
Workgroup: Network Working Group
Internet-Draft: draft-evm-charge-00
Published: 18 May 2026
Intended Status: Informational
Expires: 19 November 2026
Authors: B. DiNovi C. Swenberg K. Scott
MegaETH Labs Coinbase Monad Foundation

EVM Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "evm" payment method in the Payment HTTP Authentication Scheme [I-D.httpauth-payment]. It specifies how clients and servers exchange one-time ERC-20 token transfers on any EVM-compatible blockchain.

Four credential types are supported: `type="permit2"` (**RECOMMENDED**), where the client signs an off-chain Permit2 authorization and the server submits the transfer; `type="authorization"` (opt-in for EIP-3009 tokens), where the client signs an off-chain transfer authorization and the server submits it directly to the token contract; `type="transaction"`, where the client signs and the server broadcasts a standard ERC-20 transfer transaction; and `type="hash"` (optional fallback), where the client broadcasts the transaction itself and presents the on-chain transaction hash for server verification.

This specification covers ERC-20 token transfers only. Native token transfers (ETH, etc.) are out of scope.

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 19 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	4
1.1. Design Rationale	4
1.2. Client-Broadcast Fallback	5
1.3. Credential Types	5
1.4. Charge Flow	5
1.5. Relationship to the Charge Intent	6
2. Requirements Language	6
3. Terminology	6
4. Request Schema	7
4.1. Shared Fields	7
4.2. Method Details	7
4.2.1. Chain Identification	8
4.2.2. Credential Type Negotiation	8
4.2.3. Payment Splits	8
5. Credential Schema	9
5.1. Credential Structure	9
5.2. Permit2 Payload (type="permit2")	10
5.2.1. Permit Object	10
5.2.2. Transfer Details	11
5.2.3. Witness Data (Challenge Binding)	11
5.2.4. Server Behavior	12
5.2.5. Example: Single Transfer	13

5.2.6. Example: Batch Transfer with Splits	14
5.3. Authorization Payload (type="authorization")	15
5.3.1. Challenge Binding	15
5.3.2. Constraints	16
5.3.3. Example	16
5.4. Transaction Payload (type="transaction")	16
5.5. Hash Payload (type="hash")	17
5.6. Gas Sponsorship Model	18
6. Verification Procedure	18
6.1. Permit2 Verification	19
6.2. Authorization Verification	20
6.3. Transaction Verification	20
6.4. Hash Verification	20
7. Settlement Procedure	21
7.1. Permit2 Settlement	21
7.2. Authorization Settlement	22
7.3. Transaction Settlement	22
7.4. Hash Settlement	23
7.5. Confirmation Requirements	23
7.6. Receipt Generation	23
8. Replay Protection	24
9. Error Responses	24
10. Security Considerations	25
10.1. Transport Security	25
10.2. Transaction Replay	25
10.3. Amount Verification	25
10.4. Hash Credential Binding	25
10.5. Permit2-Specific Risks	26
10.6. Fee Payer Risks	26
10.7. Split Payment Risks	26

10.8. RPC Trust	27
11. IANA Considerations	27
11.1. Payment Method Registration	27
11.2. Payment Intent Registration	27
12. References	27
12.1. Normative References	27
12.2. Informative References	28
Appendix A. Full Example: Permit2 Charge on MegaETH	28
Appendix B. Full Example: Transaction Charge on Sei	31
Appendix C. Acknowledgements	31
Authors' Addresses	32

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 "evm" payment method.

The Ethereum Virtual Machine (EVM) is the execution environment shared by Ethereum and a growing number of compatible blockchains. These chains share a common smart contract interface (ERC-20 [[ERC-20](#)]), transaction format (EIP-1559 [[EIP-1559](#)]), address encoding (EIP-55 [[EIP-55](#)]), and JSON-RPC API — making it possible to define a single payment method that works across all of them.

This specification inherits the shared request semantics of the "charge" intent from [[I-D.payment-intent-charge](#)]. It defines only the EVM-specific `methodDetails`, `payload`, and verification procedures.

1.1. Design Rationale

Prior drafts proposed separate payment methods for individual EVM chains. However, the control flow, data structures, and verification logic are identical across these chains — the only differences are chain ID and optional RPC extensions. A unified `evm` method avoids fragmenting the registry while still allowing chain-specific optimizations at the implementation level.

1.2. Client-Broadcast Fallback

Some clients (custodial wallets, hardware signers) cannot hand off a signed-but-unbroadcast transaction. For these cases, `type="hash"` allows the client to broadcast the transaction itself and present the on-chain hash. This mode provides weaker challenge binding and does not support splits or server-paid fees. Servers **SHOULD** prefer `type="permit2"` or `type="transaction"` when possible.

