOULD** use this blockhash instead of fetching one from an RPC node. This avoids an extra RPC round-trip and ensures the server can verify blockhash freshness. This field is advisory and short-lived; it **MUST NOT** be assumed to remain valid for the full lifetime of the payment challenge. If omitted, clients **MUST** fetch a recent blockhash themselves.

confidential **OPTIONAL**. A boolean indicating that the charge **MUST** be settled as a Token-2022 confidential transfer (Section 10), in which the transferred amount is encrypted on-chain. Defaults to `false` if omitted. When `true`: `currency` **MUST** be the mint address of a Token-2022 mint whose `ConfidentialTransferMint` extension is enabled; `tokenProgram`, if present, **MUST** be the Token-2022 Program; the credential **MUST** use `type="bundle"` (Section 8.3); and `splits` **MUST NOT** be present. Servers **MUST** reject a confidential challenge that violates any of these constraints. **MUST NOT** be `true` when `currency` is `"sol"`.

auditorElgamalPubkey **OPTIONAL**. The base64-encoded twisted-ElGamal public key of the mint's confidential-transfer auditor, when the mint configures one. This is informational: it lets the client confirm it is transmitting on the mint it expects. It is NOT used for charge verification — the auditor is the mint issuer's compliance facility, not the server's amount-check mechanism (see Section 10.3). When present, clients **SHOULD** verify it matches the on-chain `ConfidentialTransferMint` auditor key. **MUST** be absent when `confidential` is not `true`.

recipientElgamalPubkey **OPTIONAL**. The base64-encoded twisted-ElGamal public key of the recipient's confidential token account, supplied as a hint to save an RPC lookup. When present, clients **MUST** verify it matches the recipient's on-chain confidential account state before use. Regardless of this hint, clients **MUST** confirm the recipient has a configured (and, on mints that do not auto-approve, approved) confidential token account; if it does not, the client **MUST** reject the challenge, because confidential transfers cannot be received by an unconfigured account (Section 10.2). **MUST** be absent when `confidential` is `false` or omitted.

7.2.1. Native SOL Example

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet"
  }
}
```

This requests a transfer of 0.01 SOL (10,000,000 lamports).

7.2.2. SPL Token Example

```
{
  "amount": "1000000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Premium API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA"
  }
}
```

This requests a transfer of 1 USDC (1,000,000 base units).

7.2.3. Fee Sponsorship Example

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet",
    "feePayer": true,
    "feePayerKey": "9aE3Fg7HjKLmNpQr5TuVwXyZ2AbCdEf8GhIjKlMnOp1R"
  }
}
```

This requests a transfer of 0.01 SOL where the server pays transaction fees.

7.2.4. Payment Splits Example

```
{
  "amount": "1050000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Marketplace purchase",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "splits": [
      { "recipient": "3pF8Kg2aHbNvJkLMwEqR7YtDxZ5sGhJn4UV6mWcXrT9A",
"amount": "50000", "memo": "platform fee" }
    ]
  }
}
```

This requests a total payment of 1.05 USDC. The platform receives 0.05 USDC and the primary recipient (seller) receives 1.00 USDC.

7.2.5. Split Recipient ATA Creation Example

```
{
  "amount": "1000000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Marketplace purchase with bridge settlement",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA",
    "feePayer": true,
    "feePayerKey": "Gh9ZwEmdLJ8DscKNTkTqPbNwLNNBjuSzaG9Vp2KGtKJr",
    "splits": [
      {
        "recipient": "3pF8Kg2aHbNvJkLMwEqR7YtDxZ5sGhJn4UV6mWcXrT9A",
        "amount": "990000",
        "memo": "bridge deposit",
        "ataCreationRequired": true
      }
    ]
  }
}
```

This requests a total payment of 1 USDC. The bridge deposit split receives 0.99 USDC and the primary recipient receives 0.01 USDC. Because the split sets `ataCreationRequired: true`, the fee payer authorizes funding the split recipient's ATA if it does not already exist. The top-level recipient is not covered by this authorization.

7.2.6. Confidential Transfer Example

```
{
  "amount": "1000000",
  "currency": "HVWf8JmLoHs99Lw8Psf3fyqAtA4crWxCPkrmSdNjhNH3",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Confidential API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb",
    "feePayer": true,
    "feePayerKey": "9aE3Fg7HjKLmNpQr5TuVwXyZ2AbCdEf8GhIjKlMnOp1R",
    "confidential": true
  }
}
```

This requests a confidential transfer of 1 token (1,000,000 base units). The on-chain transfer encrypts the amount; the server, as the recipient, recovers and verifies it by decrypting the amount credited to its own confidential account with its own ElGamal key. See Section 10.

8. Credential Schema

The `Authorization` header carries a single base64url-encoded JSON token (no auth-params). The decoded object contains the following top-level fields:

challenge **REQUIRED**. An echo of the challenge auth-params from the `WWW-Authenticate` header: `id`, `realm`, `method`, `intent`, `request`, and (if present) `expires`. This binds the credential to the exact challenge that was issued.

source **OPTIONAL**. A payer identifier string, as defined by [I-D.httpauth-payment]. Solana implementations **MAY** use the payer's base58-encoded public key or a DID.

payload **REQUIRED**. A JSON object containing the Solana-specific credential fields. The `type` field determines which additional fields are present. Three payload types are defined: `"transaction"` (default) and `"signature"` (fallback) for ordinary transfers, and `"bundle"` for confidential transfers (Section 8.3).

8.1. Transaction Payload — Pull Mode

In pull mode (`type="transaction"`), the client sends the signed transaction bytes to the server for broadcast. The `transaction` field contains the base64-encoded serialized signed transaction.

Field	Type	Required	Description
`type`	string	**REQUIRED**	`"transaction"`
`transaction`	string	**REQUIRED**	Base64-encoded serialized signed transaction bytes (max 1232 bytes decoded)

Table 1

The transaction **MUST** be a valid Solana versioned transaction that does not exceed the 1232-byte transaction size limit. containing the transfer instruction(s) matching the challenge parameters. The client **MUST** sign the transaction with the transfer authority key. When `feePayer` is `false` or absent, the client **MUST** also be the fee payer and the transaction **MUST** be fully signed. When `feePayer` is `true`, the transaction **MUST** set the server's `feePayerKey` as fee payer, and the client signs only as transfer authority; the server adds the fee payer signature before broadcasting (see Section 9).

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "AQAAAA...base64-encoded-signed-tx..."
  }
}
```

8.2. Signature Payload — Push Mode

In push mode (`type="signature"`), the client has already broadcast the transaction to the Solana network. The `signature` field contains the base58-encoded transaction signature for the server to verify on-chain.

Field	Type	Required	Description
`type`	string	**REQUIRED**	`"signature"`
`signature`	string	**REQUIRED**	Base58-encoded Solana transaction signature

Table 2

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "signature",
    "signature": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny..."
  }
}
```

8.3. Bundle Payload — Confidential Transfers

When `methodDetails.confidential` is `true`, the credential **MUST** use `type="bundle"`. A confidential transfer cannot be expressed as a single transaction: its validity, equality, and range proofs are too large to share a transaction with the transfer instruction, so they are first verified into proof context state accounts (Section 10). The `transactions` field carries the ordered list of signed transactions that implement the transfer.

Field	Type	Required	Description
`type`	string	**REQUIRED**	`"bundle"`
`transactions`	array	**REQUIRED**	Ordered, non-empty array of base64-encoded serialized signed transactions. Each element **MUST** individually satisfy the 1232-byte transaction size limit.

