ZeroProof AI

ZKTLS ATTESTATION · INTEGRATION SKILL

zkFetch

Integrate the ZeroProof zkFetch API: generate and verify zkTLS proofs of HTTPS requests with selective disclosure. Written for coding agents and developers. The service is a REST API; call it with plain HTTP from any language.

Integrating the ZeroProof zkFetch API

zkFetch produces a cryptographic proof that a specific HTTPS request was made and returned a specific response, without trusting the party who made the request. Secrets (API keys, tokens) stay hidden; chosen response fields can be selectively revealed or redacted. Proofs verify off-chain in milliseconds and can be transformed for on-chain (EVM) verification.

This file is the complete integration reference. The service is a REST API; call it with plain HTTP from any language. No SDK is required.

Base URL: the zkFetch deployment you call, referred to as `$ZKFETCH_URL` below. The service runs the attestor connection and Reclaim configuration for you; there is nothing to request and no credentials to manage. No API key is required on the API itself.

Critical: always confirm you are getting real proofs

The service has a mock mode. If its Reclaim credentials are missing or misconfigured, `/zkfetch` still returns `success: true` with a FAKE proof. A mock proof looks structurally real but proves nothing.

Rules for every integration, no exceptions:

1. On startup, call `GET /health` and require `"mode": "production"`. If it reports `"mode": "mock"`, fail hard; do not proceed.
2. After every `/zkfetch` call, reject the result if `response.metadata.mock === true` or `response.verified !== true`.
3. Never treat `success: true` alone as evidence of a valid proof.

```
const health = await fetch(`${ZKFETCH_URL}/health`).then(r => r.json());
if (health.mode !== 'production') {
  throw new Error(`zkFetch is in ${health.mode} mode; refusing to run`);
}
```

Endpoints

Endpoint	Purpose
`GET /health`	Service status and mode (production vs mock)
`POST /zkfetch`	Make a proven HTTPS request; returns data + proof
`POST /verify`	Verify a proof's attestor signatures (fast, primary check)
`POST /verify-full`	Signatures plus Groth16 zk-SNARK verification
`POST /transform-onchain`	Reshape a proof for EVM smart-contract submission

POST /zkfetch

Request body:

```
{
  "url": "https://api.example.com/endpoint",
  "publicOptions": {
    "method": "POST",
    "headers": { "Content-Type": "application/json" },
    "body": { "from": "NYC", "to": "LAX" }
  },
  "privateOptions": {
    "headers": { "Authorization": "Bearer YOUR_SECRET" },
    "responseMatches": [
      { "type": "regex", "value": "\"status\"\\s*:\\s*\"(?<status>[A-Z]+)\"" }
    ]
  },
  "redactions": [
    { "jsonPath": "$.booking_id" }
  ]
}
```

- `url` (required): the HTTPS endpoint to fetch with proof.

- `publicOptions` (optional): method, headers, body. These are visible to anyone who inspects the proof. Default method is GET.
- `privateOptions.headers` (optional): secrets the target API needs. These are cryptographically hidden from proof verifiers. Put every credential here, never in `publicOptions`.
- `privateOptions.responseMatches` (optional): what to extract and REVEAL.
- `redactions` (optional): what to HIDE from the response. Top-level field, not inside `privateOptions`.

Response (200):

```
{
  "success": true,
  "data": "parsed response body after redaction",
  "proof": "full proof object, store this",
  "onchainProof": "proof reshaped for smart contracts",
  "verified": true,
  "metadata": {
    "timestamp": 1735603200000,
    "url": "https://api.example.com/endpoint",
    "method": "POST",
    "onchain_compatible": true
  }
}
```

Store the entire `proof` object verbatim; it is what verifiers need. Errors return 400 (missing url) or 500 with `{ "success": false, "error" }`.

POST /verify

Body: `{ "proof": <stored proof object> }` Response: `{ "success": true, "valid": true, "extractedData": { ... } }`

Signature verification of the attestor chain. This is the standard check. `extractedData` is only returned when the proof is valid.