1.3. Credential Types

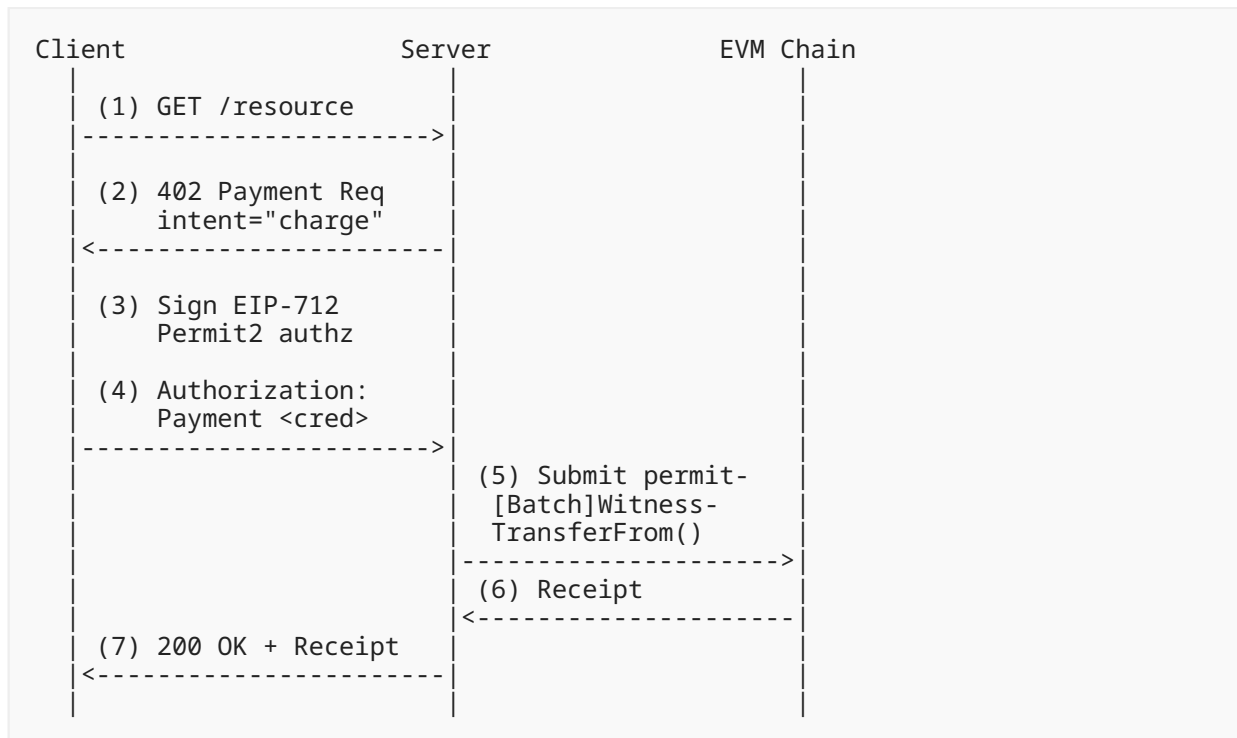
This specification defines four credential types:

- **`type="permit2"` (RECOMMENDED)**: The client signs an off-chain EIP-712 Permit2 authorization. The server constructs and submits the on-chain transaction. This is the preferred flow because:
 - The client never interacts with the chain directly
 - The server naturally sponsors gas (fee payer)
 - Split payments are atomic via batch transfers
 - No nonce management burden on the client
 - `externalId` is cryptographically bound via witness data
- **`type="authorization"`**: The client signs an off-chain EIP-3009 [EIP-3009] `transferWithAuthorization` message. The server submits it to the token contract. This credential is available for tokens that natively implement EIP-3009 (e.g., USDC, EURC). Benefits:
 - No Permit2 approval prerequisite — zero setup
 - Server naturally sponsors gas
 - Challenge binding via the on-chain nonce
 - Simpler signature structure
- **`type="transaction"`**: The client signs a complete ERC-20 `transfer` transaction. The server broadcasts it. This is the compatible fallback for chains where Permit2 is not deployed or clients that prefer direct transaction signing.
- **`type="hash"`**: The client signs and broadcasts a standard ERC-20 `transfer` transaction and presents the confirmed transaction hash for server verification. This is the fallback for some hardware wallets that cannot hand off a signed-but-unbroadcast transaction.

Servers that support Permit2 **SHOULD** advertise it as the preferred credential type. Clients **SHOULD** prefer `type="permit2"` when available.

1.4. Charge Flow

The following diagram illustrates the recommended charge flow using a Permit2 credential:



1.5. 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 EVM-specific methodDetails, payload, and verification procedures for the "evm" 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

ERC-20 The standard token interface on EVM-compatible chains [ERC-20]. Tokens expose `transfer(address,uint256)` and emit `Transfer` events on successful transfers.

Permit2 Uniswap's universal token approval contract [PERMIT2], deployed at the canonical address `0x00000000000022D473030F116dDEE9F6B43aC78BA3` on all supported chains. Enables off-chain signed approvals for any ERC-20 token via `PermitWitnessTransferFrom` and `PermitBatchWitnessTransferFrom`.

EIP-712 A standard for typed structured data hashing and signing [EIP-712], used by Permit2 to produce human-readable, replay-protected authorization signatures.

Base Units The smallest transferable unit of a token, determined by the token's decimal precision. For example, USDC (6 decimals) uses 1,000,000 base units per 1 USDC; USDm (18 decimals) uses 10^{18} base units per 1 USDm.

4. Request Schema

The request parameter in the WWW-Authenticate challenge contains a base64url-encoded JSON object. The JSON **MUST** be serialized using JSON Canonicalization Scheme (JCS) [RFC8785] before base64url encoding, per [I-D.httpauth-payment].

4.1. Shared Fields

Field	Type	Required	Description
amount	string	REQUIRED	Amount in base units (stringified integer)
currency	string	REQUIRED	ERC-20 token contract address
recipient	string	REQUIRED	Recipient address, EIP-55 encoded [EIP-55]
description	string	OPTIONAL	Human-readable payment description
externalId	string	OPTIONAL	Merchant's reference (order ID, invoice number, etc.)

Table 1

Challenge expiry is conveyed by the expires auth-param in WWW-Authenticate per [I-D.httpauth-payment].

Addresses in currency and recipient **MUST** be 0x-prefixed, 20-byte hex strings. Implementations **SHOULD** use EIP-55 mixed-case encoding but **MUST** compare addresses by decoded 20-byte value, not string form.

4.2. Method Details

Field	Type	Required	Description
chainId	number	REQUIRED	EIP-155 chain ID
permit2Address	string	REQUIRED	Permit2 contract address (default: canonical address)
credentialTypes	array	OPTIONAL	Ordered list of accepted credential types

Field	Type	Required	Description
decimals	number	OPTIONAL	Token decimal precision (e.g., 6 for USDC, 18 for USDm). Aids client-side display verification.
splits	array	OPTIONAL	Additional payment splits (max 10)

Table 2

4.2.1. Chain Identification

The `chainId` field is **REQUIRED** and identifies the target blockchain using its EIP-155 chain ID. Clients **MUST** reject challenges whose `chainId` does not match a chain they support.