Table 3

The transactions **MUST** be ordered for sequential submission: each proof context state account **MUST** be created and its proof verified before the transaction that consumes it, and any context-account close instructions **MUST** appear no earlier than the transaction that last references the account. The final transaction in the array **MUST** contain the confidential `Transfer` (or `TransferWithFee`) instruction. The server submits the transactions in array order, waiting for the required commitment level on each before submitting the next (see Section 11.5 and Section 12.3).

Confidential charges are normally fee-sponsored (`feePayer: true`), because the paying client typically holds no SOL. Every transaction in the bundle **MUST** then set the server's `feePayerKey` as fee payer. The client signs each transaction only as the transfer authority and for the ephemeral proof / record account keypairs it generates; the server co-signs the fee-payer slot before broadcast.

Because the client funds nothing, the server's `feePayerKey` **MUST** also be the rent funder for the proof context and record accounts and their authority (`context_state_authority` and the record-account authority), and every close instruction **MUST** return the reclaimed lamports to the server. The server does NOT shift rent to the client (it has no SOL); instead it protects itself from rent drain by (a) verifying each transaction contains only allow-listed, non-draining instructions before co-signing (Section 11.5), and (b) reclaiming rent from accounts orphaned by partial failures — which it can do because it is their authority (see Section 14.10).

The server absorbs the (small) SOL transaction fee for a confidential charge: it cannot be recovered through a stablecoin `splits` entry (splits are forbidden for confidential charges, Section 10), and the confidential `Transfer` is single-recipient. Servers **SHOULD** price this into their fee model rather than attempt on-chain recovery.