POST /verify-full

Body: `{ "proof": <proof>, "options": {} }` Adds Groth16 zk-SNARK verification of correct transcript decryption on top of signature checks. Slower; use when the application demands maximum assurance. If the SNARK layer is unavailable the signature result is still authoritative.

POST /transform-onchain

Body: `{ "proof": <proof> }` Response: `{ "success": true, "onchainProof": { ... } }` Produces the ABI-ready shape expected by Reclaim verifier contracts on EVM chains (Ethereum, Base, Optimism, Polygon, zkSync Era). Note that `/zkfetch` already returns `onchainProof`; this endpoint exists to transform proofs you stored earlier.

Selective disclosure

Two independent mechanisms; use them together.

Reveal with `privateOptions.responseMatches`. Use regex with NAMED capture groups; each named group becomes a revealed value in `proof.extractedParameterValues`:

```
{ "type": "regex", "value": "\"usd\":(?<price>[0-9.]+)" }
```

Given `{"ethereum":{"usd":3631.24}}`, the proof reveals `price: "3631.24"` and nothing else. `{ "type": "contains", "value": "APPROVED" }` proves a substring is present without extracting anything. Prefer these two types; do not rely on a `jsonPath` type inside `responseMatches`.

Hide with `redactions`. Each entry is one of:

```
{ "jsonPath": "$.user.email" }
{ "regex": "\"passenger_name\"\\s*:\\s*\".*?\"" }
{ "xPath": "//creditCard" }
```

JSONPath supports nesting (`$.a.b`), recursive descent (`$..email`), and array wildcards (`$.items[*]`). Redacted values are cryptographically hidden, not replaced: there is no `replacement` option, and any `replacement` field you send is silently discarded.

To prove "the call succeeded" while hiding the entire response body: `"redactions": [ { "regex": ".*" } ]`.

Recommended integration pattern

```
async function provenFetch(zkfetchUrl, request) {
  const res = await fetch(`${zkfetchUrl}/zkfetch`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(request),
  });
  const out = await res.json();
  if (!out.success) throw new Error(`zkfetch failed: ${out.error}`);
  if (out.metadata?.mock) throw new Error('zkfetch returned a MOCK proof');
  if (out.verified !== true) throw new Error('proof failed local verification');
  return out; // { data, proof, onchainProof }
}
```

Later, anyone (including a counterparty) confirms the proof:

```
const { valid, extractedData } = await fetch(`${zkfetchUrl}/verify`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ proof: storedProof }),
}).then(r => r.json());
```

Common mistakes

- Trusting `success: true` without checking `metadata.mock` and `verified`.
- Putting credentials in `publicOptions.headers` (they become publicly visible in the proof). Secrets belong in `privateOptions.headers`.
- Placing `redactions` inside `privateOptions`; it is a top-level field.
- Unnamed regex capture groups: only NAMED groups `(?<name>...)` are revealed.
- Single-escaped regex in JSON: `\s` must be written `\\s` inside a JSON string.
- Expecting redaction to substitute placeholder text; redaction hides bytes, it does not replace them.
- Re-verifying with `/verify-full` in hot paths; use `/verify` for routine checks.

Troubleshooting

- **`Provider returned error 5xx`**: the TARGET endpoint returned that status to the attestor, which fetches it server-side. This is the upstream API being down or rate-limited, not a zkFetch fault, and it can differ from what you see from your own IP. Retry, or prove against a stable endpoint. A proof is only valid over a real 2xx response.

- `Method/Host mismatch, or first 2 headers were not Host and Connection`: the attestor could not reconstruct a clean request line for this target. Prefer simple HTTP/1.1 JSON endpoints, set `publicOptions.method` explicitly, and avoid targets that redirect or require unusual header handling.
- `success: true` **but** `mode: mock`: Reclaim credentials are not loaded. Fix the server config; never ship mock proofs. Always gate on `GET /health` reporting `"mode": "production"` first.
- On-chain tests need a funded testnet wallet key and an RPC URL configured server-side; without them the on-chain path is skipped, not broken.