A registry of EVM chain IDs is maintained at <https://chainlist.org>. This specification is not limited to any particular set of chains.

4.2.2. Credential Type Negotiation

Servers **MAY** indicate accepted credential types via the `credentialTypes` field in `methodDetails`:

Valid values: "permit2", "authorization", "transaction", "hash".

If omitted, servers **MUST** accept "transaction" and **SHOULD** accept "hash". Servers that support Permit2 **SHOULD** include "permit2" as the first entry to indicate preference. Servers **MUST** only include "authorization" when the currency token is known to implement EIP-3009. Clients **SHOULD** use the first type in the list that they support.

4.2.3. Payment Splits

The `splits` field enables a single charge to distribute payment across multiple recipients. This is useful for platform fees, revenue sharing, and marketplace payouts.

Splits **REQUIRE** `type="permit2"` credentials. The Permit2 batch transfer mechanism ensures all transfers (primary + splits) execute atomically in a single on-chain transaction. Servers **MUST** reject split requests fulfilled with `type="transaction"`, `type="authorization"`, or `type="hash"` credentials.

Each entry in the `splits` array is a JSON object:

Field	Type	Required	Description
recipient	string	REQUIRED	EIP-55 address of split recipient
amount	string	REQUIRED	Amount in base units for this recipient
memo	string	OPTIONAL	Human-readable label (max 256 chars)

Table 3

The top-level amount represents the total the client pays. The primary recipient receives the remainder: amount minus the sum of all split amounts.

Constraints:

- The sum of `splits[] . amount` **MUST** be strictly less than amount. Clients **MUST** reject any request that violates this constraint.
- If present, `splits` **MUST** contain at least 1 entry. Servers **SHOULD** limit splits to 10 entries.
- All transfers **MUST** target the same currency token.
- Address fields are compared by decoded 20-byte value, not by string form.

The order of entries in `splits` is significant for verification. Clients **MUST** emit transfers in array order. Servers **MUST** verify `transferDetails[0]` as the primary transfer and `transferDetails[i+1]` as `splits[i]`.

Example:

```
{
  "amount": "1050000",
  "currency": "0xe15fc38f6d8c56af07bbcbe3baf5708a2bf42392",
  "recipient": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
  "description": "Marketplace purchase",
  "methodDetails": {
    "chainId": 1329,
    "credentialTypes": ["permit2"],
    "splits": [
      {
        "recipient": "0x8Ba1f109551bD432803012645Ac136ddd64DBA72",
        "amount": "50000",
        "memo": "platform fee"
      }
    ]
  }
}
```

This requests a total payment of 1.05 USDC. The platform receives 0.05 USDC and the primary recipient receives 1.00 USDC. Both transfers execute atomically via Permit2 batch.

5. Credential Schema

The credential in the Authorization header contains a base64url-encoded JSON object per [I-D.httpauth-payment].

5.1. Credential Structure

Field	Type	Required	Description
challenge	object	REQUIRED	Echo of the challenge from the server

Field	Type	Required	Description
payload	object	REQUIRED	EVM-specific payload
source	string	OPTIONAL	Payer identifier as a DID

Table 4

The source field, if present, **SHOULD** use the `did:pkh` method with the chain ID from the challenge and the payer's address (e.g., `did:pkh:eip155:4326:0x1234...`).

5.2. Permit2 Payload (type="permit2")

The **RECOMMENDED** credential type. The client signs an off-chain EIP-712 Permit2 authorization message. The server constructs and submits the on-chain transaction, paying gas from its own balance. The client never interacts with the chain directly.

This type requires that the Permit2 contract is deployed on the target chain and that the client has an active ERC-20 approval to the Permit2 contract (a one-time operation per token per chain).

Field	Type	Required	Description
type	string	REQUIRED	"permit2"
permit	object	REQUIRED	Permit2 permit data
transferDetails	array	REQUIRED	Array of transfer details
witness	object	REQUIRED	Challenge binding witness data
signature	string	REQUIRED	EIP-712 signature (0x-prefixed)

Table 5

5.2.1. Permit Object

The permit object describes the token permissions:

Field	Type	Description
permitted	array	Array of { token, amount } objects. One entry per transfer (primary + each split).
nonce	string	Permit2 nonce (stringified integer)
deadline	string	Unix timestamp (stringified integer)

Table 6

The permitted array **MUST** always be an array, even for single transfers (length 1). Each entry specifies:

Field	Type	Description
token	string	ERC-20 token address (MUST match currency)
amount	string	Maximum transfer amount in base units

Table 7

5.2.2. Transfer Details

The transferDetails array **MUST** have the same length as permitted. Each entry specifies:

Field	Type	Description
to	string	Recipient address
requestedAmount	string	Exact transfer amount in base units

Table 8

The first entry corresponds to the primary recipient. Subsequent entries (if any) correspond to split recipients in array order.

5.2.3. Witness Data (Challenge Binding)

The Permit2 witness mechanism provides cryptographic binding between the payment authorization and the challenge. When externalId is present in the challenge request, the client **MUST** include it in the EIP-712 witness struct. The server **MUST** verify the witness matches before submitting the transaction.

The witness type is defined as:

```
struct PaymentWitness {
    bytes32 challengeHash;
}
```

Where challengeHash is computed as:

```
challengeHash = keccak256(abi.encodePacked(
    challenge.id,
    challenge.realm
))
```

This binds the Permit2 signature to the specific challenge instance. The signature cannot be reused against a different challenge, even if the payment parameters are identical.

The witness type string for EIP-712 is: `"PaymentWitness witness)PaymentWitness(bytes32 challengeHash)TokenPermissions(address token,uint256 amount)"`

This specification defines that witness schema directly for challenge binding. Implementations **MUST** use the exact type string above when constructing the EIP-712 typed data.

This binding applies to both single and batch transfers — the same `witness` parameter is used by `permitWitnessTransferFrom()` and `permitBatchWitnessTransferFrom()`.