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "bundle",
    "transactions": [
      "AQAAAA...proof-context-setup-tx...",
      "AQAAAA...confidential-transfer-tx..."
    ]
  }
}
```

8.4. Limitations of Push Mode

The `type="signature"` credential has the following limitations:

- **MUST NOT** be used when `feePayer` is `true` in the challenge request. Since the client has already broadcast the transaction, the server cannot add its fee payer signature. Servers **MUST** reject `type="signature"` credentials when the challenge specifies `feePayer: true`.
- The server cannot modify or enhance the transaction (e.g., add priority fees, adjust compute units, or retry with different parameters).

9. Fee Sponsorship

When a challenge includes `feePayer: true` in `methodDetails`, the server commits to paying Solana transaction fees on behalf of the client. This section describes the fee sponsorship mechanism.

9.1. Server-Paid Fees

When `feePayer` is `true`:

1. **Client constructs transaction**: The client builds the transfer transaction with the server's `feePayerKey` set as the transaction fee payer. The client's account is the transfer authority but NOT the fee payer.
2. **Client partially signs**: The client signs the transaction with only its own key (the transfer authority). The fee payer signature slot remains empty.
3. **Client sends credential**: The client sends the partially signed transaction as a `type="transaction"` credential.
4. **Server adds fee payer signature**: The server verifies the transaction contents, then signs with the fee payer key to complete the transaction.

5. **Server broadcasts**: The fully signed transaction (containing both the client's transfer authority signature and the server's fee payer signature) is broadcast to the Solana network.

9.2. Client-Paid Fees

When `feePayer` is `false` or omitted, the client **MUST** set itself as the fee payer and fully sign the transaction. The server broadcasts the transaction as-is without adding any signatures.

9.3. Server Requirements

When acting as fee payer, servers:

- **MUST** maintain sufficient SOL balance in the fee payer account to cover transaction fees
- **MUST** verify the transaction contents before signing (see Section 11.1)
- **SHOULD** implement rate limiting to mitigate fee exhaustion attacks (see Section 14.6)

9.4. Client Requirements

- When `feePayer` is `true`: clients **MUST** set `feePayerKey` from `methodDetails` as the transaction fee payer and **MUST** sign only with the transfer authority key. Clients **MUST** use `type="transaction"` credentials.
- When `feePayer` is `false` or omitted: clients **MUST** set themselves as the fee payer and fully sign the transaction. Clients **MAY** use either `type="transaction"` or `type="signature"` credentials.

9.5. Split Recipient ATA Creation

ATA creation is permitted only as setup for payment recipients. It does not authorize the creation of token accounts for unrelated owners. The challenge expresses a required split-recipient ATA setup on the split entry itself with `ataCreationRequired: true`.

Every ATA creation instruction in an SPL token payment transaction **MUST** satisfy all of the following:

- The instruction **MUST** be the Associated Token Program's idempotent create instruction.
- The instruction owner **MUST** be a recipient listed in `splits`, subject to the fee payer restrictions below.
- The instruction mint **MUST** be the challenge `currency`.
- The instruction token program **MUST** be the challenge `tokenProgram`. If `tokenProgram` is omitted, the token program resolved from the mint account is used.
- The ATA address **MUST** be the canonical Associated Token Account PDA for the owner, mint, and token program.
- The instruction payer **MUST** be the transaction fee payer.

When a split sets `ataCreationRequired: true`:

- The challenge `currency` **MUST** be an SPL token mint address.

- The client **MUST** include an ATA creation instruction for that split recipient before the split transfer.
- The ATA creation instruction does not create an additional payment recipient. The client **MUST** still include the split's `transferChecked` payment instruction.

In fee-sponsored pull mode (`feePayer: true`), the server fee payer only authorizes ATA creation for split recipients whose split entry sets `ataCreationRequired: true`. Clients **MUST NOT** include fee-payer-funded ATA creation instructions for the top-level `recipient`, unmarked split recipients, or arbitrary owners. If the top-level recipient's ATA does not exist, the server **MUST NOT** issue a challenge that requires creating it.

When the client is the transaction fee payer (`feePayer` is `false` or omitted), clients **MAY** include ATA creation instructions only for split recipients, and **MUST** include one for each split whose entry sets `ataCreationRequired: true`. Clients **MUST NOT** include ATA creation instructions for the top-level `recipient` or any other owner.

10. Confidential Transfers

When a challenge sets `methodDetails.confidential` to `true`, the charge **MUST** be settled as a Token-2022 Confidential Transfer [CONFIDENTIAL-TRANSFER]. The transferred amount is encrypted on-chain under twisted-ElGamal public keys and does not appear in cleartext in the transaction. This profile applies only to Token-2022 SPL tokens; it **MUST NOT** be used for native SOL.

10.1. Overview

A confidential transfer differs from an ordinary `transferChecked` in three ways that this specification must accommodate:

1. **The amount is encrypted.** The server cannot read the transferred amount from parsed transaction data. Instead, the server — which is the payment recipient — confirms the amount it received by decrypting its own confidential balance with its own ElGamal key (see Section 11.5).
2. **The transfer spans multiple transactions.** The transfer requires a ciphertext-validity proof, a ciphertext-ciphertext equality proof, and a range proof (and, on mints with a confidential transfer fee, additional fee proofs). These do not fit in a single transaction and are verified into proof context state accounts first. The credential therefore carries a transaction bundle (Section 8.3).
3. **Delivery is two-phase.** Funds received by a confidential transfer land in the recipient's pending balance and become spendable only after the recipient applies them. The receipt reflects this (see Section 12.8).

10.2. Prerequisites

A confidential charge can only succeed when all of the following hold. Servers **MUST NOT** issue a confidential challenge, and clients **MUST** reject one, unless they are satisfied:

- The mint identified by `currency` is owned by the Token-2022 Program and has the `ConfidentialTransferMint` extension enabled.
- The sender (client) holds a configured confidential token account for the mint, with sufficient confidential available balance to cover the amount.
- The recipient holds a configured confidential token account for the mint. On a mint that does not auto-approve new accounts, that account **MUST** already be approved by the mint's confidential-transfer authority. The server or client cannot configure or approve the recipient's account on its behalf: the recipient owns the ElGamal secret, and approval is the mint authority's prerogative.

Account configuration (`ConfigureAccount`), approval (`ApproveAccount`), deposit, and `ApplyPendingBalance` are account-lifecycle operations outside the scope of the charge intent. They are NOT carried in the charge credential and **MUST NOT** appear in the bundle's transactions; the bundle is limited to proof setup, the transfer, and proof-context cleanup (Section 11.5).

10.3. Amount Verification

The server is the payment recipient, so it confirms the charged amount the same way any confidential-account holder reads an incoming payment: it decrypts the amount credited to its own confidential token account using its own ElGamal secret key. The grouped validity proof binds the receiver ciphertext to the same amount the sender debits, so this recipient-side value is the authoritative settled amount (Section 11.5). The server holds this key already — it is the key of the recipient account named by the charge — so no additional secret custody is required beyond the recipient wallet.

The mint's auditor ElGamal key (the `auditorElgamalPubkey` in the `ConfidentialTransferMint` extension), when present, is a distinct facility owned by the mint **issuer** for compliance: it lets the issuer decrypt amounts across the mint. It is NOT used for, or required by, charge verification, and the server is not expected to hold the auditor secret. Servers **MUST NOT** treat the auditor handle as their amount-verification mechanism.

10.4. Proof Context State Accounts

The client builds the bundle so that, for each required proof, a proof context state account is created and the proof is verified into it by the ZK ElGamal Proof Program [ZK-ELGAMAL-PROOF] before the transfer instruction that references it. After the transfer, the bundle **MUST** close those context state accounts and return their rent to whoever funded it — the server, under fee sponsorship (Section 14.10).

Clients **SHOULD** minimize the number of transactions by batching proof verifications subject to the transaction size limit, but **MUST NOT** exceed it. Servers **MUST** treat the bundle as opaque in count but **MUST** verify its structure per Section 11.5.

Before generating proofs (which is comparatively expensive), clients **SHOULD** pre-flight the transfer: confirm the mint has the Confidential Transfer extension, that the recipient's account is configured and allows confidential credits, and that the sender's decrypted available balance covers the amount. This turns the common failure modes into fast, local errors instead of a built bundle that fails on-chain.

10.5. Pending Balance and Delivery Semantics

A confidential transfer credits the recipient's pending balance, not its available balance. The funds are delivered but not yet spendable; only the recipient can convert them with `ApplyPendingBalance`. A successful confidential charge therefore attests that the encrypted amount was transferred to the recipient's confidential account, not that it is immediately spendable by the recipient. The receipt signals this with `delivery: "pending"` (Section 12.8).

10.6. Restrictions

- `splits` **MUST NOT** be combined with `confidential`. Each confidential split would require its own proof set and transfer; this is out of scope for `draft-00`. Servers **MUST** reject a challenge that sets both.
- Confidential charges **MUST** use `type="bundle"`. `type="signature"` and a bare `type="transaction"` **MUST NOT** be used, and servers **MUST** reject them when `confidential` is `true`.

11. Verification Procedure

Upon receiving a request with a credential, the server **MUST**:

1. Decode the base64url credential and parse the JSON.
2. Verify that `payload.type` is present and is one of `"transaction"`, `"signature"`, or `"bundle"`. If the challenge sets `confidential: true`, the type **MUST** be `"bundle"`; otherwise it **MUST** be `"transaction"` or `"signature"`.
3. Look up the stored challenge using `credential.challenge.id`. If no matching challenge is found, reject the request.
4. Verify that all fields in `credential.challenge` exactly match the stored challenge auth-params.
5. If `payload.type` is `"signature"` and the challenge specifies `feePayer: true`, reject the request (see Section 8.4).
6. Proceed with type-specific verification:
   - For `type="transaction"`: see Section 11.1.
   - For `type="signature"`: see Section 11.2.
   - For `type="bundle"`: see Section 11.5.

11.1. Pull Mode Verification

For credentials with `type="transaction"`:

1. Decode the base64 `payload.transaction` value.
2. Deserialize the transaction and verify that it structurally matches the challenge request:
   - the fee payer matches the challenge policy;
   - the transfer authority is signed by the client;
   - the transaction contains only expected transfer, ATA-creation, memo, and compute-budget instructions;
   - when `feePayer` is `true`, ATA-creation instructions funded by the server fee payer are limited to split recipients whose split entry sets `ataCreationRequired` to `true`, as described in Section 9.5;
   - when `feePayer` is `false` or omitted, ATA-creation instructions are limited to split recipients;
   - the payment semantics match the challenge request, as described in Section 11.3 or Section 11.4.
3. If `feePayer` is `true`, add the server's fee payer signature using the `feePayerKey` and re-serialize. The transaction **MUST** have the server's `feePayerKey` set as the fee payer account.
4. If `feePayer` is `true`, simulate the transaction using the `simulateTransaction` RPC method. The server **MUST** reject the credential if simulation fails. If `feePayer` is `false` or omitted, the server **SHOULD** simulate the transaction before broadcast and **SHOULD** reject the credential if simulation indicates the transaction will fail. This catches invalid transactions without spending fees, which is especially important in fee payer mode (see Section 14.6).
5. Broadcast the transaction to the Solana network using `sendTransaction`.
6. Wait for confirmation at the required commitment level.
7. Fetch the confirmed transaction using `getTransaction` with `jsonParsed` encoding and verify the transfer details still match the challenge request, as described in Section 11.3 or Section 11.4.
8. Record the transaction signature as consumed to prevent replay (see Section 11.6).
9. Return the resource with a Payment-Receipt header.

11.2. Push Mode Verification

For credentials with `type="signature"`:

1. Verify that `payload.signature` is present and is a valid base58-encoded string.
2. Verify the transaction signature has not been previously consumed (see Section 11.6).
3. Fetch the transaction from the Solana network using the RPC `getTransaction` method with `jsonParsed` encoding and the `confirmed` commitment level.
4. Verify the transaction was successful (no error in the transaction metadata).

5. Verify the transfer details match the challenge request, as described in Section 11.3 or Section 11.4.
6. Mark the transaction signature as consumed to prevent replay.
7. Return the resource with a Payment-Receipt header.

Note: both credential types reuse the same on-chain transfer verification logic defined in Section 11.3 and Section 11.4.

11.3. Native SOL Verification

For native SOL payments (`currency` is `"sol"`), the server **MUST**:

1. Compute the primary payment amount as the top-level `amount` minus the sum of all `splits`, if any.
2. Locate a System Program `transfer` instruction in the transaction's parsed instructions whose `destination` matches the top-level `recipient` and whose `lamports` field matches that primary payment amount.
3. For each split in `splits`, if any, locate an additional System Program `transfer` instruction whose `destination` and `lamports` fields match that split.

   Each required payment leg **MUST** be matched to a distinct transfer instruction. A single transfer instruction **MUST NOT** satisfy more than one required payment leg, even if multiple legs share the same recipient.

If any required transfer instruction is missing, the server **MUST** reject the credential.

11.4. SPL Token Verification

For SPL token payments (`currency` is a mint address, not `"sol"`), the server **MUST**:

1. Compute the primary payment amount as the top-level `amount` minus the sum of all `splits`, if any.
2. Locate a `transferChecked` instruction from the appropriate token program (Token Program or Token-2022) in the transaction's parsed instructions whose `mint` field matches the top-level `currency` field from the challenge request.
3. Derive the expected destination associated token account for the top-level `recipient` from the `recipient`, `currency`, and `tokenProgram` in the challenge request. Verify that at least one matching `transferChecked` instruction uses that derived ATA as `destination` and has `tokenAmount.amount` equal to the primary payment amount.
4. For each split in `splits`, if any, derive the expected destination ATA for that split recipient and verify that at least one additional `transferChecked` instruction uses that ATA as `destination` and has `tokenAmount.amount` equal to the split amount.

   Each required payment leg **MUST** be matched to a distinct `transferChecked` instruction. A single instruction **MUST NOT** satisfy more than one required payment leg, even if multiple legs resolve to the same destination ATA.

If any required `transferChecked` instruction is missing, the server **MUST** reject the credential.

Split recipient ATA creation does not alter SPL transfer verification. The selected challenge request defines the full set of required payment legs. A `transferChecked` instruction to an ATA created by the transaction satisfies a required payment leg only if that ATA owner appears in `splits` for the selected challenge.

11.5. Confidential Transfer Verification

For credentials with `type="bundle"` (confidential charges), the server **MUST**:

1. Decode each element of `payload.transactions` and deserialize it. Reject the credential if any element is not a valid transaction or exceeds the transaction size limit.
2. Verify the bundle contains only expected instructions: proof context state account creation, ZK ElGamal Proof Program [ZK-ELGAMAL-PROOF] verification instructions, the confidential `Transfer` or `TransferWithFee` instruction on the Token-2022 Program, context-account close instructions, and optionally memo and compute-budget instructions. Any other instruction — in particular `ConfigureAccount`, `ApproveAccount`, deposit, withdraw, or `ApplyPendingBalance` — **MUST** cause rejection (Section 10.2).
3. Verify the confidential transfer instruction:
   - operates on the mint identified by `currency`;
   - uses, as its destination, the recipient's confidential token account derived from the top-level `recipient` and the mint;
   - references the proof context state accounts created earlier in the bundle;
   - credits the recipient's confidential account (the receiver ciphertext is bound to the transferred amount by the grouped validity proof).