5.2.4. Server Behavior

For single transfers (no splits, `permitted length` 1), the server calls `permitWitnessTransferFrom()`.

For batch transfers (with splits, `permitted length` > 1), the server calls `permitBatchWitnessTransferFrom()`. This executes all transfers in a single on-chain transaction — if any transfer fails, the entire batch reverts.

The server pays gas from its own balance in both cases. This is the natural fee sponsorship model for Permit2: the client signs only the off-chain authorization and the server handles all chain interaction.

5.2.5. Example: Single Transfer

```
{
  "challenge": {
    "id": "aB3cDeF4gHiJkLmN",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "type": "permit2",
    "permit": {
      "permitted": [
        {
          "token": "0xFAfDdbb3FC7688494971a79cc65DCa3EF82079E7",
          "amount": "1000000000000000000"
        }
      ],
      "nonce": "1",
      "deadline": "1743523500"
    },
    "transferDetails": [
      {
        "to": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
        "requestedAmount": "1000000000000000000"
      }
    ],
    "witness": {
      "challengeHash": "0x8a3b...f1c2"
    },
    "signature": "0x1b2c3d4e5f..."
  },
  "source": "did:pkh:eip155:4326:0x1234...5678"
}
```

5.2.6. Example: Batch Transfer with Splits

```
{
  "challenge": {
    "id": "sPlitBatchEx4mple",
    "realm": "marketplace.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "type": "permit2",
    "permit": {
      "permitted": [
        {
          "token": "0xFAfDdbb3FC7688494971a79cc65DCa3EF82079E7",
          "amount": "1000000000000000000"
        },
        {
          "token": "0xFAfDdbb3FC7688494971a79cc65DCa3EF82079E7",
          "amount": "500000000000000000"
        }
      ]
    },
    "nonce": "1",
    "deadline": "1743523500"
  },
  "transferDetails": [
    {
      "to": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
      "requestedAmount": "1000000000000000000"
    },
    {
      "to": "0x8Ba1f109551bD432803012645Ac136ddd64DBA72",
      "requestedAmount": "500000000000000000"
    }
  ],
  "witness": {
    "challengeHash": "0x7d4e...a3b9"
  },
  "signature": "0x9a8b7c6d5e..."
},
"source": "did:pkh:eip155:4326:0x1234...5678"
}
```

This transfers 1.0 USDm to the primary recipient and 0.05 USDm to the platform — atomically, in a single transaction. The client signs one EIP-712 message covering both transfers.

5.3. Authorization Payload (type="authorization")

Opt-in credential type for tokens that implement EIP-3009 [EIP-3009] (e.g., USDC, EURC). The client signs an off-chain `transferWithAuthorization` message. The server submits it directly to the token contract, paying gas from its own balance. The client never interacts with the chain directly.

Unlike `type="permit2"`, this type requires no prior token approval — EIP-3009 authorization is built into the token contract itself.

Field	Type	Required	Description
<code>type</code>	string	REQUIRED	"authorization"
<code>from</code>	string	REQUIRED	Payer address
<code>to</code>	string	REQUIRED	Recipient address (MUST match challenge recipient)
<code>value</code>	string	REQUIRED	Transfer amount in base units (MUST match challenge amount)
<code>validAfter</code>	string	REQUIRED	Unix timestamp — authorization not valid before this time (stringified integer, typically "0")
<code>validBefore</code>	string	REQUIRED	Unix timestamp — authorization not valid after this time (stringified integer)
<code>nonce</code>	string	REQUIRED	bytes32 nonce — MUST be set to <code>challengeHash</code> (see below)
<code>signature</code>	string	REQUIRED	EIP-712 signature (0x-prefixed)

Table 9

5.3.1. Challenge Binding

The EIP-3009 `nonce` field is a random bytes32 value chosen by the caller. This specification requires the nonce to be set to the `challengeHash`:

```
nonce = keccak256(abi.encodePacked(
    challenge.id,
    challenge.realm
))
```

This provides cryptographic challenge binding equivalent to the Permit2 witness mechanism. The token contract enforces nonce uniqueness on-chain — a nonce that has been consumed cannot be reused, providing replay protection at the contract level.

5.3.2. Constraints

- Splits are NOT supported. Servers **MUST** reject type="authorization" credentials when the challenge includes splits.
- Servers **MUST NOT** advertise "authorization" in credentialTypes unless the currency token is known to implement EIP-3009.
- validBefore **SHOULD** correspond to the challenge expires timestamp.
- to **MUST** match the challenge recipient.
- value **MUST** match the challenge amount.

5.3.3. Example

```
{
  "challenge": {
    "id": "aB3cDeF4gHiJkLmN",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "type": "authorization",
    "from": "0x1234567890abcdef1234567890abcdef12345678",
    "to": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
    "value": "1000000",
    "validAfter": "0",
    "validBefore": "1743523500",
    "nonce": "0x8a3b...f1c2",
    "signature": "0x..."
  },
  "source": "did:pkh:eip155:4326:0x1234567890abcdef1234567890abcdef12345678"
}
```

5.4. Transaction Payload (type="transaction")

The compatible fallback. The client signs a complete ERC-20 transfer transaction targeting the currency contract. The server broadcasts the transaction to the chain. The client pays gas.

This type is intended for chains where Permit2 is not deployed or clients that prefer direct transaction signing.

Field	Type	Required	Description
type	string	REQUIRED	"transaction"
signature	string	REQUIRED	Hex-encoded RLP-serialized signed transaction

Table 10

The signature field contains an EIP-1559 (type 2) transaction, RLP-encoded and hex-prefixed with `0x`. The transaction **MUST** call `transfer(address,uint256)` on the ERC-20 token specified in the challenge.

The client **MUST** sign a fully valid transaction including gas parameters and pay gas from their own balance.

Splits are NOT supported with `type="transaction"`. Servers **MUST** reject `type="transaction"` credentials when the challenge includes splits.

Example:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "signature": "0x02f8...signed transaction bytes...",
    "type": "transaction"
  },
  "source": "did:pkh:eip155:1329:0x1234567890abcdef1234567890abcdef12345678"
}
```

5.5. Hash Payload (type="hash")

Optional fallback for clients that broadcast transactions themselves (e.g., custodial wallets, hardware signers). The client broadcasts a standard ERC-20 transfer transaction to the chain and presents the confirmed transaction hash.

Field	Type	Required	Description
type	string	REQUIRED	"hash"
hash	string	REQUIRED	Transaction hash (0x-prefixed, 32 bytes hex)

Table 11

Constraints:

- Splits are NOT supported. Servers **MUST** reject `type="hash"` credentials when the challenge includes splits.
- The client pays gas.
- The server cannot modify or retry the transaction.
- Weaker challenge binding than other types (see [Section 10.4](#)).

Example:

```

{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "hash":
"0x1a2b3c4d5e6f7890abcdef1234567890abcdef1234567890abcdef1234567890",
    "type": "hash"
  },
  "source": "did:pkh:eip155:4326:0x1234567890abcdef1234567890abcdef12345678"
}

```

5.6. Gas Sponsorship Model

Gas sponsorship is structurally determined by the credential type — no explicit field is needed:

Credential Type	Who Pays Gas	Mechanism
permit2	Server	Client signs off-chain; server constructs and submits tx
authorization	Server	Client signs off-chain; server submits to token contract
transaction	Client	Client signs a full transaction with gas parameters
hash	Client	Client broadcasts the transaction themselves

Table 12

For type="permit2" and type="authorization", the client signs only an off-chain message — they never submit a transaction and therefore cannot pay gas. The server **MUST** maintain sufficient native token balance to cover gas costs for these credential types.

For type="transaction" and type="hash", the client is responsible for gas. The server broadcasts or verifies the transaction but does not subsidize it.

6. 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 "permit2", "authorization", "transaction", or "hash".

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 the challenge includes `splits` and `payload.type` is not "permit2", reject the request. (Only Permit2 supports atomic batch transfers for splits.)
6. Proceed with type-specific verification:
 - For `type="permit2"`: see [Section 6.1](#).
 - For `type="authorization"`: see [Section 6.2](#).
 - For `type="transaction"`: see [Section 6.3](#).
 - For `type="hash"`: see [Section 6.4](#).

6.1. Permit2 Verification

Before submitting, servers **MUST** verify:

1. The EIP-712 signature is valid and recovers to the source address
2. The deadline has not passed
3. The signer has sufficient token balance for the total amount (primary + all splits)
4. The signer has sufficient Permit2 allowance
5. The `witness.challengeHash` matches the expected value derived from the challenge id and realm
6. `permitted` and `transferDetails` arrays have equal length
7. Each `permitted[i].token` matches `currency`
8. `transferDetails[0].to` matches recipient
9. `transferDetails[0].requestedAmount` matches the primary transfer amount (amount minus sum of splits, or amount if no splits)
10. For each split at index `i` (if present), `transferDetails[i+1].to` matches `splits[i].recipient` and `transferDetails[i+1].requestedAmount` matches `splits[i].amount`

After verification:

1. For single transfers (`permitted` length 1): call `Permit2.permitWitnessTransferFrom()`
2. For batch transfers (`permitted` length > 1): call `Permit2.permitBatchWitnessTransferFrom()`
3. Verify the transaction receipt indicates success
4. Verify `Transfer` event logs match all expected transfers

Servers **SHOULD** simulate the transaction via `eth_call` before submitting to detect failures without spending gas.

6.2. Authorization Verification

Before submitting, servers **MUST** verify:

1. The `currency` token implements EIP-3009
2. The EIP-712 signature is valid and recovers to the `from` address
3. `to` matches the challenge recipient
4. `value` matches the challenge amount
5. `nonce` matches the expected `challengeHash` derived from the challenge `id` and `realm`
6. `validBefore` has not passed
7. The signer has sufficient token balance
8. Call `transferWithAuthorization(from, to, value, validAfter, validBefore, nonce, v, r, s)` on the `currency` token contract
9. Verify the transaction receipt indicates success
10. Verify the `Transfer` event log matches the expected parameters