4. If `feePayer` is `true`, verify every transaction sets the server's `feePayerKey` as fee payer, that the only rent-funding instructions are `create_accounts` funded by the server and assigning the new account to the ZK ElGamal Proof Program or the record program, and that the proof/record accounts name the server as their authority and rent-reclaim destination (Section 14.10); then add the server's fee payer signature to each transaction. The server **MUST** also bound the number of transactions it will co-sign and submit per bundle.
5. If `feePayer` is `true`, simulate each transaction before broadcast and reject the credential on simulated failure. Otherwise the server **SHOULD** simulate before broadcast.
6. Submit the transactions in array order, waiting for at least the `confirmed` commitment level on each before submitting the next. If any transaction fails to land, the server **MUST** reject the credential and **MUST NOT** return a success receipt, even if earlier transactions in the bundle have landed.
7. Confirm the received amount: as the payment recipient, the server decrypts the amount credited to its own confidential token account (the increase in its pending balance) using its own recipient ElGamal secret key, and verifies it equals the top-level `amount`. If the decrypted amount does not match, the server **MUST** reject the credential. This needs no auditor key — the server already holds the recipient account's key (see Section 10.3).

8. Record the signature of the final (transfer) transaction as consumed to prevent replay (Section 11.6).
9. Return the resource with a Payment-Receipt header (Section 12.8).

Because the proofs are verified on-chain by the ZK ElGamal Proof Program, the server does not re-verify the zero-knowledge proofs itself; it relies on the proofs having been accepted by the program as a precondition for the transfer instruction succeeding. The server's independent check is the recipient-side decryption in step 7, which binds the amount actually credited to its account to the challenged `amount`.

11.6. Replay Protection

Servers **MUST** maintain a set of consumed transaction signatures. Before accepting a credential, the server **MUST** check whether the signature has already been consumed. After successful verification, the server **MUST** atomically mark the signature as consumed.

The transaction signature is globally unique on the Solana network, making it a natural replay prevention token. A signature that has been consumed **MUST NOT** be accepted again, even if presented with a different challenge ID.

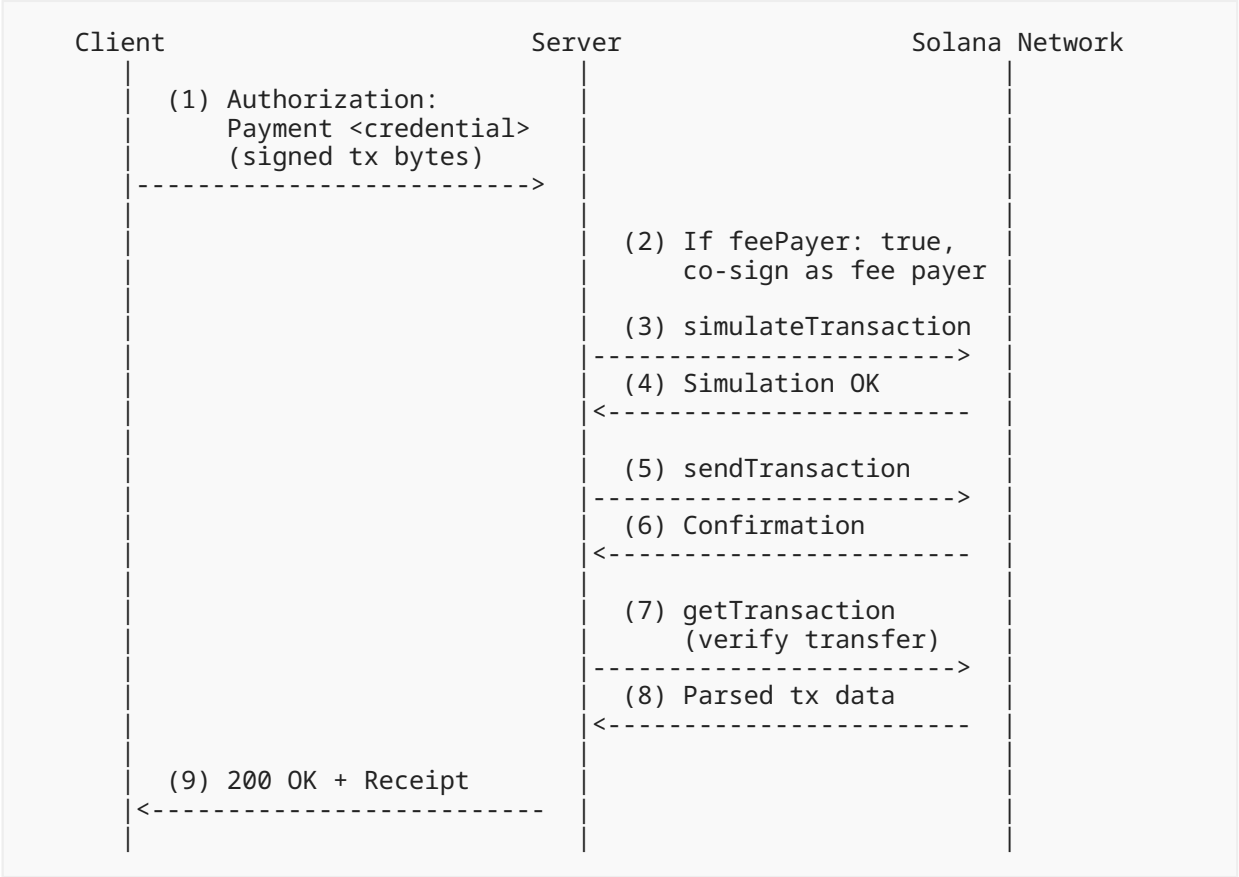
For `type="transaction"` credentials, the transaction signature is derived after broadcast. For `type="signature"` credentials, the signature is provided directly by the client.

12. Settlement Procedure

Two settlement flows are supported, corresponding to the two credential types.

12.1. Pull Mode Settlement (type="transaction")

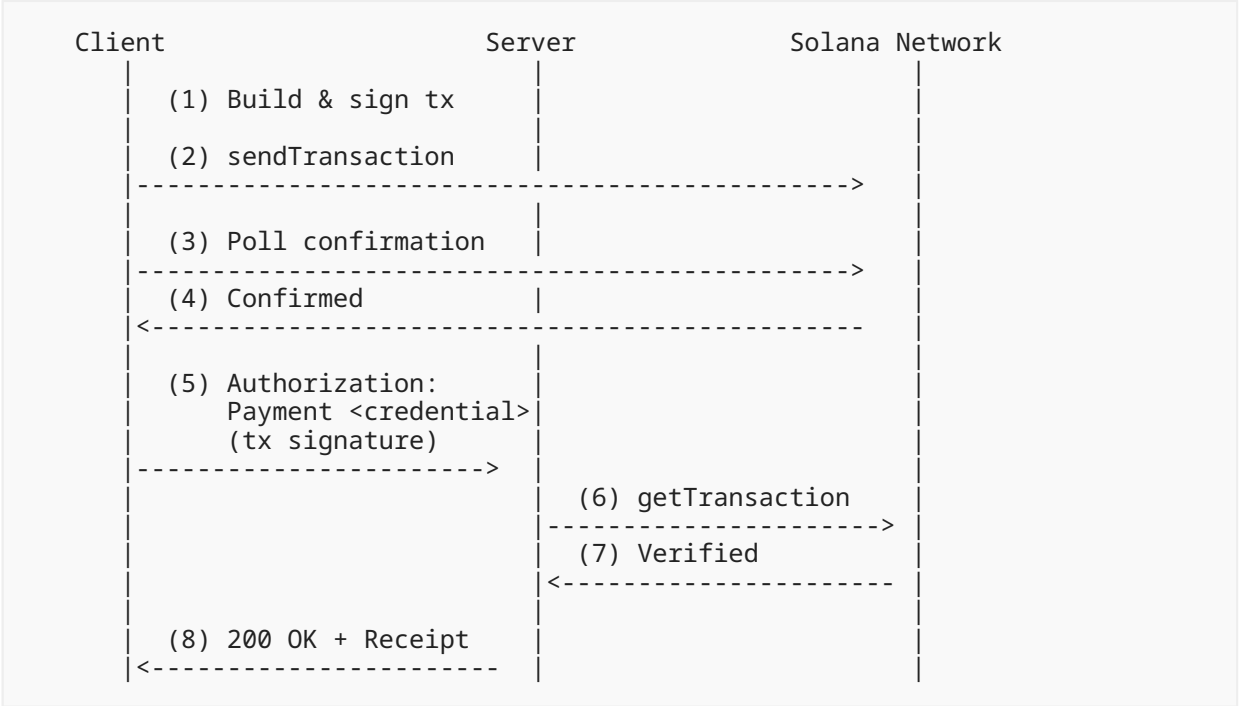
For `type="transaction"` credentials, the client signs the transaction and sends it to the server. The server optionally adds a fee payer signature and broadcasts:



1. Client submits credential containing signed transaction bytes.
2. If `feePayer` is `true`, the server co-signs with its fee payer key.
3. Server simulates the transaction to catch failures without spending fees.
4. Server broadcasts the transaction to Solana.
5. Transaction reaches the required commitment level.
6. Server fetches the confirmed transaction and verifies the transfer details match the challenge request.
7. Server records the signature as consumed and returns the resource with a Payment-Receipt header whose `reference` field is the transaction signature.

12.2. Push Mode Settlement (type="signature")

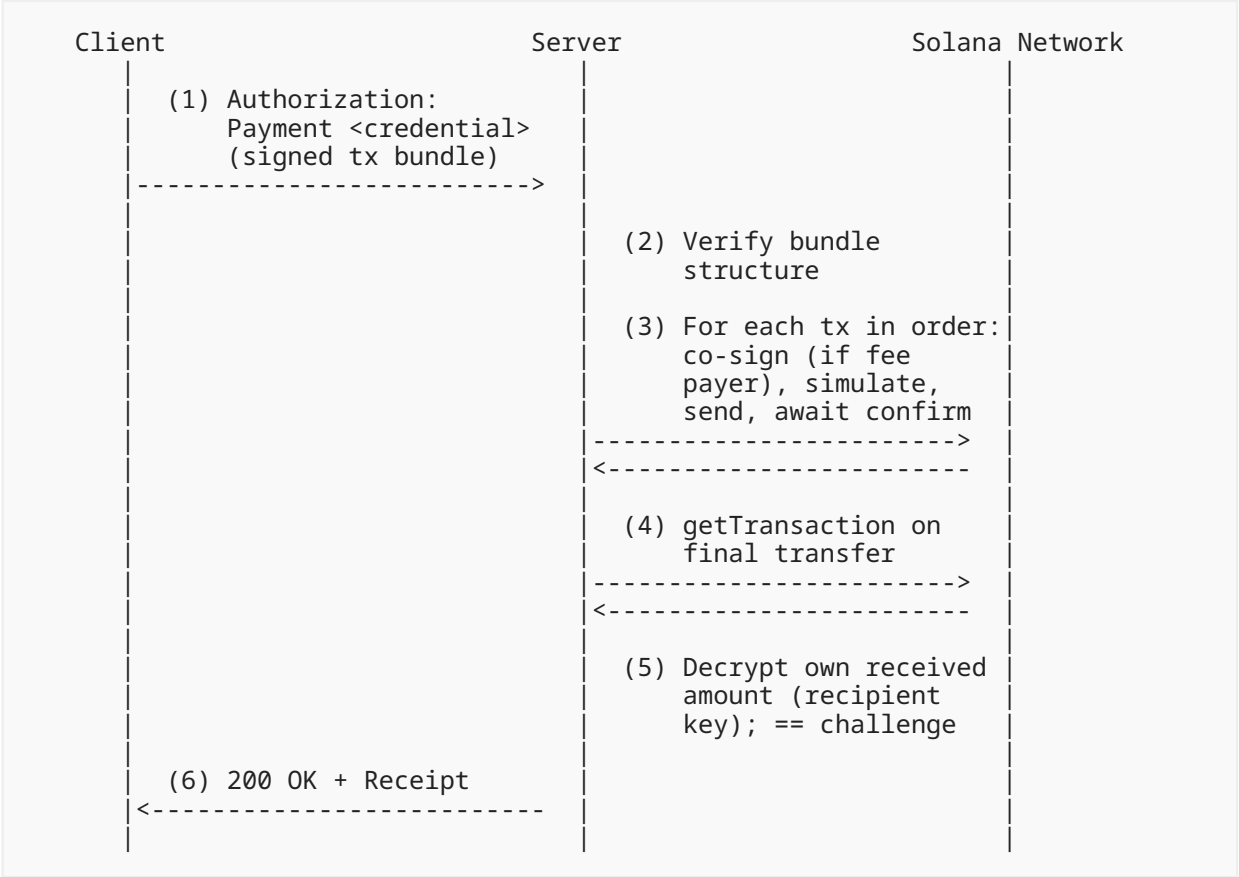
For `type="signature"` credentials, the client broadcasts the transaction itself and presents the confirmed signature:



1. Client builds a transfer transaction and signs it.
2. Client sends the transaction to the Solana network.
3. Client polls for confirmation status.
4. Transaction reaches `confirmed` commitment level.
5. Client presents the transaction signature as the credential.
6. Server fetches the transaction via RPC and verifies transfer details.
7. Server confirms the payment matches the challenge.
8. Server returns the resource with a Payment-Receipt.

12.3. Bundle Settlement (type="bundle")

For `type="bundle"` credentials (confidential charges), the client submits an ordered set of signed transactions and the server settles them sequentially:



1. Client submits the credential containing the ordered transaction bundle.
2. Server verifies the bundle structure (Section 11.5).
3. Server co-signs (when fee payer), simulates, broadcasts, and confirms each transaction in array order. If any transaction fails to land, settlement aborts and no success receipt is returned.
4. Server fetches the confirmed final transfer transaction.
5. As the recipient, the server decrypts the amount credited to its own confidential account with its own ElGamal key and verifies it equals the challenge `amount`.
6. Server records the final transfer signature as consumed and returns the resource with a Payment-Receipt header whose `reference` is that signature and whose `delivery` is `"pending"` (Section 12.8).

12.4. Client Transaction Construction

12.4.1. Native SOL

The client **MUST** construct a transaction containing a System Program `transfer` instruction with:

- `source`: the client's signing account
- `destination`: the `recipient` from the challenge

- `lamports`: the `amount` from the challenge

12.4.2. SPL Tokens

The client **MUST** construct a transaction containing:

1. Zero or more idempotent Associated Token Account creation instructions permitted by Section 9.5:
   - When `feePayer` is `true`, the client **MUST** include an idempotent ATA creation instruction for each split recipient whose split entry sets `ataCreationRequired` to `true`, and **MUST NOT** include ATA creation instructions for the top-level `recipient`, unmarked split recipients, or arbitrary owners.
   - When `feePayer` is `false` or omitted, the client **MAY** include idempotent ATA creation instructions only for split recipients, and **MUST** include one for each split whose entry sets `ataCreationRequired` to `true`. The client **MUST NOT** include ATA creation instructions for the top-level `recipient`.

   The transaction fee payer covers the rent-exempt minimum (~0.002 SOL) if the account does not exist.
2. A `transferChecked` instruction on the appropriate token program for the primary payment, and one additional `transferChecked` instruction for each split. Each transfer uses:
   - `source`: the client's associated token account
   - `mint`: the `currency` field
   - `destination`: the payment recipient's derived ATA
   - `authority`: the client's signing account
   - `amount`: the primary remainder or split amount
   - `decimals`: the `decimals` from `methodDetails`

12.4.3. Fee Payer Configuration

When `feePayer` is `true` in the challenge:

- The client **MUST** set the server's `feePayerKey` as the transaction fee payer.
- The client **MUST** sign the transaction only with its own key (transfer authority).
- The fee payer signature slot **MUST** be left empty for the server to fill.

When `feePayer` is `false` or absent:

- The client **MUST** set itself as the transaction fee payer.
- The client **MUST** fully sign the transaction.

Clients **SHOULD** set a compute unit limit and priority fee appropriate for current network conditions.

12.5. Confirmation Requirements

For `type="signature"` credentials, clients **MUST** wait for at least the `confirmed` commitment level before presenting the credential. Servers **MUST** fetch the transaction with at least `confirmed` commitment. Servers **MAY** require `finalized` commitment for high-value transactions.

For `type="transaction"` credentials, the server controls the broadcast and confirmation process. Servers **MUST** wait for at least `confirmed` commitment before returning the receipt.

12.6. Finality

Solana provides two commitment levels relevant to payment verification:

- `confirmed`: optimistic confirmation from a supermajority of validators (~400ms). Sufficient for most payment use cases.
- `finalized`: deterministic finality after ~31 slots (~12 seconds). Required for high-value transactions where rollback risk is unacceptable.

In theory, a `confirmed` transaction could be rolled back if validators shift consensus to a competing fork that excludes the confirmed block. In practice, this has never occurred on Solana mainnet. The `confirmed` level is **RECOMMENDED** as the default for payment verification to minimize latency.

12.7. Receipt Generation

Upon successful verification, the server **MUST** include a `Payment-Receipt` header in the 200 response.

The receipt payload for Solana charge:

Field	Type	Description
`method`	string	`"solana"`
`challengeId`	string	The challenge `id` from `WWW-Authenticate`
`reference`	string	The transaction signature (base58-encoded)
`status`	string	`"success"`
`timestamp`	string	[RFC3339] verification time

Table 4

Example (decoded):

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny...",
  "status": "success",
  "timestamp": "2026-03-10T21:00:00Z"
}
```