Servers **SHOULD** simulate the transaction via `eth_call` before submitting to detect failures without spending gas.

6.3. Transaction Verification

Before broadcasting, servers **MUST** verify:

1. Deserialize the RLP-encoded transaction from `payload.signature`
2. Verify the transaction `chainId` matches `methodDetails.chainId`
3. Verify the transaction `to` address matches the `currency` token contract
4. Verify the transaction `calldata` begins with the `transfer(address,uint256)` function selector (`0xa9059cbb`)
5. Decode the `calldata` and verify `recipient` and `amount` match the challenge request
6. Broadcast the transaction via `eth_sendRawTransaction`
7. Wait for confirmation and fetch the transaction receipt
8. Verify the receipt `status` is `0x1` (success)
9. Verify the receipt contains a `Transfer` event log matching the challenge parameters

6.4. Hash Verification

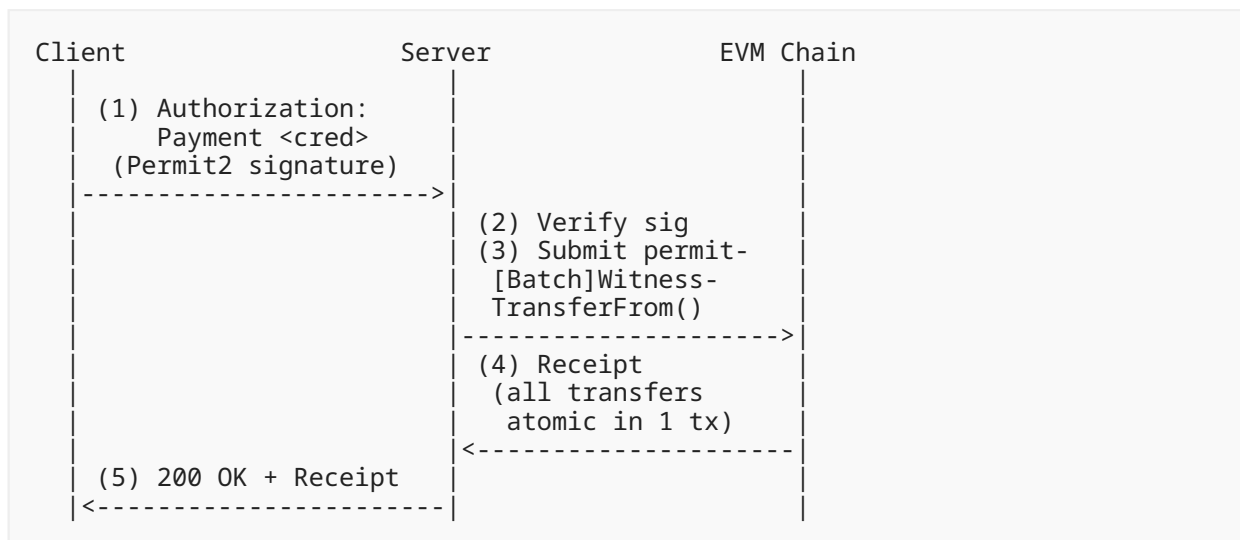
For hash credentials, servers **MUST**:

1. Verify `payload.hash` has not been previously consumed (see [Section 8](#))
2. Fetch the transaction receipt via `eth_getTransactionReceipt`
3. Verify `status` is `0x1` (success)

4. Verify the receipt contains a Transfer event log (topic `0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef`):
 - Log address matches currency
 - to parameter matches recipient
 - value parameter matches amount
5. Mark the hash as consumed

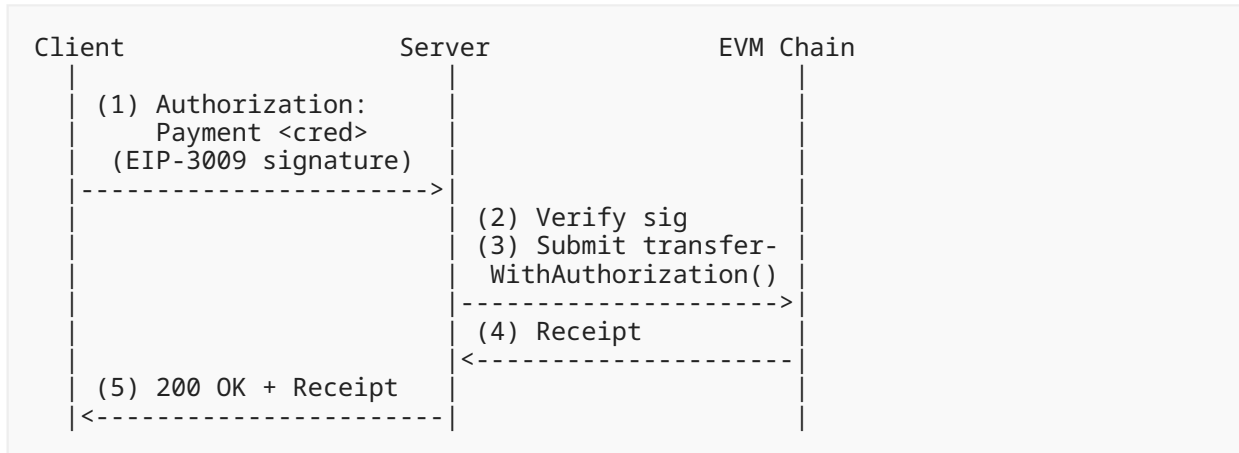
7. Settlement Procedure

7.1. Permit2 Settlement

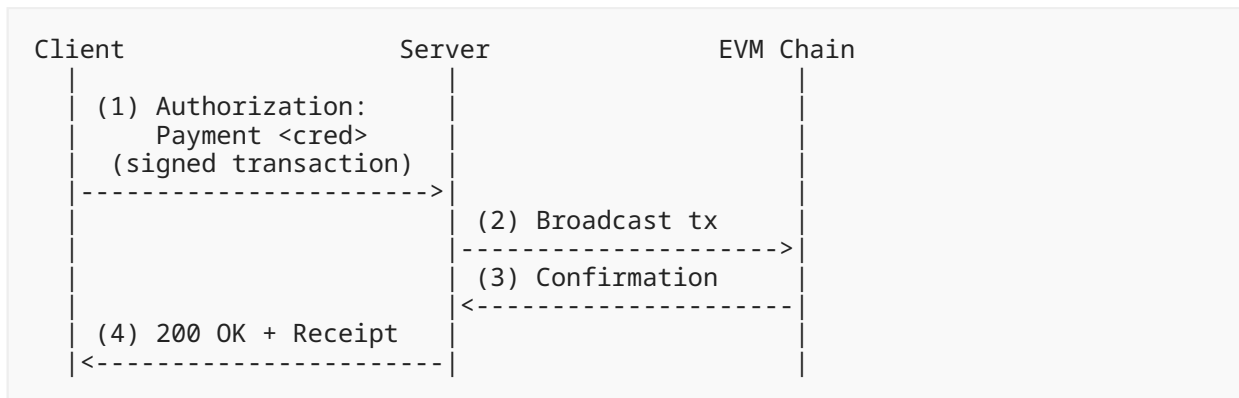


For single transfers, the server calls `permitWitnessTransferFrom()`. When splits are present, the server calls `permitBatchWitnessTransferFrom()`, executing the primary transfer and all splits atomically in a single transaction.

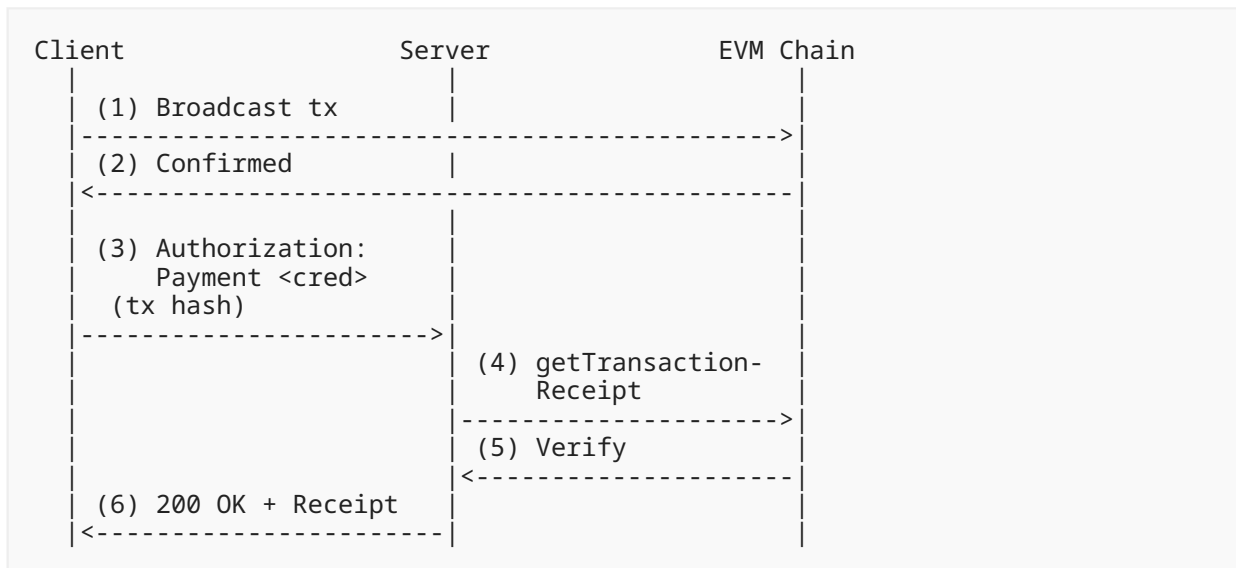
7.2. Authorization Settlement



7.3. Transaction Settlement



7.4. Hash Settlement



7.5. Confirmation Requirements

Servers **MUST** wait for a successful transaction receipt (i.e., the transaction has been included in at least one block) before returning a Payment-Receipt header.

The time between transaction submission and receipt availability varies by chain and current network conditions. Servers **SHOULD NOT** assume a fixed confirmation latency. Servers **MAY** use chain-specific RPC optimizations (e.g., WebSocket subscriptions, synchronous send methods) to minimize wait time.

This specification does not prescribe a required confirmation depth beyond the initial receipt. Finality semantics vary across chains — some offer single-slot finality, while others (including L2 rollups) settle to a separate layer for stronger guarantees. The appropriate confirmation depth for a given transaction is a server policy decision outside the scope of this specification.

7.6. Receipt Generation

Upon successful settlement, servers **MUST** return a Payment-Receipt header per [I-D.httpauth-payment]. Servers **MUST NOT** include a Payment-Receipt header on error responses; failures are communicated via HTTP status codes and Problem Details [RFC9457].

The receipt payload:

Field	Type	Description
method	string	"evm"

Field	Type	Description
challengeId	string	The id from the original challenge
reference	string	Transaction hash (0x-prefixed)
status	string	"success"
timestamp	string	[RFC3339] settlement time
chainId	number	Chain ID where settlement occurred
externalId	string	OPTIONAL. Echoed from the challenge request

Table 13

8. Replay Protection

Servers **MUST** maintain a set of consumed credential identifiers. The replay prevention token depends on the credential type:

- **type="permit2"**: The combination of signer address and Permit2 nonce serves as the replay token. The nonce is consumed on-chain by the Permit2 contract.
- **type="authorization"**: The combination of signer address and EIP-3009 nonce serves as the replay token. The nonce is consumed on-chain by the token contract itself — providing contract-level replay protection.
- **type="transaction"**: The transaction hash (derived after broadcast) serves as the replay token.
- **type="hash"**: The transaction hash provided by the client serves as the replay token.

Before accepting a credential, the server **MUST** check whether its replay token has already been consumed. After successful verification, the server **MUST** atomically mark it as consumed.

9. 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 Problem Details [RFC9457] 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.

Example:

```
{
  "type": "https://paymentauth.org/problems/verification-failed",
  "title": "Transfer Mismatch",
  "status": 402,
  "detail": "Transfer amount does not match challenge request"
}
```

10. Security Considerations

10.1. Transport Security

All communication **MUST** use TLS 1.2 or higher per [I-D.httpauth-payment]. Credentials **MUST** only be transmitted over HTTPS connections.

10.2. Transaction Replay

EIP-1559 transactions include chain ID and nonce, preventing cross-chain and same-chain replay. Permit2 signatures include chain ID in the EIP-712 domain separator and consume nonces on-chain. The `expires` auth-param limits the temporal window for credential use.

10.3. Amount Verification

Clients **MUST** parse and verify the request payload before signing:

1. Verify amount is reasonable for the service
2. Verify currency is the expected token address
3. Verify recipient is controlled by the expected party
4. Verify chainId matches the expected network
5. If `splits` are present, verify the sum of split amounts is strictly less than `amount` and all split recipients are expected

10.4. Hash Credential Binding

Hash credentials (`type="hash"`) and transaction credentials (`type="transaction"`) provide weaker challenge binding than Permit2 credentials. The server verifies that a payment matching the challenge terms exists on-chain, but cannot prove the payment was created for a specific challenge instance.

By contrast, `type="permit2"` and `type="authorization"` credentials include a `challengeHash` — in the EIP-712 witness data (Permit2) or as the on-chain nonce (EIP-3009) — cryptographically binding the signature to the specific challenge `id` and `realm`. This prevents signature reuse across challenges, even if payment parameters are identical. This is a key reason off-chain signature types are preferred.

Servers **MAY** mitigate this by:

- Requiring unique `externalId` values per challenge
- Preferring `type="permit2"` or `type="transaction"` in `credentialTypes`
- Restricting `type="hash"` to low-value transactions

10.5. Permit2-Specific Risks

Allowance Prerequisite: Permit2 requires a one-time `ERC-20 approve()` to the Permit2 contract. Clients should understand they are granting approval to a third-party contract. The Permit2 contract is widely deployed and audited, but clients **SHOULD** verify the contract address matches the canonical deployment.

Nonce Management: Permit2 nonces are consumed on-chain. If a server fails to submit a Permit2 credential, the nonce remains unconsumed and the client can reuse it. Servers **MUST** handle nonce conflicts gracefully.

10.6. Fee Payer Risks

With `type="permit2"`, the server pays gas on every settlement. This creates financial risk:

Denial of Service: Malicious clients could submit credentials that fail on-chain, causing the server to pay gas without receiving payment. Mitigations:

- Simulate transactions via `eth_call` before broadcast
- Rate limit per client address and IP
- Verify client token balance before submitting
- Require client authentication before accepting credentials

Balance Exhaustion: Servers **MUST** monitor their native token balance and reject new requests when insufficient to cover gas.

Gas costs vary significantly across EVM chains. On low-fee chains, fee sponsorship is negligible (<\$0.001/tx). On Ethereum L1, gas costs may be significant and servers **SHOULD** factor this into pricing.

10.7. Split Payment Risks

Recipient Transparency: Clients **SHOULD** present each split recipient and amount so the user can verify the payment distribution. Clients **SHOULD** highlight when the primary recipient receives a small remainder relative to the total amount.

Batch Failure: With Permit2 batch transfers, splits are atomic — all succeed or all revert. A failure in any split causes the entire payment (including the primary transfer) to revert. Servers **SHOULD** simulate the batch via `eth_call` before submitting to detect failures early.

10.8. RPC Trust

Servers rely on their RPC endpoint for transaction data. A compromised RPC could return fabricated data. Servers **SHOULD** use trusted RPC providers or run their own nodes.

11. IANA Considerations

11.1. Payment Method Registration

This document registers the following payment method in the "HTTP Payment Methods" registry established by [I-D.httpauth-payment]:

Method Identifier	Description	Reference
evm	EVM-compatible blockchain ERC-20 token transfer	This document

Table 14

Contact: Brett DiNovi (bread@megaeth.com)

11.2. Payment Intent Registration

This document registers the following payment intent in the "HTTP Payment Intents" registry established by [I-D.httpauth-payment]:

Intent	Applicable Methods	Description	Reference
charge	evm	One-time ERC-20 token transfer on any EVM chain	This document

Table 15

12. References

12.1. 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.httppauth-payment] Moxey, J., "The 'Payment' HTTP Authentication Scheme", January 2026, <<https://datatracker.ietf.org/doc/draft-ietf-httppauth-payment/>>.

12.2. Informative References

- [EIP-712] Bloemen, R., "Typed structured data hashing and signing", September 2017, <<https://eips.ethereum.org/EIPS/eip-712>>.
- [EIP-1559] "Fee market change for ETH 1.0 chain", n.d., <<https://eips.ethereum.org/EIPS/eip-1559>>.
- [EIP-55] "Mixed-case checksum address encoding", n.d., <<https://eips.ethereum.org/EIPS/eip-55>>.
- [ERC-20] "Token Standard", n.d., <<https://eips.ethereum.org/EIPS/eip-20>>.
- [EIP-3009] Kim, P. J., "Transfer With Authorization", September 2019, <<https://eips.ethereum.org/EIPS/eip-3009>>.
- [PERMIT2] Uniswap Labs, "Permit2", n.d., <<https://github.com/Uniswap/permit2>>.

Appendix A. Full Example: Permit2 Charge on MegaETH

1. Challenge (402 response):


```
{
  "challenge": {
    "id": "aB3cDeF4gHiJkLmN",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "type": "permit2",
    "permit": {
      "permitted": [
        {
          "token": "0xFAfDdbb3FC7688494971a79cc65DCa3EF82079E7",
          "amount": "1000000000000000000"
        }
      ],
      "nonce": "1",
      "deadline": "1743523500"
    },
    "transferDetails": [
      {
        "to": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
        "requestedAmount": "1000000000000000000"
      }
    ],
    "witness": {
      "challengeHash": "0x8a3b...f1c2"
    },
    "signature": "0x1b2c3d4e5f..."
  },
  "source": "did:pkh:eip155:4326:0x1234...5678"
}
```

3. Response (with receipt):

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

{"response": "resource data"}
```

Decoded receipt:

```
{
  "method": "evm",
  "challengeId": "aB3cDeF4gHiJkLmN",
  "reference": "0xabc123...",
  "status": "success",
  "timestamp": "2026-04-01T12:04:58Z",
  "chainId": 4326
}
```

Appendix B. Full Example: Transaction Charge on Sei

Challenge requests 1.0 USDC on Sei (chain 1329):

```
{
  "amount": "1000000",
  "currency": "0xe15fc38f6d8c56af07bbcbe3baf5708a2bf42392",
  "recipient": "0x742d35Cc6634C0532925a3b844Bc9e7595f8fE00",
  "description": "Premium API call",
  "methodDetails": {
    "chainId": 1329
  }
}
```

Credential (signed EIP-1559 transaction):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "evm",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-04-01T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "signature": "0x02f8...signed transaction bytes..."
  },
  "source": "did:pkh:eip155:1329:0x1234567890abcdef1234567890abcdef12345678"
}
```

Appendix C. Acknowledgements

The authors thank Georgios Konstantopoulos for guidance on consolidating chain-specific specs into a unified EVM method, Brendan Ryan and Jake Moxey at Tempo Labs for the MPP framework, and the Sei and MegaETH communities for their contributions to earlier drafts.

Authors' Addresses

Brett DiNovi

MegaETH Labs

Email: bread@megaeth.com**Conner Swenberg**

Coinbase

Email: conner.swenberg@coinbase.com**Kyle Scott**

Monad Foundation

Email: kscott@monad.foundation