12.8. Confidential Charge Receipt

For a confidential charge (`type="bundle"`), the `reference` is the signature of the final confidential transfer transaction, and the receipt includes one additional field:

Field	Type	Description
`delivery`	string	`"pending"` — the transferred amount was credited to the recipient's pending balance and becomes spendable only after the recipient applies it (see Section 10).

Table 5

The receipt **MUST NOT** include the cleartext transferred amount; the amount is encrypted on-chain, and the server learns it only by decrypting the amount credited to its own recipient account for verification purposes.

Example (decoded):

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny...",
  "status": "success",
  "delivery": "pending",
  "timestamp": "2026-03-10T21:00:00Z"
}
```

13. Error Responses

When rejecting a credential, the server **MUST** return HTTP 402 (Payment Required) with a fresh `WWW-Authenticate: Payment` challenge per [I-D.httpauth-payment]. The server **SHOULD** include a response body conforming to RFC 9457 [RFC9457] Problem Details, with `Content-Type: application/problem+json`. Servers **MUST** use the standard problem types defined in [I-D.httpauth-payment]: `malformed-credential`, `invalid-challenge`, and `verification-failed`. The `detail` field **SHOULD** contain a human-readable description of the specific failure (e.g., "Transaction not found", "Amount mismatch", "Signature already consumed").

All error responses **MUST** include a fresh challenge in `WWW-Authenticate`.

Example error response body:

```
{
  "type": "https://paymentauth.org/problems/verification-failed",
  "title": "Transfer Mismatch",
  "status": 402,
  "detail": "Destination token account does not belong to expected recipient"
}
```

14. Security Considerations

14.1. Transport Security

All communication **MUST** use TLS 1.2 or higher. Solana credentials **MUST** only be transmitted over HTTPS connections.

14.2. Replay Protection Considerations

Servers **MUST** track consumed transaction signatures and reject any signature that has already been accepted. The check-and-consume operation **MUST** be atomic to prevent race conditions where concurrent requests present the same signature. Transaction signatures are globally unique on the Solana network (derived from the signer's key and the blockhash), making them natural replay prevention tokens.

14.3. Client-Side Verification

Clients **MUST** verify the challenge before signing:

1. `amount` is reasonable for the service
2. `currency` matches the expected asset
3. `recipient` is the expected party
4. If `currency` is a mint address, verify it is a known token
5. `splits`, if present, contain expected recipients and amounts — malicious servers could add splits to redirect funds
6. `feePayerKey`, if present, is the expected server
7. If `confidential` is `true`, the recipient has a configured (and, where required, approved) confidential token account, and `auditorElgamalPubkey`, if present, matches the mint's on-chain `ConfidentialTransferMint` auditor key

Malicious servers could request excessive amounts, direct payments to unexpected recipients, or add hidden splits.

14.4. RPC Trust

The server relies on its Solana RPC endpoint to provide accurate transaction data for on-chain verification. A compromised RPC could return fabricated transaction data, causing the server to accept payments that were never made. Servers **SHOULD** use trusted RPC providers or run their own nodes.

14.5. Front-running (Push Mode)

In push mode, the client broadcasts the transaction before presenting the credential, making it visible on-chain. A party monitoring the chain could attempt to present the same signature to the server. The challenge binding (the credential echoes the challenge `id`, which is HMAC-verified) and single-use signature enforcement mitigate this: only the party that received the challenge can construct a valid credential.

Push mode does not require the on-chain transaction to carry a challenge-specific marker. It proves that a payment matching the challenged terms was made, but not necessarily that the payment was created for one unique challenge instance. If multiple valid challenges have identical terms, the same confirmed transaction could satisfy any one of them, and the first accepted presentation wins.

Requiring an on-chain marker such as a Memo carrying the challenge `id` would provide stronger binding, but would also reveal extra correlation metadata on chain. This specification does not require such a marker in the base flow, but implementations **MAY** define a backward-compatible profile that does.

Pull mode is not susceptible to front-running because the transaction is not broadcast until the server receives and validates the credential.

14.6. Fee Payer Risks

Servers acting as fee payers accept financial risk in exchange for providing a seamless payment experience.

Denial of Service via Bad Transactions Malicious clients could submit transactions that fail on-chain (insufficient balance, invalid instructions), causing the server to pay ~5,000 lamports per failed transaction. Mitigations:

- **Transaction simulation**: `simulateTransaction` catches most failures before broadcast, without spending fees. Servers **MUST** simulate fee-sponsored pull mode transactions before broadcasting. Servers **SHOULD** simulate non-fee-sponsored pull mode transactions before broadcasting.
- **Rate limiting**: per client address, per IP, or per time window.
- **Balance verification**: check the client's balance covers the transfer amount before signing.

- **Client authentication**: require API keys or OAuth tokens before accepting fee-sponsored transactions.

ATA Rent Drain When the fee payer funds creation of an Associated Token Account (ATA), it pays ~0.002 SOL in rent. The recipient can close the ATA to reclaim rent, then the next payment re-creates it at the fee payer's expense. Servers **SHOULD** verify the top-level recipient's ATA exists before issuing a challenge, because top-level recipient ATA creation is not allowed by this specification. For split recipients, servers that set `ataCreationRequired: true` are explicitly accepting rent risk for those split recipient ATAs and **SHOULD** apply stricter rate limits, authentication, and cost-recovery policy.

Fee Payer Balance Exhaustion Servers **MUST** monitor fee payer balance and reject new fee-sponsored requests when insufficient. The server **SHOULD** return a 402 with `feePayer: false`, allowing the client to pay its own fees as fallback.

14.7. Transaction Payload Security

In pull mode, the server receives raw transaction bytes from the client. A malicious client could craft a transaction that transfers funds FROM the server's fee payer account rather than simply paying fees.

Servers **MUST** verify that the transaction contains only the expected instructions: transfer instruction(s) matching the challenge parameters, ATA creation (idempotent), and optionally compute budget instructions. Any unexpected instructions **MUST** cause rejection.

14.8. Blockhash Freshness

When the server provides `recentBlockhash` in the challenge, clients **SHOULD** verify it is plausible (not obviously stale). A malicious server could provide an expired blockhash, causing the client to sign a transaction that will never land — wasting the signing effort. However, since the transaction is not broadcast by the client in pull mode, the practical risk is limited to a failed payment attempt that the client can retry.

14.9. Recipient Key (Confidential)

Confidential charge verification uses the server's **recipient** ElGamal key — the key of the confidential token account named as the payee. The server derives this key from the recipient wallet it already controls, so verification introduces no new high-value secret beyond that wallet. Servers **SHOULD** derive it on demand rather than persist it separately. Compromise of the recipient key reveals the amounts that account received (the same exposure as losing the payee wallet); it does not by itself permit theft.

The mint's **auditor** key, if the mint configures one, is a separate facility held by the mint **issuer** for compliance — it is not held by the server and plays no role in charge verification. Conflating the auditor with the payee would couple the payment provider to the stablecoin issuer; this specification deliberately keeps them distinct.

14.10. Proof Context Rent Drain (Confidential)

Proof context state accounts (and the range-proof record account) are rent-bearing. The paying client typically holds no SOL, so it cannot fund this rent — the server (fee payer) does. A naive design where the server funds rent but the client controls those accounts would let a malicious client withhold the closing transaction and strand the server's rent, analogous to the ATA rent drain in Section 14.6. This profile avoids that by making the server the **authority** of every proof/record account it funds: the bundle **MUST** create these accounts with the server's `feePayerKey` as funder and as `context_state_authority` (and record authority), and **MUST** close them back to the server within the bundle, so that net rent is approximately zero on success.

The server protects itself two ways: (1) before co-signing, it verifies each transaction contains only allow-listed instructions and that every `create_account` is server-funded and assigns the new account to the ZK ElGamal Proof Program or the record program (Section 11.5); and (2) because it is the authority of any account a partial failure leaves open, it can reclaim that rent itself (see below). Servers **MUST** reject bundles whose `create_account` instructions name a funder other than the server or assign the account to any other program.

14.11. Partial Bundle Settlement (Confidential)

A confidential transfer is settled across multiple transactions. If settlement aborts after some transactions have landed, on-chain state may include created proof/record accounts without a completed transfer. The server **MUST NOT** return a success receipt in this case. Because the server funded the rent and is the authority of those accounts, the stranded rent is the server's to reclaim: servers **SHOULD** run a periodic sweep that closes proof/record accounts they own that were left open by aborted bundles, returning the rent to themselves. To avoid closing an account that belongs to a settlement still in flight (which creates and closes its accounts within one bounded settlement), the sweep **SHOULD** only close an account observed across two successive sweeps separated by more than the settlement window. Servers **SHOULD** also bound the time and number of transactions they will settle for a single credential to limit resource exhaustion.

14.12. Amount Privacy Scope (Confidential)

The confidential profile hides the transferred amount from public on-chain observers. It does not hide the amount from the challenge exchange itself: the `amount` field travels in the challenge and credential over TLS, and the server learns the amount both from the challenge and by decrypting the amount credited to its own recipient account. Confidentiality is therefore relative to third-party chain observers, not to the counterparties (nor any auditor the mint issuer configures). Sender and recipient account identities, and the fact that a confidential transfer occurred, remain visible on-chain.

Metadata beyond the amount also stays visible — in particular, any memo. To avoid re-linking a confidential payment to a specific order, this profile does NOT write an order/`externalId` memo on-chain by default; servers **SHOULD** reconcile a confidential settlement by the globally-unique signature of its final (transfer) transaction, returned on the receipt, rather than an on-chain

marker. A server that does include a memo **MUST** treat the resulting order-id linkage as an explicit privacy trade-off, and **MUST** verify any memo present in the bundle matches the value it issued in the challenge.

15. IANA Considerations

15.1. Payment Method Registration

This document requests registration of the following entry in the "HTTP Payment Methods" registry established by [I-D.httpauth-payment]:

Method Identifier	Description	Reference
`solana`	Solana blockchain native SOL and SPL token transfer	This document

Table 6

15.2. Payment Intent Registration

This document requests registration of the following entry in the "HTTP Payment Intents" registry established by [I-D.httpauth-payment]:

Intent	Applicable Methods	Description	Reference
`charge`	`solana`	One-time SOL or SPL token transfer	This document

Table 7

16. References

16.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/info/rfc9457>.

[I-D.payment-intent-charge] Moxey, J., Ryan, B., and T. Meagher, "'charge' Intent for HTTP Payment Authentication", 2026, <https://datatracker.ietf.org/doc/draft-payment-intent-charge/>.

[I-D.httpauth-payment] Moxey, J., "The 'Payment' HTTP Authentication Scheme", January 2026, <https://datatracker.ietf.org/doc/draft-ryan-httpauth-payment/>.

16.2. Informative References

[SOLANA-DOCS] Solana Foundation, "Solana Documentation", 2026, <https://solana.com/docs>.

[SPL-TOKEN] Solana Foundation, "SPL Token Program", 2026, <https://solana.com/docs/tokens>.

[SPL-TOKEN-2022] Solana Foundation, "SPL Token-2022 Program", 2026, <https://solana.com/docs/tokens/extensions>.

[CONFIDENTIAL-TRANSFER] Solana Foundation, "Token-2022 Confidential Transfer Extension", 2026, <https://www.solana-program.com/docs/confidential-balances>.

[ZK-ELGAMAL-PROOF] Anza, "ZK ElGamal Proof Program", 2026, <https://docs.anza.xyz/runtime/zk-elgamal-proof>.

[BASE58] Sporny, M., "Base58 Encoding Scheme", 2023, <https://datatracker.ietf.org/doc/html/draft-msporny-base58-03>.

Appendix A. Examples

The following examples illustrate the complete HTTP exchange for each flow. Base64url values are shown with their decoded JSON below.

A.1. Native SOL Charge (Pull Mode)

A 0.01 SOL charge for weather API access.

1. Challenge (402 response):

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="kM9xPqWvT2nJrHsY4aDfEb",
  realm="api.example.com",
  method="solana",
  intent="charge",
  request="eyJhbW91bnQiOiIxMDAwMDAwMCIsImN1cnJlbmN5Ij
    oiU09MIiwiZGVzY3JpcHRpb24iOiJXZWF0aGVyIEFQSSBhY2
    Nlc3MiLCJtZXRob2REZXRhaWxzIjp7Im5ldHdvcmsiOiJtYWl
    ubmV0LWJldGEiLCJyZWZlcmVuY2UiOiJmNDdhYzEwYi01OGNj
    LTQzNzItYTU2Ny0wZTAyYjJjM2Q0NzkifSwicmVjaXBpZW50I
    joiN3hLWHRnMkNXODdkOTdUWEpTRHBiRDVqQmtoZVRxQTgzVF
    pSdUpvc2dBc1UifQ",
  expires="2026-03-15T12:05:00Z"
Cache-Control: no-store
```

Decoded `request`:

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet"
  }
}
```

2. Credential (retry with signed transaction):

```
GET /weather HTTP/1.1
Host: api.example.com
Authorization: Payment <base64url-encoded credential>
```

Decoded credential:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "<base64url-encoded request>",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "<base64-encoded signed transaction>"
  }
}
```

3. Response (with receipt):

```
HTTP/1.1 200 OK
Payment-Receipt: <base64url-encoded receipt>
Content-Type: application/json

{"temperature": 72, "condition": "sunny"}
```

Decoded receipt:

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "4vJ9YFuPzUgdLkWYJf3KqfNM8cTnBp3jXx...",
  "status": "success",
  "timestamp": "2026-03-15T12:04:58Z"
}
```

A.2. SPL Token (USDC) Charge with Fee Sponsorship

A 1 USDC charge where the server sponsors transaction fees and includes a `recentBlockhash` to eliminate client RPC dependency.

Decoded `request`:

```
{
  "amount": "1000000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Premium API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA",
    "feePayer": true,
    "feePayerKey": "Gh9ZwEmdLJ8DscKNTkTqPbNwLNNBjuSzaG9Vp2KGtKJr",
    "recentBlockhash": "EkSnNWid2cvwEVnVx9aBqawnmiCNiDgp3gUdkDPTKN1N"
  }
}
```

The client uses `recentBlockhash` from the challenge (no RPC call needed), sets `feePayerKey` as the transaction fee payer, and partially signs with its own key only. The server verifies the transaction contents, co-signs as fee payer, and broadcasts.

Decoded credential:

```
{
  "challenge": { "..." : "echoed challenge" },
  "payload": {
    "type": "transaction",
    "transaction": "<base64-encoded partially-signed tx>"
  }
}
```

A.3. Push Mode (type="signature")

The client broadcasts the transaction itself and presents the confirmed signature. Cannot be used with fee sponsorship.

Decoded credential:

```
{
  "challenge": { "..." : "echoed challenge" },
  "payload": {
    "type": "signature",
    "signature": "4vJ9YFuPzUgdLkWYJf3KqfNM8cTnBp3jXx..."
  }
}
```

A.4. Payment Splits

A marketplace charge of 1.05 USDC where 0.05 USDC goes to the platform as a fee.

Decoded `request`:

```
{
  "amount": "1050000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Marketplace purchase",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "splits": [
      {
        "recipient": "3pF8Kg2aHbNvJkLMwEqR7YtDxZ5sGhJn4UV6mWcXrT9A",
        "amount": "50000",
        "memo": "platform fee"
      }
    ]
  }
}
```

The client builds a transaction with two transfers: 1,000,000 base units to the primary recipient and 50,000 to the platform. The total paid remains 1,050,000 base units, matching the top-level `amount`.

A.5. Confidential Charge (Bundle)

A 1-token confidential charge with server-sponsored fees. The amount is encrypted on-chain; the server, as recipient, verifies it by decrypting the amount credited to its own account.

Decoded `request`:

```
{
  "amount": "1000000",
  "currency": "HVWf8JmLoHs99Lw8Psf3fyqAtA4crWxCPkrmSdNjhNH3",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Confidential API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb",
    "feePayer": true,
    "feePayerKey": "9aE3Fg7HjKLmNpQr5TuVwXyZ2AbCdEf8GhIjKlMnOp1R",
    "confidential": true
  }
}
```

The client confirms the recipient has an approved confidential token account, builds the proof-context-setup and confidential-transfer transactions (each with `feePayerKey` as fee payer and signed only by the transfer authority), and presents them as a bundle.

Decoded credential:

```
{
  "challenge": { "..." : "echoed challenge" },
  "payload": {
    "type": "bundle",
    "transactions": [
      "<base64-encoded proof-context-setup tx>",
      "<base64-encoded confidential-transfer tx>"
    ]
  }
}
```

Decoded receipt:

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny...",
  "status": "success",
  "delivery": "pending",
  "timestamp": "2026-03-15T12:04:58Z"
}
```

Appendix B. Acknowledgements

The authors thank the Tempo team for their input on this specification.

Authors' Addresses

Ludo Galabru
Solana Foundation
Email: ludo.galabru@solana.org

Ilan Gitter
Solana Foundation
Email: ilan.gitter@solana.org